

# TQS: Quality Assurance manual

*Maria Rafaela [107658], Marta Oliveira [107826], Gabriel Teixeira [107876], Fabio Matias [108011]*

v2024-06-03

<b>1 Project management.....</b>	<b>2</b>
1.1 Team and roles.....	2
1.2 Agile backlog management and work assignment.....	2
<b>2 Code quality management.....</b>	<b>3</b>
2.1 Guidelines for contributors (coding style).....	3
2.2 Code quality metrics and dashboards.....	3
<b>3 Continuous delivery pipeline (CI/CD).....</b>	<b>5</b>
3.1 Development workflow.....	5
3.2 CI/CD pipeline and tools.....	6
<b>4 Software testing.....</b>	<b>6</b>
4.1 Overall strategy for testing.....	6
4.2 Functional testing/acceptance.....	6
4.3 Unit tests.....	7
4.4 System and integration testing.....	7
4.5 Performance testing [Optional].....	8

# 1 Project management

## 1.1 Team and roles

- **Maria Rafaela Abrunhosa - Team Manager + Developer/Tester**

As Team Manager, Maria has the responsibility to oversee the project's progress, ensuring that the team fulfills each iteration with the best results, and coordinating between the rest of the team members. Her role as a Developer/Tester means she's also directly involved in coding and quality assurance.

- **Marta Oliveira Inácio - DevOps Master + Developer/Tester**

Marta's role as a DevOps Master involves managing the CI/CD (Continuous Integration/Continuous Deployment) pipelines, ensuring that software development practices integrate with system operations smoothly. Additionally, her role in development and testing helps her ensure that operational perspectives are considered during the software development phase.

- **Gabriel Melo Teixeira - Software Architect + Developer/Tester**

As a Software Architect, Gabriel is responsible for the overall design of the software system, selecting appropriate technologies, and ensuring that the system is scalable and maintainable. His role as a Developer/Tester means he's also directly involved in coding and quality assurance.

- **Fábio António Teixeira Matias - Product Owner + Developer/Tester**

Fábio acts as the Product Owner, which involves defining the product vision, managing the product backlog, and ensuring that the development team delivers value to the business. His role as a Developer/Tester means he's also directly involved in coding and quality assurance.

## 1.2 Agile backlog management and work assignment

The backlog management was made through the Jira platform.

Due to the agile practices, during the development phase, the project development was guided by the conclusion of user stories since the value of the product and the work is dictated by the number of story points concluded.

The backlog management was divided by the creation of tasks, user stories, tests and all of them were assigned to one or more developers. The work assignment was not entirely decided by the team roles but by the facility and knowledge a member had about a certain task.

The first step was to make a list of user stories and add them to the project backlog. Then, and after some discussion with the professor, the user stories were prioritized using a system of points, the most important user stories had the higher number of story points associated.

We had some sprints and the user stories were added to them according to their value. The sprints had the duration of one week so the user stories had the ideal maximum time of that sprint. The developers were assigned to some user stories and were in charge of developing their assignments.

After the development of the user stories and their testing and quality assurance, the user story is concluded to the dev branch with a pull request.

All the team checks that the acceptance criteria initially defined are met and, if all agree, the user story is approved and marked as "Done".

