TQS: Quality Assurance manual

André Butuc [103530], Artur Correia [102477], Daniel Carvalho [77036]
v2023-06-05

1 Project management

1.1 Team and roles

The following roles were assigned to each member, by the team, at the start of the project:

Table 1- Team roles used in the project, and their corresponding responsibilities.

Role	Responsibilities	Member
Team Coordinator	Ensure that there is a fair distribution of tasks and that members work according to the plan. Actively promote the best collaboration in the team and take the initiative to address problems that may arise. Ensure that the requested project outcomes are delivered in time.	Daniel
Product Owner	Represents the interests of the stakeholders. Has a deep understand of the product and the application domain; the team will turn to the Product Owner to clarify the questions about expected product features. Should be involved in accepting the solution increments.	Daniel
QA Engineer	Responsible, in articulation with other roles, to promote the quality assurance practices and put in practice instruments to measure que quality of the deployment. Monitors that team follows agreed QA practices.	Artur
DevOps Master	Responsible for the (development and production) infrastructure and required configurations. Ensures that the development framework works properly. Leads the preparing the deployment machine(s)/containers, git repository, cloud infrastructure, databases operations, etc.	André
Developer	Contributes to the development tasks which can be tracked by monitoring the pull requests/commits in the team repository.	All Members

1.2 Agile backlog management and work assignment

The backlog management practices, as well as the work assignment practices done by the team, are based on the Agile approach.

For this, a new sprint, lasting 7 days, is started on a weekly meeting scheduled for Thursdays at 2PM, during the class. In this iteration planning meeting, the team goes over the work that has to be completed and prioritized for the following week.

First, the Product Owner (PO), decides on the user stories present in the backlog to be prioritized in that sprint, after a discussion with all team members.

After the stories for the sprint are chosen, the PO leads a discussion on which the different sub-tasks required for the completion of that story are defined and outlined. These tasks will be the main unit of work for the sprint.

Once this activity is done, the team decides how to evaluate each user story, as well as each sub-task, on a story point estimate. This estimate, though subjective, should consider the amount of work the task will need, as well as its importance, and should thus be used to prioritize the work done by each member. Tasks are evaluated on a scale of 1-5 points, with the exception being given to really urgent or necessary tasks, that receive a 6 points evaluation. Once this is done, the work is distributed among the team members, taking into account their areas of expertise, with these members becoming the owners of the story/task.

The team uses JIRA to do this backlog management, with the board having four categories for the tasks in a sprint: “Not-started”, “In progress”, “Done” and “Waiting for QA”. Once a team member starts working on a task, they should move it to the “In Progress” category. The “Waiting for QA” category is reserved for user stories that already had their backend and frontend code, as well as the backend tests, implemented, but that are yet waiting for the implementation of the related functional tests on the frontend side.

Tasks should be moved to the “Done” category once their related code has been implemented. For this to happen, their related code (including the tests) must be sent to the dev branch of the respective repository using a pull request, having gone through the CI pipeline to guarantee that the quality gates are met.

2 Code quality management

2.1 Guidelines for contributors (coding style)

The coding style defined for this project follows the general Java language coding standards and conventions. Among some of the most important guidelines in these standards, that should be followed on all the subprojects of the DropMate implementation, are the following:

- Class names should be written in the CamelCase format. The same is true for the method and variable names. Meanwhile, constants should be named using uppercase letters, with words separated by underscore.
- When an exception is catched, the exception should not be ignored, but rather dealt with in a try/catch block, or thrown to a function above it in the flow of execution.
- When it comes to the REST API, exceptions related to the business logic of the service, such has a resource not being found, should be treated by personalized exceptions associated with a HTTP status to be returned as part of the API call.
- Classes and non-trivial public methods should always be accompanied by comments in the Javadoc style, describing the purpose of these classes/methods, and if necessary, the inputs and outputs obtained.
- Methods should be small and focused – if a method grows to big, with more than 40-50 lines, we should check whether it's possible to break it down into smaller methods while reducing the complexity of the understanding of the methods.
- Attributes and parameters should be declared, as much as possible, on the begininng of the code block or methods where they are used.
- The body of conditionals or loops should be written in between brackets, with the opening bracket preferably following the conditional expressions (on the same line). Closing brackets, on the other hand, should be placed on a new line.
- Logical sections of code should be separated with blank lines for clarity.
- Temporary code, or code that needs to be reworked, should be commented with TODO comments.

Though we're not very rigid in forcing a certain style, it's important that the coding style is consistent between the projects, and that there's no glaring differences between bordering code blocks.

2.2 Code quality metrics

To ensure code quality and facilitate continuous improvement, we utilized SonarCloud, a cloud-based code analysis service. SonarCloud was integrated into our CI pipeline, allowing for automated code analysis and issue detection.

