

TQS: Quality Assurance manual

Miguel Miragaia [108317], Vasco Faria [107323], Cristiano Nicolau [108536], André Gomes [97541]
v2024-05-31

1	Project management	1
1.1	Team and roles	1
1.2	Agile backlog management and work assignment	1
2	Code quality management	2
2.1	Guidelines for contributors (coding style)	2
2.2	Code quality metrics	2
3	Continuous delivery pipeline (CI/CD)	2
3.1	Development workflow	2
3.2	CI/CD pipeline and tools	2
3.3	System observability	2
4	Software testing	2
4.1	Overall strategy for testing	2
4.2	Functional testing/acceptance	3
4.3	Unit tests	3
4.4	System and integration testing	3

1 Project management

1.1 Team and roles

NAME	ROLE
Miguel Miragaia	Team Leader
Vasco Faria	Product Owner
Cristiano Nicolau	DevOps Master
André Gomes	QA Engineer
Miguel Miragaia Vasco Faria Cristiano Nicolau	Developer

1.2 Agile backlog management and work assignment

Jira was used as the backlog management tool. The Team Manager created the sprint board, which operated on a weekly cycle.

Initially, in a team meeting, we developed the User Stories, which were later added to the backlog by the Team Manager. Epics were created related to each project springboot, and User Stories were added to the sprint board as Issues linked to these Epics. Each Issue had associated tasks, allowing us to break down the User Story into smaller, manageable tasks that could be distributed among the team.

At the start of each sprint, the team, with the help of the Product Owner, prioritized the tasks. After assigning the tasks to different developers and setting their status to "In Progress" by the Project Manager, each developer created a new branch including the SCRUM ID of the User Stories/Tasks (e.g., SCRUM-11). Each commit included this ID to manage branches and commits associated with each task in Jira.

Upon completing the task, the responsible developer created a pull request to the "dev" branch, which was reviewed and approved by another team member. Once the User Story was completed, the Project Manager updated its status to "Done."

