TQS: Quality Assurance manual

Gonçalo Ferreira [107853]*, João Carlos Santos *[110555], Matilde Teixeira [108193], Sara Almeida [108796]

V2024/06/03

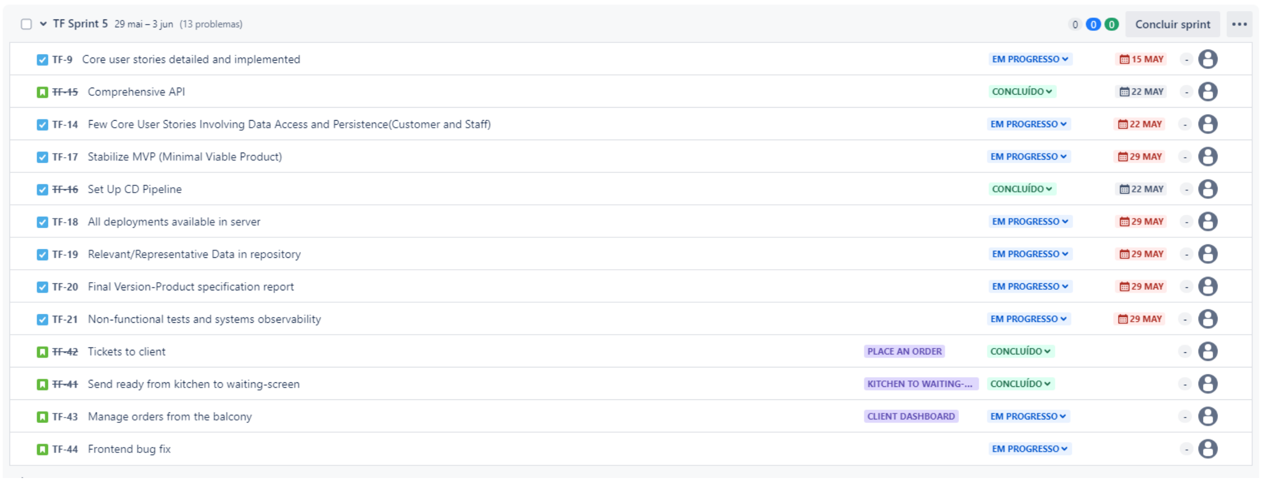
1 Project management

1.1 Team and roles

Gonçalo Ferreira	Matilde Teixeira	Sara Almeida	João Carlos
Team Manager	QA Engineer	Product Owner	DevOps Master

1.2 Agile backlog management and work assignment

We used Jira for backlog management. In Jira we used a board to keep track of our issues and user stories, we also used a backlog to manage our sprints and organize the tasks within the sprints. Whenever a new week of work starts, we start the sprint on Jira. We also linked our GitHub to Jira for every time we create a new issue, we can create a new branch on GitHub.



TF Sprint 5 29 mai – 3 jun (13 problemas) 0 0 0 Concluir sprint

TF-9	Core user stories detailed and implemented		EM PROGRESSO	15 MAY	-
~~TF-15~~	Comprehensive API		CONCLUÍDO	22 MAY	-
TF-14	Few Core User Stories Involving Data Access and Persistence(Customer and Staff)		EM PROGRESSO	22 MAY	-
TF-17	Stabilize MVP (Minimal Viable Product)		EM PROGRESSO	29 MAY	-
~~TF-16~~	Set Up CD Pipeline		CONCLUÍDO	22 MAY	-
TF-18	All deployments available in server		EM PROGRESSO	29 MAY	-
TF-19	Relevant/Representative Data in repository		EM PROGRESSO	29 MAY	-
TF-20	Final Version-Product specification report		EM PROGRESSO	29 MAY	-
TF-21	Non-functional tests and systems observability		EM PROGRESSO	29 MAY	-
~~TF-42~~	Tickets to client	PLACE AN ORDER	CONCLUÍDO		-
~~TF-41~~	Send ready from kitchen to waiting-screen	KITCHEN TO WAITING-...	CONCLUÍDO		-
TF-43	Manage orders from the balcony	CLIENT DASHBOARD	EM PROGRESSO		-
TF-44	Frontend bug fix		EM PROGRESSO		-

Fig1. - Jira Backlog

In the beginning of the project, we set the work to be done in every sprint. Each week we start the sprint, and we choose to not attribute the task to a single person to work on, sometimes we work in pairs, other times we work alone. At the end of the week, we close the sprint and if some tasks were left to be done, they are added to the next sprint. We defined Epics which have associated different User stories.