By default, SonarCloud provides a standard Quality Gate. However, we customized this gate to create two additional personalized quality gates tailored to our specific requirements. For the backend repositories, namely DropMate and Floral-Fiesta, we defined the following criteria:

- Coverage: The code should have a test coverage of over 70%.
- Duplicated Lines: The presence of duplicated lines should be less than 10%.
- Maintainability Rating: The code should have a Maintainability Rating of level A or higher.
- Reliability Rating: The code should have a Reliability Rating of level A or higher.
- Security Hotspots: The code should have zero security hotspots.
- Security Rating: The code should have a Security Rating of level A or higher.

As for the frontend repositories, Floral-Fiesta-UI and DropMate-UI, we applied similar criteria but disabled the coverage percentage. This decision was made since functional tests were not directly performed on these repositories, but rather on their corresponding backend repositories.

Implementing these quality gates allowed us to gain valuable insights during development. SonarCloud effectively detected issues such as Code Smells, Bugs, and other code-related problems. This provided developers with an obligation to address these issues, ultimately resulting in improved code quality. Additionally, the review process for accepting new functionalities was streamlined as reviewers could rely on the feedback provided by SonarCloud.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

The development workflow used for this project is an adapted version of the Git Feature Branch workflow.

First of all, it's worth pointing out that different sub projects will be kept on different repositories, as described below:

- The **DropMate** repository will store the development of the backend of the DropMate platform
- The **DropMate UI** repository will store the UI for both the admin view of the DropMate platform frontend, as well as the UI for the ACP operators
- The **Floral-Fiesta** repository will store the UI for the Floral Fiesta e-store, developed as an example of a e-store partnered up with DropMate
- The **MainHub** is the central repository for documentation.

As such, for each of the 3 development repositories (DropMate, DropMateUI and Floral-Fiesta), there will be two branches to record the history of the project, according to the gitflow workflow:

- The **main** branch, which will store the official release history of the project
- The **dev** branch, which will serve as an integration branch for different features

Besides these two, each new feature should reside in its own **feature branch,** using dev as a parent branch.

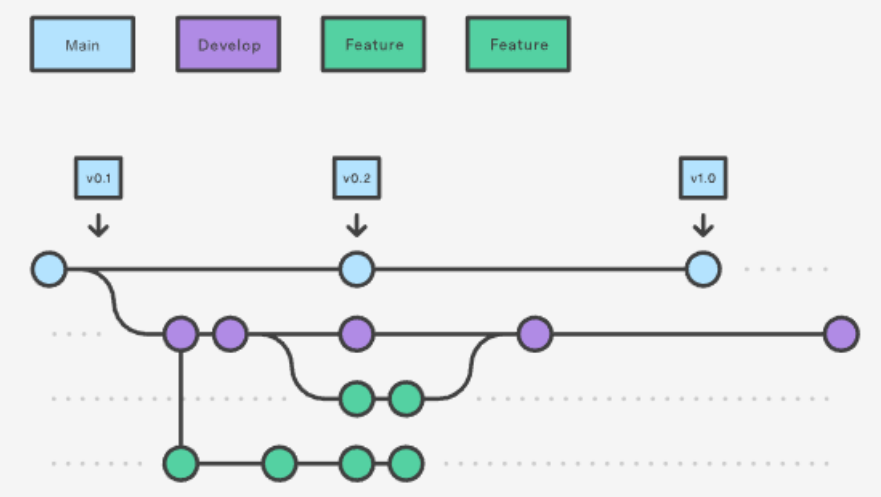

Figure 1 – A diagram representing the workflow of the Git feature branch workflow, used as the basis for the development workflow implemented in this project.

In this context, the features will be taken from each of the development user stories present on the JIRA board. Some of the user stories require work on different repositories (for example, the user story for the browsing of ACP's in the DropMate admin UI by a system administrator requires work on the UI frontend on the DropMate UI repository, as well as the corresponding API calls and business logic, as well as testing, on the DropMate repository). In these cases, each sub-task will have it's own branch on the corresponding repository.

The naming of these branches should follow a convention so that they are easily track to the corresponding user story on the JIRA platform. For example, for the "Admin Removes an ACP from the platform", the branch should be named "acp-removal-from-platform".

Feature branches should be merged into the dev branch, once they are tested, finished, and passing the defined quality gates.

Once a feature is ready to be merged into the dev branch, a Pull Request should be created by the developer responsible for this feature. In this Pull Request, the developer should detail the changes made and the main implementations done on the corresponding code. If the feature is minimal, the dev can then merge the pull request themselves. However, if the feature is more important, it should be reviewed by someone else from the developers team.

As a consequence of the nature of the user stories development previously described, a user story will only be considered done, and should only be marked as such in the JIRA tracker, when all it's composing sub-tasks and features are completed (relating to the frontend, backend and/or testing of the user story).