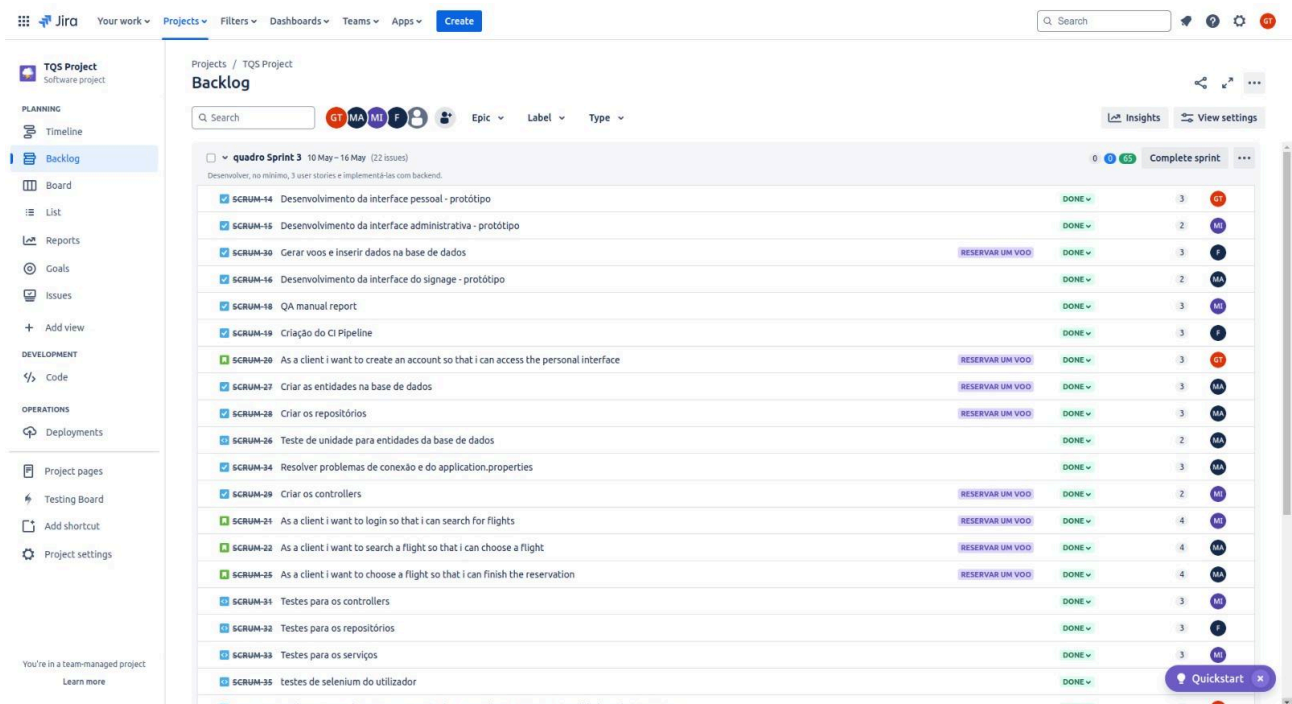


Figure 1: Backlog no Jira

## 2 Code quality management

### 2.1 Guidelines for contributors (coding style)

For the frontend we tried to use the library ReactJs for some good programming practices. We tried to use this guideline coding style: <https://github.com/airbnb/javascript/tree/master/react>.

For some good practices with Java, we tried to follow the coding style defined here: <https://source.android.com/docs/setup/contribute/code-style>.

### 2.2 Code quality metrics and dashboards

Static code analysis is a method of debugging code without executing the program.

We used SonarQube to test this code quality because it detects bugs, bad smells, issues and security vulnerabilities. Beyond that, the sonarQube enables continuous code analysis. The continuous analysis shows the analysis for the new code added to the project. We can also define quality gates, metrics for integrating new code into the project under development. If a new code submission doesn't pass a quality gate, the code will have to be reworked until it reaches the defined standards.


The defined quality gates were the SonarQube default quality gates which we considered appropriate.

Conditions <span>?</span>		
Conditions on New Code		
Metric	Operator	Value
Issues	is greater than	0
Security Hotspots Reviewed	is less than	100%
Coverage	is less than	80.0% 
Duplicated Lines (%)	is greater than	3.0% 
Maintainability Rating	is worse than	A

Figure 2: SonarQube quality gates

However, our project did not achieve the final quality gate, failing, due to lack of time and some unsolved and not understandable errors.


Quality Gate Status ?



**Failed**  
2 failed conditions

8

New Issues



0.0% Security Hotspots Reviewed on New Code  
is less than 100%


Fix issues before they fail your Quality Gate with [SonarLint](#)  in your IDE. Power up with connected mode!

Figure 3: SonarQube Analysis Result

4 | TQS QA MANUAL

### 3 Continuous delivery pipeline (CI/CD)

#### 3.1 Development workflow

##### WorkFlow

We adopted GitHub flow, which maps our user stories as follows:

1. The main branch contains the stable version of the code in production, updated at the conclusion of each sprint;
2. The dev/development branch contains the latest code being prepared for the next release, we use "Pull Requests" to submit changes for code review, which must be approved by at a team member;
3. The feature branch is where each user story or new feature is developed in its own branch
4. The release branch is used to finalize the new version before production allowing adjustments and final bug fixes
5. The hotfix branch for critical bug fixes in production that are merged back into both the main and develop branches.

Due to the different types of collaboration, for example presential and live-share, these practices were not always respected as seen in the repository.

##### Code Review Practices

To ensure code quality, during the development process there were some code review practices used, like:

1. If the code isn't correctly written or explained the developer must rewrite it.
2. The developer must always assure that the style guidelines of the project are respected.
3. The comments in the code must be constructive.
4. All the developers must have in mind that functional improvements come first.
5. All the developers must make suggestions to keep the code clean and maintainable.
6. Each developer must do a follow up of the reviews made.