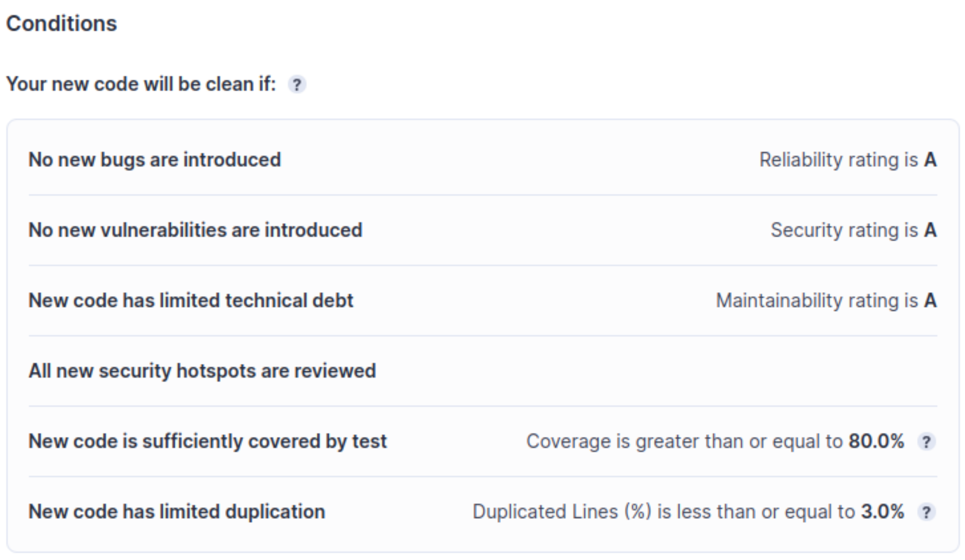
2 Code quality management

2.1 Guidelines for contributors (coding style)

Class names should be nouns that describe the entity. We also define that names and method names should be verbs or verb phrases.
Inline comments for sparingly and only for complex logic that requires explanation.
We organize classes into appropriate folders based on their functionality.

2.2 Code quality metrics and dashboards

SonarQube was the tool we chose to use on this project. This decision was due to the tool’s offer: a set of relevant evaluations of the developed code, reporting bugs, code smells, etc. There was some consideration given to modifying the default quality gates provided by the platform. However, the team did not feel the need to change them as they were already high. It is important to note that it was used a SuppressWarning for a method that has deprecated, although it is essential to test the controller.


Conditions

Your new code will be clean if:

No new bugs are introduced	Reliability rating is **A**
No new vulnerabilities are introduced	Security rating is **A**
New code has limited technical debt	Maintainability rating is **A**
All new security hotspots are reviewed	
New code is sufficiently covered by test	Coverage is greater than or equal to **80.0%**
New code has limited duplication	Duplicated Lines (%) is less than or equal to **3.0%**

Fig2. - Default Quality Gates SonarQube

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

We used **Jira** workflow for tracking the work done. On creating an issue in Jira we define the state with “open”, whenever someone starts working on it, we change to “in progress” and whenever someone resolve the issue, we change it for “done”.

For code reviewing our colleague's work, we use github pull request, so if someone wants to commit something to the development branch, another member of the group must review the work and accept the pull request.

A user story is done when the front end, backend logic and tests are done. If we can role play as the user and go through the flow of the scenario without any problems, we can consider that the user story is done.

3.2 CI/CD pipeline and tools

For the version control of the system, we use git hub. All code is stored in a Git repository, and we create branches for each issue/bug fix/other tasks. Then, whenever a branch is ready to close, we create a pull request to the development branch that must be accepted by some other student.

To ensure the quality of the code developed we use the sonar integrated in the github that checks the coverage and the test gate that we defined.

4 Software testing

4.1 Overall strategy for testing

The team initially aimed to follow a Test-Driven Development (TDD) approach, but due to development obstacles, we utilized it only for the first repositories and services and then we opted to develop the software first and later focus on testing. Extensive tests were written for controllers, services, and repositories using tools like SpringBootTest, TestRestTemplate, JUnit 5, and AssertJ.

In the later phase, we adopted a REST-assured strategy for thorough API testing, followed by implementing Behavior-Driven Development (BDD) with Cucumber for clear, readable tests, and end-to-end (E2E) testing with Selenium to simulate user interactions.

Our strategy ensures comprehensive test coverage, continuous integration (CI) integration, test independence, detailed documentation, to maintain high-quality standards and a reliable and production software delivery.

4.2 Functional testing/acceptance

Functional testing ensures that the application works as expected from an end-user perspective. This type of testing treats the system as a “black box”, focusing on validating the outputs based on given inputs without considering the internal workings of the application.

Our test coverage and scope focus on the typical usage scenarios, edge cases, and error conditions. Each feature is tested to ensure it meets the acceptance criteria defined during the requirement phase. For the test cases definition, we define the tests based on user stories. Each test should describe the test purpose, preconditions, test steps, expected results, and postconditions.

For the associated Resources we used Selenium for the front-end testing, postamn for testing API endpoints, JUnit for integration and unit tests.