3.2 CI/CD pipeline and tools

In both of our backend development repositories we used GitHub actions to define two different workflows:

a) Workflow that activated when push actions were made into our "dev" branch:

```yaml
name: SonarCloud
on:
  push:
    branches:
      - dev
  pull_request:
    types: [opened, synchronize, reopened]

jobs:
  build:
    name: Build and analyze
    runs-on: ubuntu-latest

    steps:
      - name: Checkout repository
        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'

      - 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('backend/**/pom.xml') }}
          restore-keys: ${{ runner.os }}-m2

      - name: Build and analyze
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
          SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
        run: |
          cd backend
          ./mvnw -B verify org.sonarsource.scanner.maven:sonar-maven-plugin:sonar -Dsonar.projectKey=DropMate-Corp_Floral-Fiesta
```

Figure 2 – Workflow for the dev branch

b) Workflow that activated when push actions were made into our "main" branch:

```yaml
name: Build and deploy JAR app to Azure Web App - floralfiestaAPI

on:
  push:
    branches:
      - main
  workflow_dispatch:

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout repository
        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'

      - 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('backend/**/pom.xml') }}
          restore-keys: ${{ runner.os }}-m2

      - name: Build and analyze
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
          SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
        run: |
          cd backend
          ./mvnw -B verify org.sonarsource.scanner.maven:sonar-maven-plugin:sonar -Dsonar.projectKey=DropMate-Corp_Floral-Fiesta

      - name: Upload artifact for deployment job
        uses: actions/upload-artifact@v2
        with:
          name: java-app
          path: '${{ github.workspace }}/backend/target/*.jar'

  deploy:
    runs-on: ubuntu-latest
    needs: build
    environment:
      name: 'Production'
      url: ${{ steps.deploy-to-webapp.outputs.webapp-url }}

    steps:
      - name: Download artifact from build job
        uses: actions/download-artifact@v2
        with:
          name: java-app

      - name: Deploy to Azure Web App
        id: deploy-to-webapp
        uses: azure/webapps-deploy@v2
        with:
          app-name: 'floralfiestaAPI'
          slot-name: 'Production'
          publish-profile: ${{ secrets.AZUREAPPSERVICE_PUBLISHPROFILE_61B9D1DE76BD4895B76402E6582599AB }}
          package: '*.jar'
```

Figure 3 – Workflow for the main branch.

Each repository was mapped and linked to a Sonar Cloud project to enable static-code analysis.

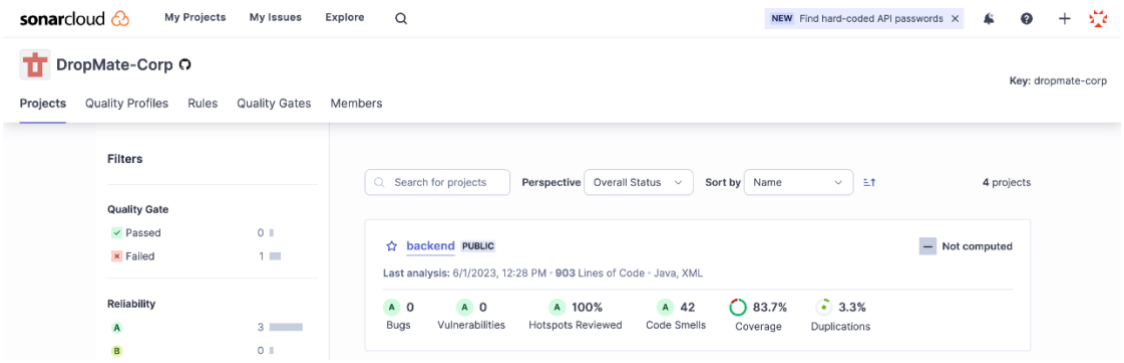

Figure 4 – Sonar Cloud page for the organization.

The a) workflow code, besides running the tests and checking each one of them passed, generated a Sonar Cloud static code analysis report on pull requests, which allowed us to validate the pull request and accept its merge to the “dev” branch:

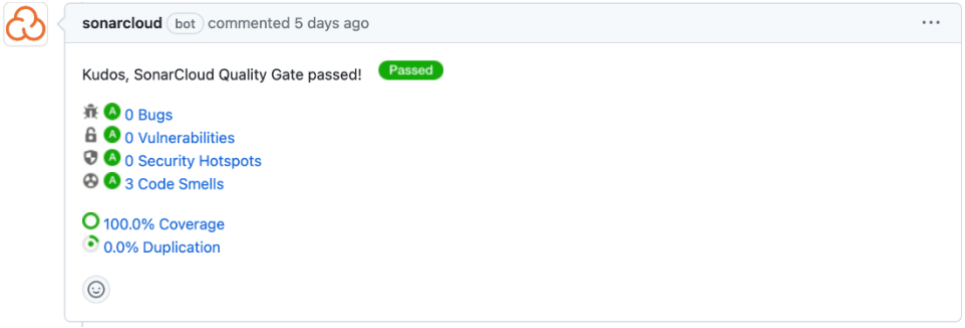

Figure 5 – Example of a quality gate analysis.