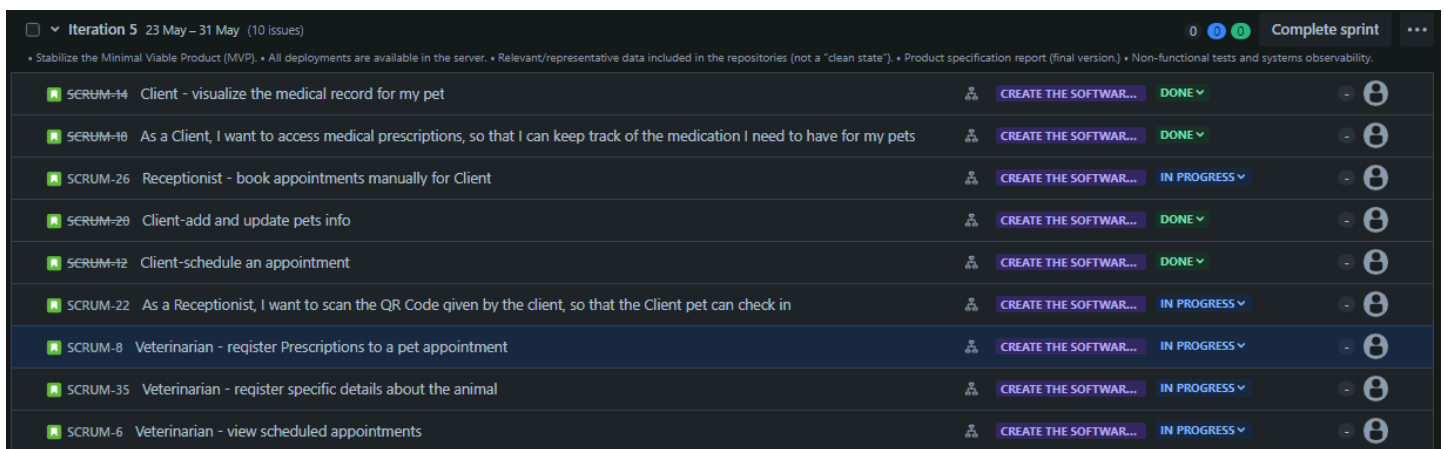


Image 1 - Jira: Backlog Management

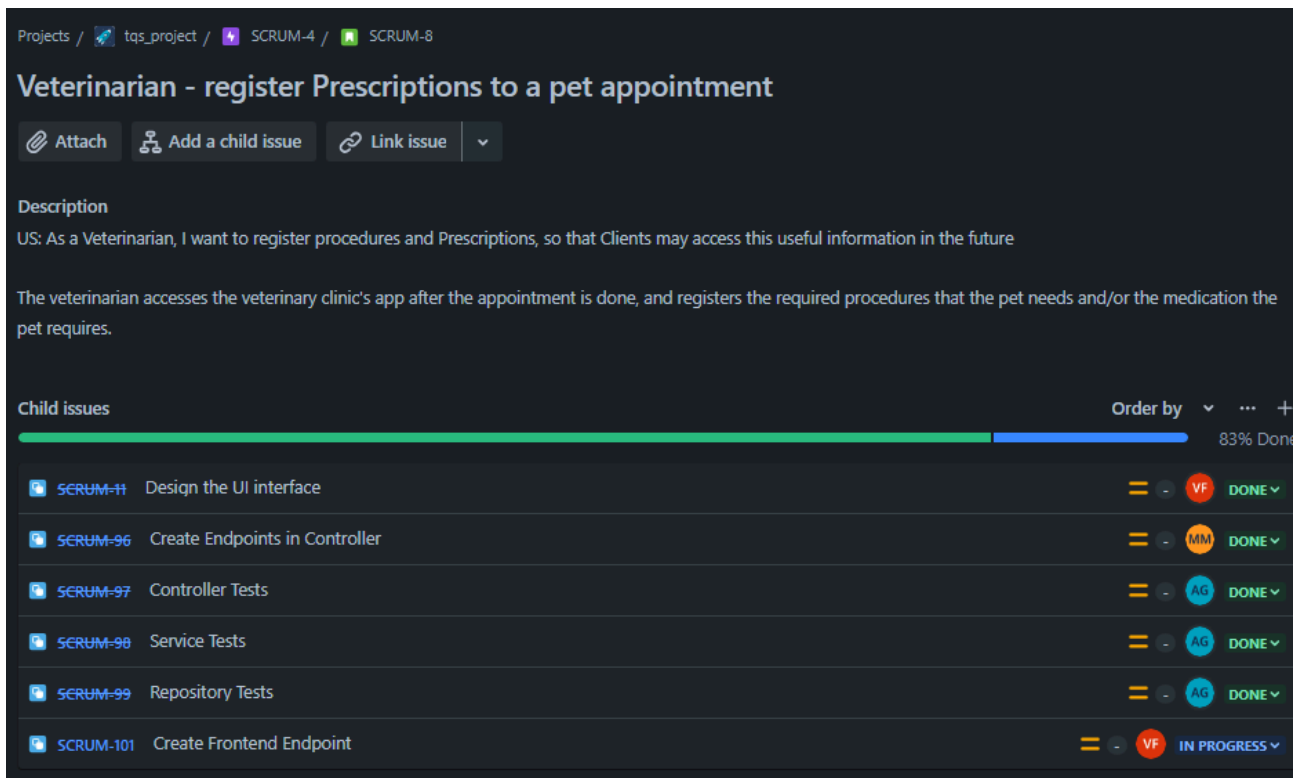


Image 2 - Jira: User Story and Tasks

2 Code quality management

2.1 Guidelines for contributors (coding style)

In order to apply the best possible coding conventions, we followed the code style guide developed by the Android team: [Android Code Style Guide](#).

This guide contains rules and conventions for the java language, which makes the majority of our project, and presents both good and bad practical usage examples.

2.2 Code quality metrics and dashboards

Static code analysis is a method for debugging code before the program is run. During this analysis, the written code is compared with general rules of code writing. To do this, we use SonarQube, more specifically SonarCloud - an open-source platform for conducting automatic reviews and static code analysis. This tool not only detects code with bugs but also identifies bad smells and security vulnerabilities. In addition, SonarCloud allows for continuous code analysis, so it not only shows us the state of the application we are developing but also shows us the problems introduced by each increment produced.