4.3 Unit tests

Unit tests are meant to test an isolated part of the system, to guarantee that they work properly. With this aim we developed tests to ensure that even the most basic parts of the system behave as expected.

To study the behavior of Services, we follow a consistent structure to ensure isolated testing of each service method. We use Mockito to create mock objects for the service’s dependencies that are being tested. These mocks are injected into the service using the @InjectMocks annotation.

Each test case is designed to validate a specific functionality of the service, that includes coverage of various scenarios including normal operation, edge cases, and error handling.

We use various assertion methods from JUnit and AssertJ to validate the results, as well as verify interactions with the mock objects using Mockito's verification methods.

Relating to the Repositories, we implemented tests using the annotation @DataJpaTest, to ensure Data Access. With this we configure an in-memory database and provide repository testing support.

The tests were developed to verify all methods declared in the interfaces of each repository, as well as some other frequently used methods. In these tests, we made sure to cover edge cases, error scenarios, and common usage cases.

We use @Autowired to inject the TestEntityManager and the tested repository. This allows direct interaction with the persistence context. We use AssertJ for the assertion syntax in the tests. This methodology allows validation of the persistence layer in an efficient way, by ensuring reliable data access and manipulation within the application.

For the Controllers, we implemented tests using the @WebMvc annotation, which focuses on testing the web layer by configuring Spring MVC components. This allows us to test the controller endpoints without starting the full application context. By using this we also used a MockMvc instance, which is autowired to simulate HTTP requests and verify responses. Mock beans are injected to isolate the controller from the service layer and repositories during testing. These tests cover edge cases, error scenarios, and common usage cases.

Each test method performs an HTTP request using MockMvc, verifying the response status code and content. We also check the interaction between the controller and the mocked service layer by asserting the number of method invocations.

For OrderController_WithMockService tests, we focus on the WebSocket functionality, which handles orders in real-time through WebSocket connections. We use similar annotations, such as @WebMvcTest and @MockBean, but in the context of WebSocket testing, we include WebSocketConfig.class in the @ContextConfiguration annotation to provide the necessary WebSocket configuration.

These tests utilize WebSocketStompClient to establish connections, subscribe to order-related topics, and handle messages. We also test for error scenarios, such as failed WebSocket connections, to ensure the resilience and reliability of the WebSocket-based features, validating the system's ability to process orders seamlessly even in challenging network conditions.

```java
    @Test
    void whenWebSocketConnectionFails_thenHandleError() {
        WebSocketClient client = new StandardWebSocketClient();
        WebSocketStompClient stompClient = new WebSocketStompClient(client);
        stompClient.setMessageConverter(new MappingJackson2MessageConverter());

        WebSocketHttpHeaders handshakeHeaders = new WebSocketHttpHeaders();

        assertThrows(Exception.class, () -> {
            CountDownLatch latch = new CountDownLatch(1);

            StompSessionHandlerAdapter sessionHandler = new StompSessionHandlerAdapter() {
                @Override
                public void afterConnected(StompSession session, StompHeaders connectedHeaders) {
                    session.subscribe("/topic/orders", new StompSessionHandlerAdapter() {
                        @Override
                        public Type getPayloadType(StompHeaders headers) {
                            return Order.class;
                        }

                        @Override
                        public void handleFrame(StompHeaders headers, Object payload) {
                            latch.countDown();
                        }
                    });

                    session.send("/app/wsorder", order);
                }
            };

            stompClient.connect("ws://invalidhost:8080/ws", handshakeHeaders, sessionHandler).get(5,
TimeUnit.SECONDS);
        });
    }
```

Fig3.- Test for WebSocket Failed Connection

```java
@Test
void testOrder() {
    Order mockOrder = mock(Order.class);

    Order result = ((OrderController) new OrderController()).order(mockOrder);

    assertEquals(mockOrder.getOrderId(), result.getOrderId());

}
```

Fig4.- Test for Order Receival

4.4 System and integration testing

The chosen approach was an open-box approach, that allows developers to test the application's internals through its public interfaces.

This policy ensures tests are independent, reliable, and maintain database integrity through transactional tests that roll back changes after execution.

The goal is to ensure comprehensive coverage of critical paths, validating application's behavior under various conditions. Each test includes a setup phase to clean and initialize the database with test data, an execution phase to perform operations through API endpoints, and a verification phase to assert outcomes.

API testing is crucial to ensure endpoint functioning. This involves validating CRUD operations for the entities, ensuring error handling for invalid requests, covering edge cases and conditions.

Using TestRestTemplate to interact with the API and JUnit 5 and AssertJ for assertions, our API testing ensures that endpoints perform as expected, providing accurate responses. This thorough testing approach helps maintain the API's robustness and reliability.

4.5 Performance testing [Optional]

We did not perform performance testing.