The b) workflow code, besides doing everything that the a) workflow did, it also ran a job to deploy the code into the linked “Azure App Service”:

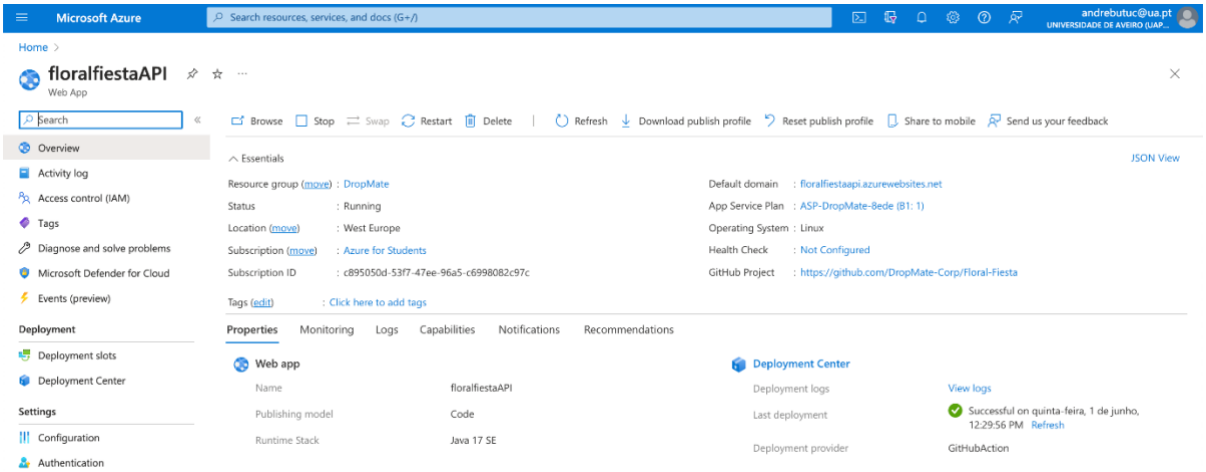

Figure 6 – Azure page for the Floral Fiesta service.

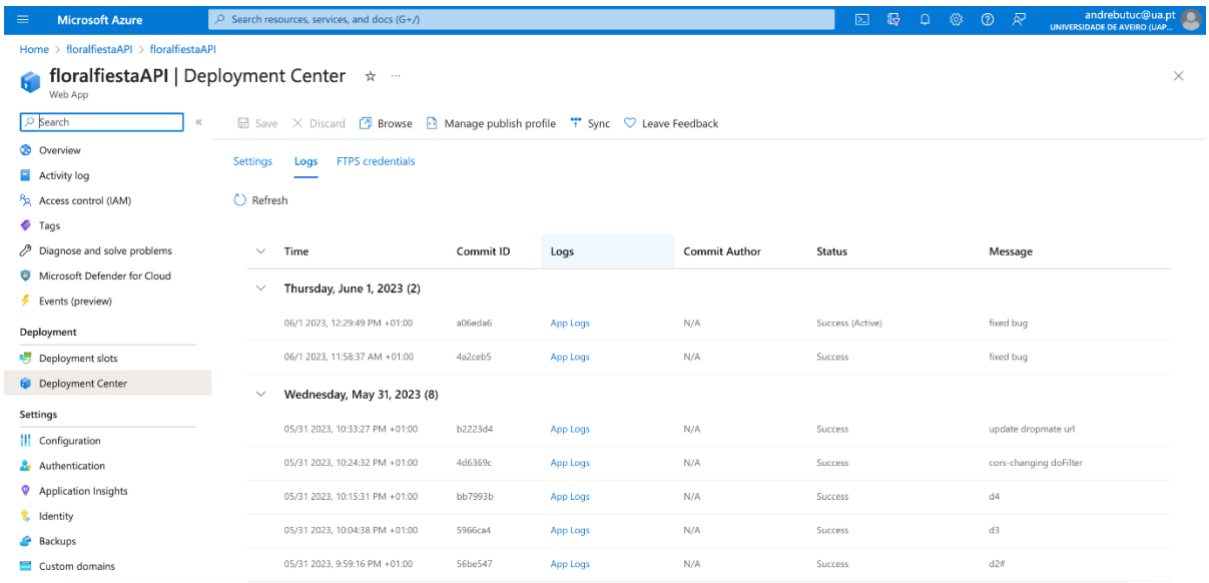

Figure 7 – Floral Fiesta's Deployment Center.