To ensure this code review process, we used pull requests to initialize code reviews of the submitted code.

## 3.2 CI/CD pipeline and tools

To keep an ambient of continuous integration and delivery, we have used GitHub Actions Tools.

```
name: SonarCloud
on:
  push:
    branches:
      - main
  pull_request:
    types: [opened, synchronize, reopened]
jobs:
  build:
    name: Build and analyze
    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' # 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: mvn -B verify -D sonar.organization=sonar-maven-plugin:sonar -D sonar.projectKey=SLIP_Projects-TGS
```

Figure 4: Sonar Cloud workflow

```
1 name: Java CI with Maven
2
3 on:
4   push:
5     branches: [ "main", "dev" ]
6
7 jobs:
8   build:
9
10    runs-on: ubuntu-latest
11
12    steps:
13      - uses: actions/checkout@v3
14
15      - name: Set up JDK 17
16        uses: actions/setup-java@v3
17        with:
18          java-version: '17'
19          distribution: 'temurin'
20
21      - name: Build and run unit tests with Maven
22        run: cd airportManager/airportManager && mvn clean test
23        continue-on-error: false
```

Figure 5: Maven Project Workflow

In the figures above, is the code responsible for the pipelines of our project, one executes and builds the maven project, the other runs the sonar cloud code quality control, both jobs are executed automatically every time there is a push to those branches.

There was also a try to implement a pipeline to do the deploy in a Docker Container, however, due to technical difficulties, this wasn't implemented.

## 4 Software testing

### 4.1 Overall strategy for testing

The test development strategy was TDD (Test-Driven Development), developing the tests first hand and then the system functionalities.

They were implemented unit tests, integration tests, using JUnit and Mockito, and selenium tests using the Selenium IDE.

The TDD is extremely important to understand the structure needed to develop all the features idealized.

### 4.2 Functional testing/acceptance

This type of testing is made from a user perspective, since these tests will validate what features are wanted. These tests usually use the cucumber and the type of testing that uses a given scenario and the steps to achieve a certain result having some parameters. We did not integrate these types of testing.

### 4.3 Unit tests

The unit tests were the first ones to be developed, testing our database entities. This testing process tests the smallest functional unit of code.

```
class FlightServiceImplTest {

    @Mock
    private FlightRepository flightRepository;

    @InjectMocks
    private FlightServiceImpl flightService;

    @BeforeEach
    void setUp() {
        MockitoAnnotations.openMocks(this);
    }

    @Test
    void testCreateFlight() {
        Flight flight = new Flight();
        flight.setFlightId(flightId:"F123");

        when(flightRepository.save(any(type:Flight.class))).thenReturn(flight);

        Flight createdFlight = flightService.createFlight(flight);

        assertNotNull(createdFlight);
        assertEquals(expected:"F123", createdFlight.getFlightId());
        verify(flightRepository, times(wantedNumberOfInvocations:1)).save(any(type:Flight.class));
    }
}
```

Figure 6: Example of an Unitary Test - Flight entity

### 4.4 System and integration testing

The integration tests were written to verify the connection and interactions between the different modules of the system.

For integration tests, the connection and access to the api was tested. The strategy chosen was MockMvc provided by SpringBoot to test every endpoint of the controller. These tests verify if the information returned equals the expected one.

```
@ExtendWith(MockitoExtension.class)
@AutoConfigureMockMvc
@SpringBootTest
class SeatControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @Mock
    private SeatService seatService;

    @InjectMocks
    private SeatController seatController;

    @BeforeEach
    void setUp() {
        mockMvc = MockMvcBuilders.standaloneSetup(seatController).build();
    }

    @Test
    void testGetSeats() throws Exception {
        List<Seat> seats = Arrays.asList(new Seat(seatId:"S123", seatNumber:"1A", flightId:"F123", flight:null));
        when(seatService.getAllSeats()).thenReturn(seats);

        mockMvc.perform(get(urlTemplate:"/api/seats/seats"))
            .andExpect(status().isOk())
            .andExpect(jsonPath(expression:"$[0].seatId").value(expectedValue:"S123"))
            .andExpect(jsonPath(expression:"$[0].seatNumber").value(expectedValue:"1A"))
            .andExpect(jsonPath(expression:"$[0].flightId").value(expectedValue:"F123"));
    }

    @Test
    void testGetSeatsByFlight() throws Exception {
        List<Seat> seats = Arrays.asList(new Seat(seatId:"S123", seatNumber:"1A", flightId:"F123", flight:null));
        when(seatService.getSeatByFlightId(anyString())).thenReturn(seats);

        mockMvc.perform(get(urlTemplate:"/api/seats/seatsByFlight")
            .param(name:"flightId", ..values:"F123"))
            .andExpect(status().isOk())
            .andExpect(jsonPath(expression:"$[0].seatId").value(expectedValue:"S123"))
            .andExpect(jsonPath(expression:"$[0].seatNumber").value(expectedValue:"1A"))
            .andExpect(jsonPath(expression:"$[0].flightId").value(expectedValue:"F123"));
    }
}
```