We can also define quality gates for our project. These are nothing more than metrics for integrating new code into the project under development. If a new code submission does not pass through a

quality gate, this code will have to be redone until it meets the defined standards. In this topic, the defined quality gates were:

- Code duplication
- Coverage
- Maintainability
- Reliability
- Security

The chosen platform also has a significant advantage when it comes to continuous integration. Since SonarCloud easily integrates with GitHub CI/CD, it is quite simple to automate this static code analysis so that it is performed continuously.

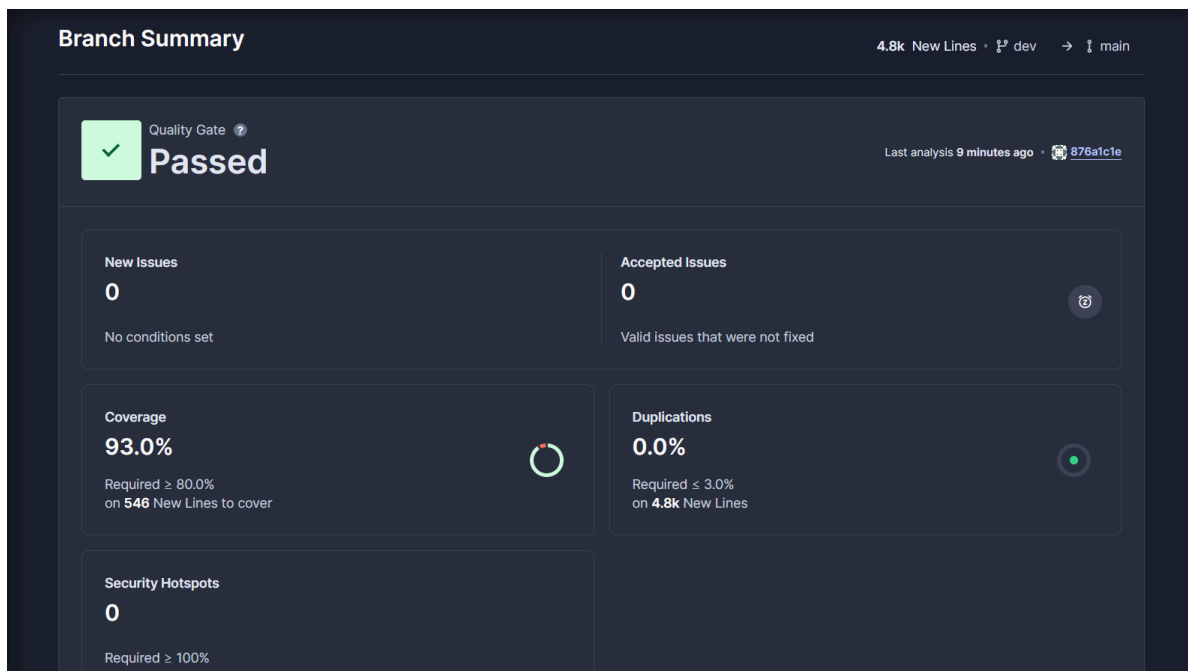


Image 3 - Sonar Analysis Results and Metrics

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

In our project, we adopted the GitHub Flow as our workflow methodology. GitHub Flow is a lightweight, branch-based workflow that allows for easy collaboration and continuous delivery.

The GitHub Flow involves the following steps:

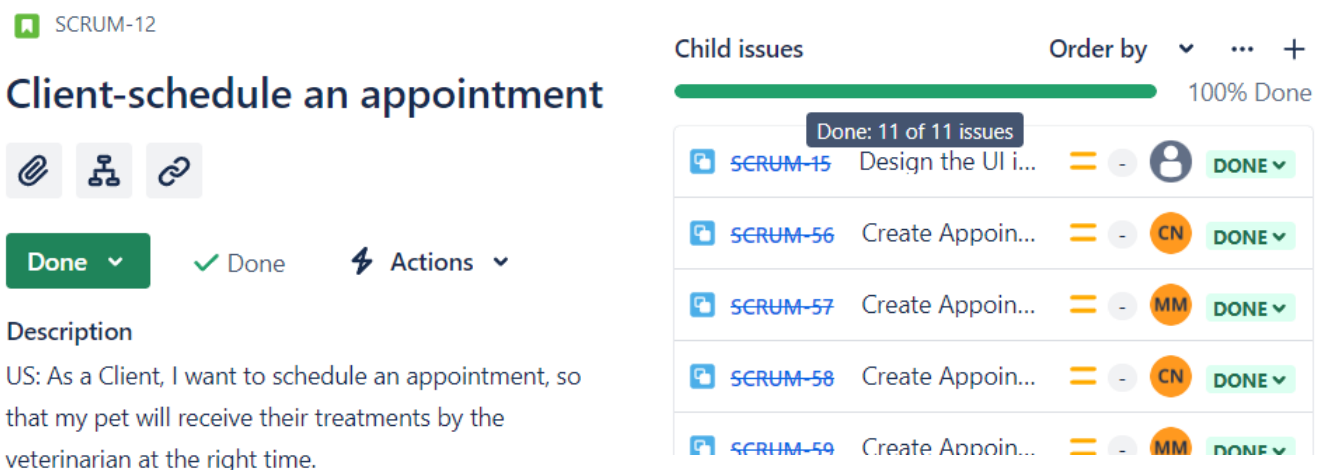
1. **Create a Branch:** Each feature begins by creating a new branch in the repository.
2. **Add Commits:** Developers make commits to the branch as they work on the feature, ensuring that changes are well-documented.
3. **Open a Pull Request:** When the work is complete, a pull request is opened to merge the changes into the main branch (in our case, 'dev' branch).

4. **Discuss and Review:** Team members review the pull request, and, if everything seems correct to the team members, the pull request is accepted; otherwise, it's declined.
5. **Merge and Deploy:** Once the pull request is approved, it is merged into the main branch, and the changes are deployed to production.

GitHub Flow aligns well with our user stories by providing a clear structure for developing and integrating new features. Each user story corresponds to a branch in the repository, allowing us to focus on one feature at a time and ensure that it is fully tested and reviewed before merging into the main codebase. This approach promotes collaboration, transparency, and continuous improvement throughout the development process.

In Agile Software Development, we use the User Story Definition Of Done (DOD) for User Stories to ensure the quality of work and to assess whether the team completes a User Story or not. In our project, we could ensure a User Story was 'done' when:

- All the child issues associated with that user story are done;
- We can ensure that the user story reflects into a fully functional method in our project



The screenshot shows a Jira user story titled "Client-schedule an appointment" (SCRUM-12). The story is marked as "Done" with a green checkmark. Below the title, the description reads: "US: As a Client, I want to schedule an appointment, so that my pet will receive their treatments by the veterinarian at the right time." To the right, a "Child issues" panel shows a progress bar at "100% Done" and a list of 11 child issues, all of which are also marked as "DONE".

Issue ID	Description	Status
SCRUM-15	Design the UI i...	DONE
SCRUM-56	Create Appoin...	DONE
SCRUM-57	Create Appoin...	DONE
SCRUM-58	Create Appoin...	DONE
SCRUM-59	Create Appoin...	DONE

Images 4 and 5 - User Story Example

Here, is this example, we can ensure that the User Story "Client - Schedule an Appointment" is done since we have completed all the child issues, and we can ensure, both by using test code and running our web app, that we, in fact, can, as a client, schedule an appointment.

3.2 CI/CD pipeline and tools

Regarding the CI pipeline, in our project, we used Github Actions in order to run the Spring Boot tests, as well as analyzing the code using SonarQube.

GitHub Actions provides a robust platform for automating workflows directly within a GitHub repository, and, by incorporating SonarQube, which automatically triggers code analysis on each commit or pull request, the results are reported back in the GitHub interface, where we can check whether our requirements were met.