For both of our frontend repositories we used a GitHub Pages workflow to deploy the user interface:

```yaml
name: Deploy

on:
  push:
    branches:
      - main

jobs:
  build:
    name: Build
    runs-on: ubuntu-latest

    steps:
      - name: Checkout repo
        uses: actions/checkout@v2

      - name: Setup Node
        uses: actions/setup-node@v1
        with:
          node-version: 16

      - name: Install dependencies
        uses: bahmutov/npm-install@v1

      - name: Build project
        run: npm run build

      - name: Upload production-ready build files
        uses: actions/upload-artifact@v2
        with:
          name: production-files
          path: ./dist
```

```
33
34      deploy:
35        name: Deploy
36        needs: build
37        runs-on: ubuntu-latest
38        if: github.ref == 'refs/heads/main'
39
40        steps:
41          - name: Download artifact
42            uses: actions/download-artifact@v2
43            with:
44              name: production-files
45              path: ./dist
46
47          - name: Deploy to GitHub Pages
48            uses: peaceiris/actions-gh-pages@v3
49            with:
50              github_token: ${{ secrets.GITHUB_TOKEN }}
51              publish_dir: ./dist
```

Figure 8 – Workflow for user interface deployment.

3.3 System observability

By using Microsoft Azure's microservices, a broad variety of tools were made accessible to our project, for example, a metrics dashboard. In this dashboard it was possible to select a metric and observe its behaviour throughout the day or even the last month.

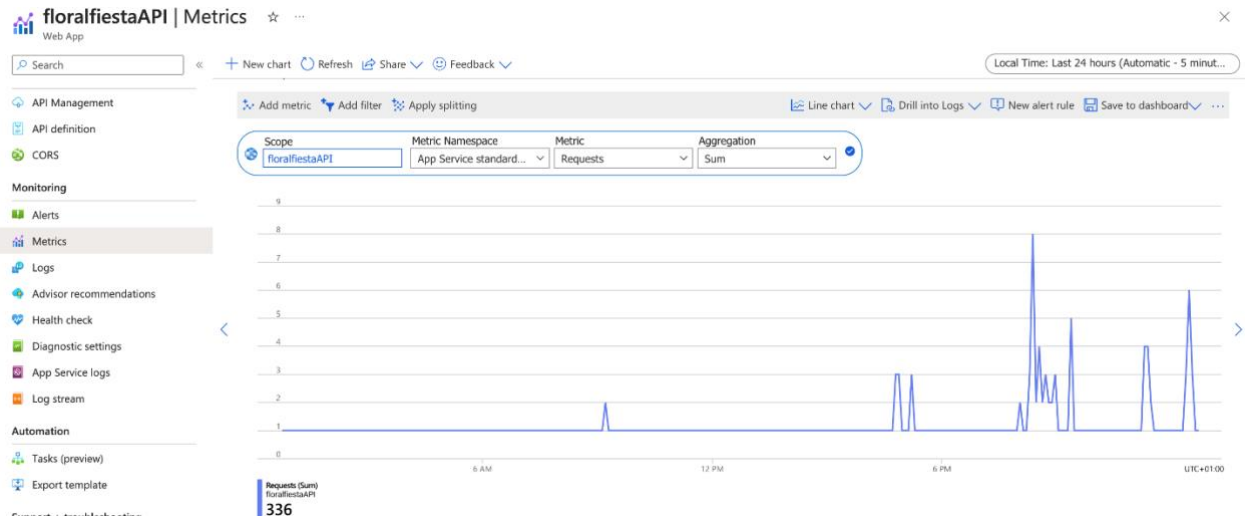

Figure 9 – Dashboard for observability in Azure.

Additionally, and most importantly, it was possible to define "alerts" that would notify us via email of occurrences or deviations to the system's performance and behaviour:

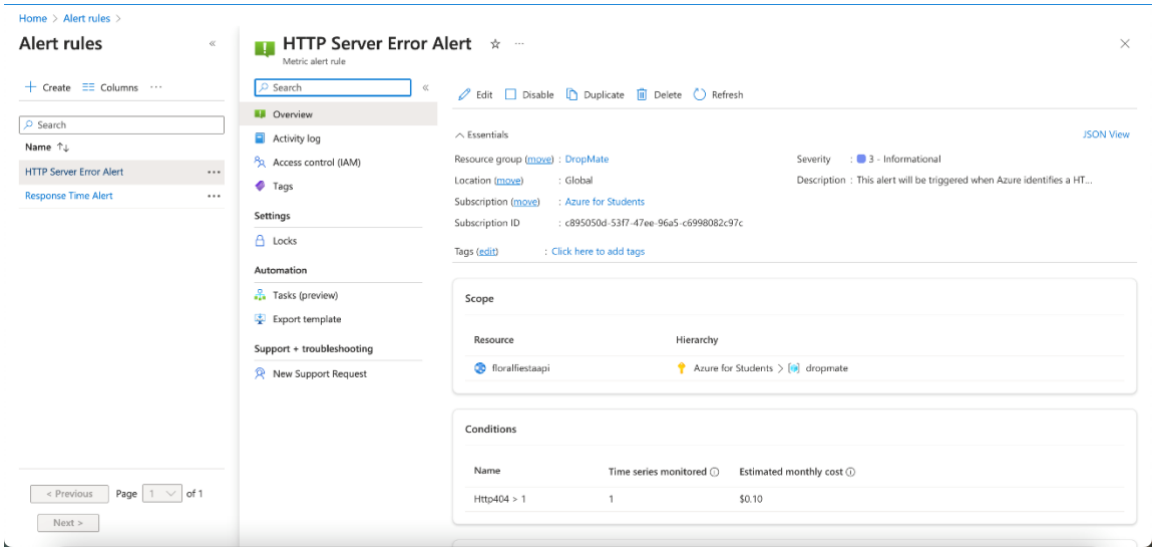

Figure 10 – Example of an alert.