Figure 7: Example of an Integration Test - Seat Controller

## 4.5 Performance testing [Optional]

Performance testing is generally a testing practice to determine the performance of a system in terms of responsiveness and stability under a specific workload. Our main goal was to use the Selenium IDE, however, our user interface created a barrier with the frontend autocomplete on the flights search part that made us not to follow the selenium test to the end.

To ensure some quality in terms of performance, we used the jacoco dependency to study the coverage of our system in general.

airportManager												
airportManager												
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
ua.tqs.airportManager.controller		62%		66%	15	37	28	73	13	34	0	6
ua.tqs.airportManager.dto		83%		39%	56	118	0	25	3	65	0	6
ua.tqs.airportManager.service.impl		92%		n/a	5	42	4	66	5	42	0	8
ua.tqs.airportManager.entity		98%		100%	2	172	2	132	2	170	0	8
ua.tqs.airportManager		99%		100%	1	62	2	237	1	41	0	3
ua.tqs.airportManager.config		100%		n/a	0	18	0	61	0	18	0	5
ua.tqs.airportManager.jwt		100%		81%	3	22	0	50	0	14	0	2
Total	307 of 5 116	93%	69 of 174	60%	82	471	36	644	24	384	0	38

Figure 8: Coverage in Jacoco

Besides, and as mentioned before, we used SonarQube to code analysis.

The average coverage is 87.8 percent, however, we needed to implement more tests using different types of testing like cucumber and improve the selenium testing.

We accepted some code issues like the use of random for data initialization, folders' names and the "@Autowired" annotation.

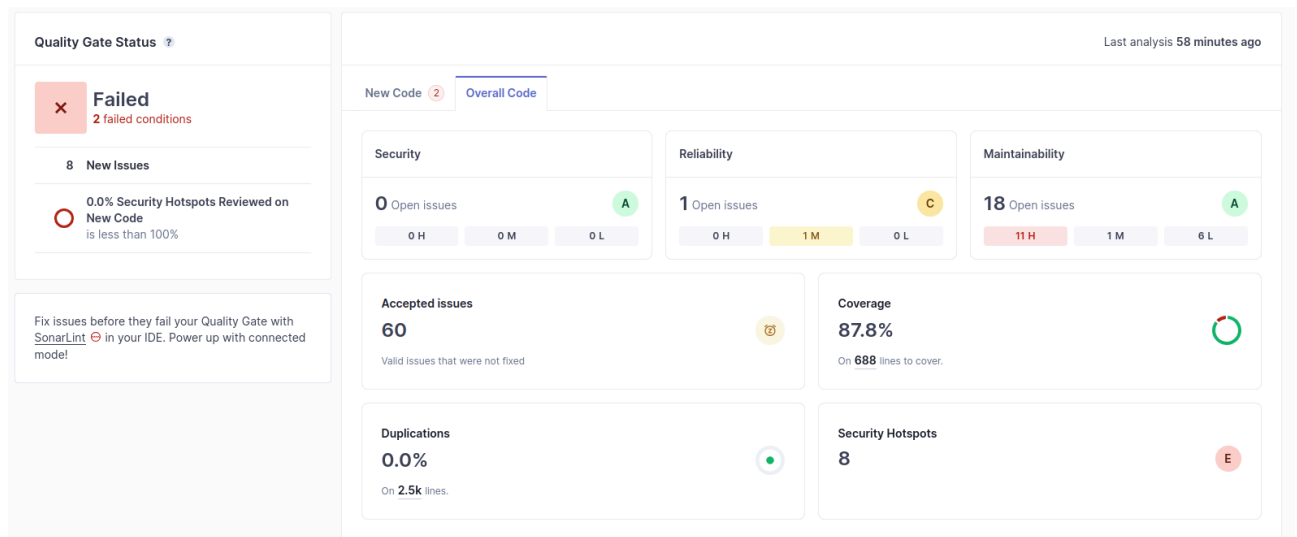


Figure 9: Coverage Page in SonarQube