We configured SonarCloud for our client and got the feedback provided in figure 3. Later, when

configuring SonarCloud for all of our services, we realized we had to delete our previous SonarCloud project, in order to configure a new one that would work for all of our services (monorepository). We had a lot of configuration issues and could not finish this milestone.

```
1  name: SonarCloud
2  on:
3    push:
4      branches:
5        - main
6        - dev
7    pull_request:
8      types: [opened, synchronize, reopened]
9      branches:
10         - dev
11  jobs:
12    build:
13      name: Build and analyze
14      runs-on: ubuntu-latest
15      steps:
16        - uses: actions/checkout@v3
17          with:
18            fetch-depth: 0 # Shallow clones should be disabled for a better relevancy of analysis
19        - name: Set up JDK 17
20          uses: actions/setup-java@v3
21          with:
22            java-version: 17
23            distribution: 'zulu' # Alternative distribution options are available.
24        - name: Cache SonarCloud packages
25          uses: actions/cache@v3
26          with:
27            path: ~/.sonar/cache
28            key: ${{ runner.os }}-sonar
29            restore-keys: ${{ runner.os }}-sonar
30
31        - name: Cache Maven packages
32          uses: actions/cache@v3
33          with:
34            path: ~/.m2
35            key: ${{ runner.os }}-m2-${{ hashFiles('**/pom.xml') }}
36            restore-keys: ${{ runner.os }}-m2
37        - name: Build and analyze
38          env:
39            GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }} # Needed to get PR information, if any
40            SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
41          run: |
42            docker-compose up -d
43            cd projPet/client
44            mvn -B verify
45            mvn org.sonarsource.scanner.maven:sonar-maven-plugin:sonar \
46              -Dsonar.projectKey=peticket_PeTicket \
47              -Dsonar.coverage.jacoco.xmlReportPaths=target/site/jacoco/jacoco.xml \
48              sonar:sonar
```

Continuous Delivery (CD) is an essential practice in modern software development, enabling teams to efficiently, quickly, and reliably deliver code changes.

In the Peticket project, this practice is supported by a series of tools and practices, including code versioning and management through Git, regular test automation to ensure code stability and quality, and the use of GitHub Actions in conjunction with Docker for automated deployment of the application in test and production environments.

Docker containers provide a flexible and portable infrastructure, allowing for the creation of isolated and replicable environments for running the application at different stages of the continuous delivery process. In our application, each of the services, both front-end and back-end, is encapsulated in a Docker container. Additionally, in each of the back-end services, there is a Dockerfile that specifies the

configuration required to build the corresponding Docker container. This approach ensures that each component of the application is isolated and executed consistently, regardless of the environment in which it is being deployed.

Our project's continuous delivery pipeline is automatically triggered after changes in the "dev" branch, ensuring efficient and reliable delivery of new features and bug fixes to end users on our server.

```

version: '3.2'

services:
  mysql-container:
    image: mysql
    container_name: mysql-container
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: peticket
      MYSQL_USER: peticket
      MYSQL_PASSWORD: peticket
    ports:
      - "3306:3306"
    networks:
      - peticket-network

  rabbitmq-container:
    image: "rabbitmq:3.8.0-management"
    container_name: rabbitmq-container
    ports:
      - 5672:5672
      - 15672:15672
    environment:
      RABBITMQ_DEFAULT_USER: guest
      RABBITMQ_DEFAULT_PASS: guest
    networks:
      - peticket-network

  funcionario-service:
    depends_on:
      - mysql-container
    build:
      context: ../projClinicFunc/func
    restart: always
    ports:
      - "8082:8082"
    networks:
      - peticket-network

  vet-service:
    depends_on:
      - mysql-container
      - rabbitmq-container
    build:
      context: ../projVet/vet
    restart: always
    ports:
      - "8081:8081"
    environment:
      - SPRING_DATASOURCE_URL=jdbc:mysql://mysql-container:3306/peticket
      - SPRING_DATASOURCE_USERNAME=peticket
      - SPRING_DATASOURCE_PASSWORD=peticket
      - SPRING_JPA_HIBERNATE_DDL_AUTO=update
      - SPRING_RABBITMQ_HOST=rabbitmq-container
    networks:
      - peticket-network

  pet-service:
    depends_on:
      - mysql-container
      - rabbitmq-container
    build:
      context: ../projPet/client
    restart: always
    ports:
      - "8080:8080"
    environment:
      - SPRING_DATASOURCE_URL=jdbc:mysql://mysql-container:3306/peticket
      - SPRING_DATASOURCE_USERNAME=peticket
      - SPRING_DATASOURCE_PASSWORD=peticket
      - SPRING_JPA_HIBERNATE_DDL_AUTO=update
    networks:
      - peticket-network