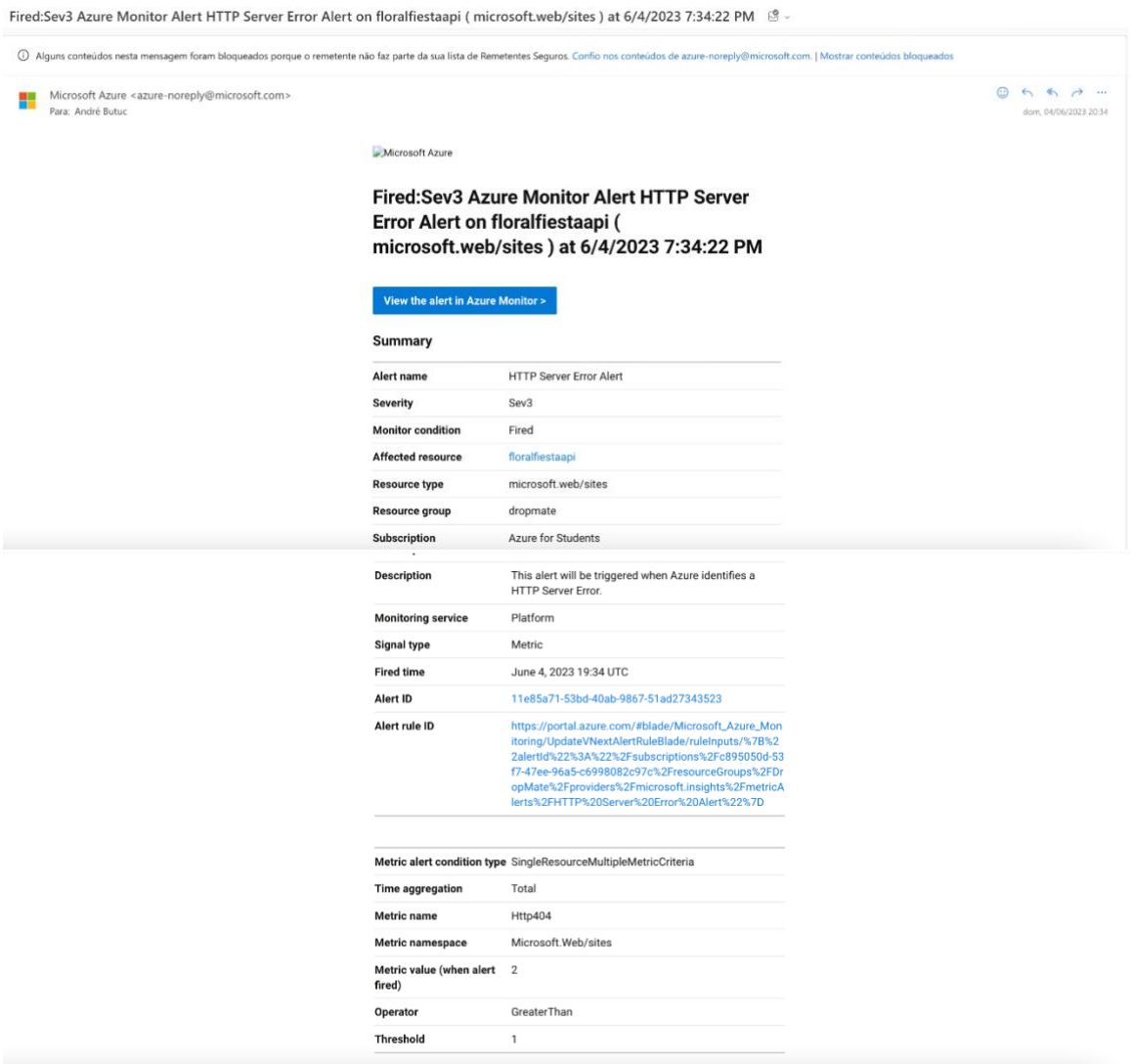

Figure 11 - Email related to an alert.

The alerts that we defined served mainly as proof-of-concept since we were not able to manipulate meaningful metrics of the system, such as, “time response”, to test the alerts.

4 Software testing

4.1 Overall strategy for testing

The overall strategy for testing applied in this project is split into two approaches, one for the development of the backend of both services, and another for the development of the frontend of both services.

In the backend, a test-driven development (TDD) approach was applied. For this, we initially started the implementation of these modules by creating the signatures for the controller and services methods of the backend. Before writing any code in these methods, we then developed the associated unit tests, boundary controller tests, and integration tests in order to test the expected behavior of such methods, and to ensure that the developed code meets the established requirements for the business logic. Afterwards, we write the minimung corresponding code to make the tests pass. When this happens, we refactor the code to improve its design and readability, we run the tests for a final time, and we consider that the implementation of these methods is complete, sending the commits to the CI/CD pipeline. Different testing tools are mixed in the development of these tests:

- For the unit tests, Mockito is used to mock the behavior of associated classes.
- For the boundary tests, we use the @WebMvcTest mode, with the use of @MockBean to simulate the access to services, and MockMvc as the REST API entry point.
- For the integration tests, with the loading of the full Spring Boot application context, we use the @SpringBootTest mode, with RestAssured being used as the REST API entry point, and the Testcontainers library, with a MySQL container, being used for the creation of a database container for testing.

Meanwhile, for the frontend of all platforms, the initial goal was to implement a behavior-driven development (BDD) approach, using both Cucumber and Selenium for the writing of the tests, implementing functional tests based on the acceptance criteria defined for each user story, and using the page object model. However, due to time constraints this wasn't possible, and instead, the tests were written fully using Selenium, while still keeping the page object model, and replicating the requirements established by the acceptance criteria. Thus, these tests were developed using Selenium Jupiter, running an instance of the Firefox Web Driver.

The results of the developed tests are being considered in the quality gates defined for the static code analysis inside the CI/CD workflow, using Sonar Cloud, as described in the previous section.

4.2 Functional testing/acceptance

The project follows a developer-oriented, open box approach to writing functional tests. These tests are designed based on the main user stories defined for the application. Once the frontend and backend development are completed, functional tests are written to verify the correct implementation of the user flow.

The functional tests are not directly implemented in the UI repositories, instead, they are integrated into the respective backend code. Consequently, these tests are not considered in the Continuous Integration (CI) process of the frontend application. They serve as a development-focused perspective to ensure the proper functioning of the system.

To facilitate the development process and accommodate future additions, the tests are implemented using the Page Object Pattern. This pattern simplifies the creation and maintenance of the tests, making it easier to incorporate new tests as needed. The tests utilize Selenium dependencies and leverage the WebDriverManager implementation by Boni García.

By adopting this approach, a user story is deemed complete only when its corresponding functional test passes, confirming that the desired functionality has been successfully implemented.

4.3 Unit tests

As previously stated, the unit tests for this project are written using a TDD approach. This considers a open box perspective, as the tests are written by the developers of the code itself, thus having access to the internal implementation details of the code being tests, including its private methods, variables and internal states.

In the backend of both of the subsystems of the project, that is, the Floral Fiesta backend and the DropMate backend, the unit tests are developed to validate the business logic and behaviour of the corresponding services, using the Mockito library to avoid the unnecessary loading of the entire Spring Boot application context, and thus, isolating the code being tested and focus on its specific behaviour without involving actual implementations of dependencies. Instead, mocks are created to simulate the behaviour of the underlying repositories with which the services need to interact in order to fulfil their logic. In the case of the Floral Fiesta backend, unit tests also require the mocking od the DropMateAPIClient, used for the creation of the HTTP client that integrates with the DropMate backend.

In both services, in the case in which objects were required to be instantiated and used in multiple tests (such as arrays of Parcels in different states, creation of ACP's, etc), these were created in a method for test setup annotated with the @BeforeEach annotation.

The model for the creation of each @Test method was based on the setting up of the required expectations using the Mockito when method, followed by assertions using the AssertJ library, and the use of Mockito's verify to ensure that the underlying expected repository mocked methods are called.

When testing a method with different outcomes and scenarios resulting from its behaviour logic, different tests should be written representing the expected outcomes for each scenario. For instance, when testing the service method for returning the parcels waiting for delivery at a specific ACP, we should test not only the positive scenario with the expected outcome, but also the scenario in which an invalid ACP ID is passed into the method, with a corresponding assertion being thrown. An example of a test, for the positive expected outcome of the example just mentioned, is shown next.

```
@Test
void whenGetAllParcelWaitDelivery_atSpecificACP_withValidACP_thenReturnOnlyCorrectACP() throws ResourceNotFoundException {
    // Set up Expectations
    when(acpRepository.findById(Mockito.any())).thenReturn(Optional.ofNullable(pickupPointOne));

    when(parcelRepository.findAll()).thenReturn(allParcels);

    // Verify the result is as expected
    List<Parcel> returnedParcels = adminService.getParcelsWaitingDeliveryAtACP( acpID: 1);
    assertThat(returnedParcels).hasSize( expected: 1);
    assertThat(returnedParcels).extracting(Parcel::getParcelStatus).containsOnly(Status.IN_DELIVERY);

    // Mockito verifications
    this.verifyFindByIdIsCalled();
    Mockito.verify(parcelRepository, VerificationModeFactory.times( wantedNumberOfInvocations: 1)).findAll();
}
```

Figure 12 – Example of a unit test.