```



```

display-service:
  depends_on:
    - mysql-container
    - rabbitmq-container
    - pet-service
  build:
    context: ../projDS/display
  restart: always
  ports:
    - "8888:8888"
  environment:
    - SPRING_DATASOURCE_URL=jdbc:mysql://mysql-container:3306/peticket
    - SPRING_DATASOURCE_USERNAME=peticket
    - SPRING_DATASOURCE_PASSWORD=peticket
    - SPRING_JPA_HIBERNATE_DDL_AUTO=update
  networks:
    - peticket-network

pet-frontend:
  depends_on:
    - pet-service
  build: ../projPet/frontend/
  ports:
    - "3001:80"
  networks:
    - peticket-network

vet-frontend:
  depends_on:
    - vet-service
  build: ../projVet/frontend/
  ports:
    - "3002:80"
  networks:
    - peticket-network

func-frontend:
  depends_on:
    - funcionario-service
  build: ../projClinicFunc/frontend/
  ports:
    - "3003:80"
  networks:
    - peticket-network

display-frontend:
  depends_on:
    - display-service
  build: ../projDS/frontend/
  ports:
    - "3004:80"
  networks:
    - peticket-network

networks:
  peticket-network:
    driver: bridge

```

4 Software testing

4.1 Overall strategy for testing

This section is dedicated to documenting the testing strategy used during our project. Although we started this project aiming to always use Test-Driven Development, we later realized that in order to meet some of the deadlines, we had to prioritize delivering functional code over developing the tests.

Despite this, when time wasn't of urgency, we followed TDD and got the test code ready before proceeding with the project development.

Technologically speaking, most of our test code uses JUnit and Mockito in order to test our REST API. We also tried implementing functional tests using Selenium/Cucumber, but due to the lack of time, these tests were incomplete.

4.2 Functional testing/acceptance

Functional testing, or acceptance testing, ensures the system meets user requirements by testing from a user perspective without knowing the internal workings. Our project policy mandates closed-box tests, focusing on outputs based on inputs, to emulate real user interactions. Our goal was to implement these tests for all of our project parts, but due to these tests only being possible to run after our system being complete, and the lack of time and urgency to fix functional requests, we only could implement a small amount of this type of tests. Nevertheless, we made sure to manually test all of our functionalities and ensure there were no errors, and when there were, to deal with them and test again.

4.3 Unit tests

To ensure that every functionality, no matter how minimal, does not disrupt the proper operation of the application and functions as intended, we subject them to unit tests. Each unit test executes the source code of only one specific functionality, hence the term "unit" test. This approach follows the white-box testing methodology, which aims to test the internal structure of an application.

```
@Test
@DisplayName("Test save appointment")
void testSaveAppointment() {
    when(appointmentRepository.save(appointment1)).thenReturn(appointment1);
    assertThat(appointmentService.save(appointment1)).isEqualTo(appointment1);
}
```

Image 6 - Example of an Unit Test

4.4 System and integration testing

To ensure that different components of the application work together seamlessly and do not disrupt overall functionality, we subject them to integration tests. Each integration test evaluates the interactions between multiple components or systems, ensuring they function collectively as intended. This approach follows the black-box testing methodology, which focuses on testing the application's behavior and interactions from an external perspective without considering its internal structure.

```
@Test
void whenGetAppointmentById_thenReturnAppointment() throws Exception {
    UUID randomId = UUID.randomUUID();
    when(authHandler.getUserId()).thenReturn(userId1);
    when(aptmService.findById(randomId)).thenReturn(aptm1);

    mvc.perform(get("/api/client/appointment/by-id/id?id=" + randomId)
        .contentType(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andExpect(jsonPath(expression: "$.date", is(value: "30/05/2024")));

    verify(aptmService, times(wantedNumberOfInvocations:1)).findById(randomId);
}
```

Image 7 - Example of an Integration Test