4.4 System and integration testing

The system and integration testing of the backend methods, likewise, also followed an open box developer perspective, using a TDD approach.

Two different kinds of tests were implemented:

- Boundary tests without loading the entire Spring Boot application context, using @WebMvcTest. The goal of these was the testing of the behavior of the REST controller endpoints themselves, to ensure that the controllers are mapped as expected, that the HTTP

requests are handled correctly, as well as the different type of possible request parameters, and that the appropriate HTTP responses are generated.

- Integration tests using @SpringBootTest, in which the entire contest of the application is loaded, thus testing the entire context of the project, to ensure that the interaction between the various components and layers of the backend works as expected, thus returning appropriate responses to the requests of the contacting REST clients.

The policy followed in this project is for the boundary tests to be written first, and for these to be passing, before the integration tests are written and tested.

As previously stated, the underlying services required for these tests are mocked using @MockBean, and MockMvc is used to mock the REST client. A similar approach to the one described for the unit tests should be followed when it comes to the creation of instances of the objects necessary for testing, using the @BeforeEach annotation. Besides, when one same REST controller endpoint has various possible responses being returned, depending on the input verification and the logic and behavior of the underlying service methods, all outcomes should be tested and verified. The following figure shows an example for the testing of an endpoint which returns the parcels waiting delivery for a specific ACP.

```
1 Daniel Carvalho
@Test
void whenGetParcelsWaitingDelivery_atSpecificACP_withInvalidID_thenReturn_statusNotFound() throws Exception {
    when(acpService.getAllParcelsWaitingDelivery( acpID: -2)).thenThrow(new ResourceNotFoundException("Couldn't find ACP with the ID -2!"));

    mockMvc.perform(
                    get( urlTemplate: "/dropmate/acp_api/parcel/all/delivery").contentType(MediaType.APPLICATION_JSON)
                            .param( name: "acpID",  ...values: "-2"))
            .andExpect(status().isNotFound());
}
```

Figure 13 – Example of a endpoint boundary test.

The integration tests, meanwhile, were developed using RestAssured as the REST client. The testcontainers library was used to validate the interaction with a test database, created on a MySql container.

When entities are required to be saved into the database prior to a test, the saveAndFlush() method should be used to interact with the test database. Junit's @TestMethodOrder is used to define the order in which the different tests should be carried out, helping in the managing of the instances of objects saved and flushed from the test database. As was the case for the unit tests and boundary tests, all the expected outcomes of a REST endpoint should be tested.

In the case of the Floral Fiesta backend, the integration tests also require the testing of the integration with the DropMate API, required to carry out behaviors such as the placing of orders by the Floral Fiesta users.

The following figure shows an example of the implementaiton of an integration test for the endpoint which returns the parcels waiting delivery for a specific ACP.

```
@Test
@Order(9)
void whenGetParcelsWaitingPickup_atSpecificACP_withValidID_thenReturn_StatusOK() throws Exception {
    parcelRepository.saveAndFlush(new Parcel( deliveryCode: "DEL123",  pickupCode: "PCK123",  weight: 1.5,  deliveryDate: null,  pickupDate: null, Status.IN_DELIVERY, testACP, testStore));
    parcelRepository.saveAndFlush(new Parcel( deliveryCode: "DEL456",  pickupCode: "PCK456",  weight: 3.2,  deliveryDate: null,  pickupDate: null, Status.IN_DELIVERY, testACP2, testStore));
    parcelRepository.saveAndFlush(new Parcel( deliveryCode: "DEL790",  pickupCode: "PCK356",  weight: 1.5, new Date( year: 2023,  month: 5,  day: 22),  pickupDate: null, Status.WAITING_FOR_PI
    parcelRepository.saveAndFlush(new Parcel( deliveryCode: "DEL367",  pickupCode: "PCK803",  weight: 2.2, new Date( year: 2023,  month: 5,  day: 22),  pickupDate: null, Status.WAITING_FOR_PI
    parcelRepository.saveAndFlush(new Parcel( deliveryCode: "DEL000",  pickupCode: "PCK257",  weight: 1.5, new Date( year: 2023,  month: 5,  day: 22), new Date( year: 2023,  month: 5,  day: 28)
    parcelRepository.saveAndFlush(new Parcel( deliveryCode: "DEL843",  pickupCode: "PCK497",  weight: 1.6, new Date( year: 2023,  month: 5,  day: 22), new Date( year: 2023,  month: 5,  day: 29)

    RestAssured.with().contentType( s: "application/json") RequestSpecification
            .when().get( s: BASE_URI + randomServerPort + "/dropmate/admin/parcels/17/pickup") Response
            .then().statusCode( i: 200) ValidatableResponse
            .body( s: "size()", is( value: 1)).and()
            .body( s: "parcelStatus", hasItems(Status.WAITING_FOR_PICKUP.toString())).and()
            .body( s: "pickupCode", hasItems("PCK356"));
}
```

Figure 14 – Example of an integration test.