deti universidade de aveiro departamento de eletrónica, telecomunicações e informática

TQS: Quality Assurance manual

João Andrade 107969, Pedro Pinho 109986, Tomás Victal 109018
v2024-06-04

1 Project management

1.1 Team and roles

Team Leader (Tomás)	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.
Product owner (Tomás)	Represents the interests of the stakeholders. Has a deep understanding 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.

QA Engineer (João)	Responsible, in articulation with other roles, to promote the quality assurance practices and put in practice instruments to measure quality of the deployment. Monitors that team follows agreed QA practices.
DevOps master (Pedro)	Responsible for the (development and production) infrastructure and required configurations. Ensures that the development framework works properly. Leads the preparation of the deployment machine(s)/containers, git repository, cloud infrastructure, databases operations, etc.
Developer (Everyone)	Contribute to the development tasks which can be tracked by monitoring the pull requests/commits in the team repository

1.2 Agile backlog management and work assignment

In our project, we follow agile practices for backlog management, focusing on user stories, and utilize Jira for creating epics and tasks. Each task is assigned to an individual team member, and we create branches for each subtask to facilitate efficient development.

Each person is assigned or assigns themselves a task from the current spring. That person then creates a branch for the specific task and is then responsible for creating the pull request and asking for a review. After that, that person is also responsible for setting the status of the task in Jira.

2 Code quality management

2.1 Guidelines for contributors (coding style)

When creating a pull request we have a template that should be filled by the user to give proper information about it.

2.2 Code quality metrics and dashboards

For static code analysis we use SonarClound in all Repositories of the project.
Code coverage:

- API-springB Repository:
  - Uses coverage of 80% to the quality gate
- S-C-DS-Nextjs Repository:
  - Uses coverage of 60% to the quality gate
- G-M-APP Repository:
  - Uses coverage of 0% to the quality gate due to lack of time of implementing tests.

The other metrics created from SonarCloud are using the default:

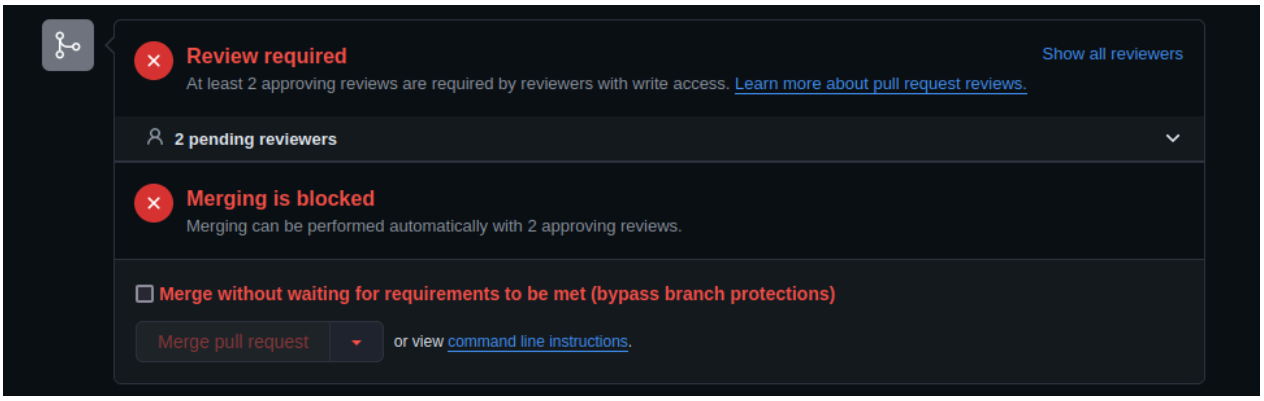
- The reliability,security and maintainability rating is A.
- The limited duplication is 3%.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

We follow the “github flow” where each person creates a branch, makes their changes, creates a pull request, addresses the comments, merges the branch and then deletes it. We use this together with

Jira where each user creates their branch and pull requests from one of the user stories. After a pull request is created at least one review is required (2 if the pull request is going to the main branch), allowing other developers to spot potential problems with the code.

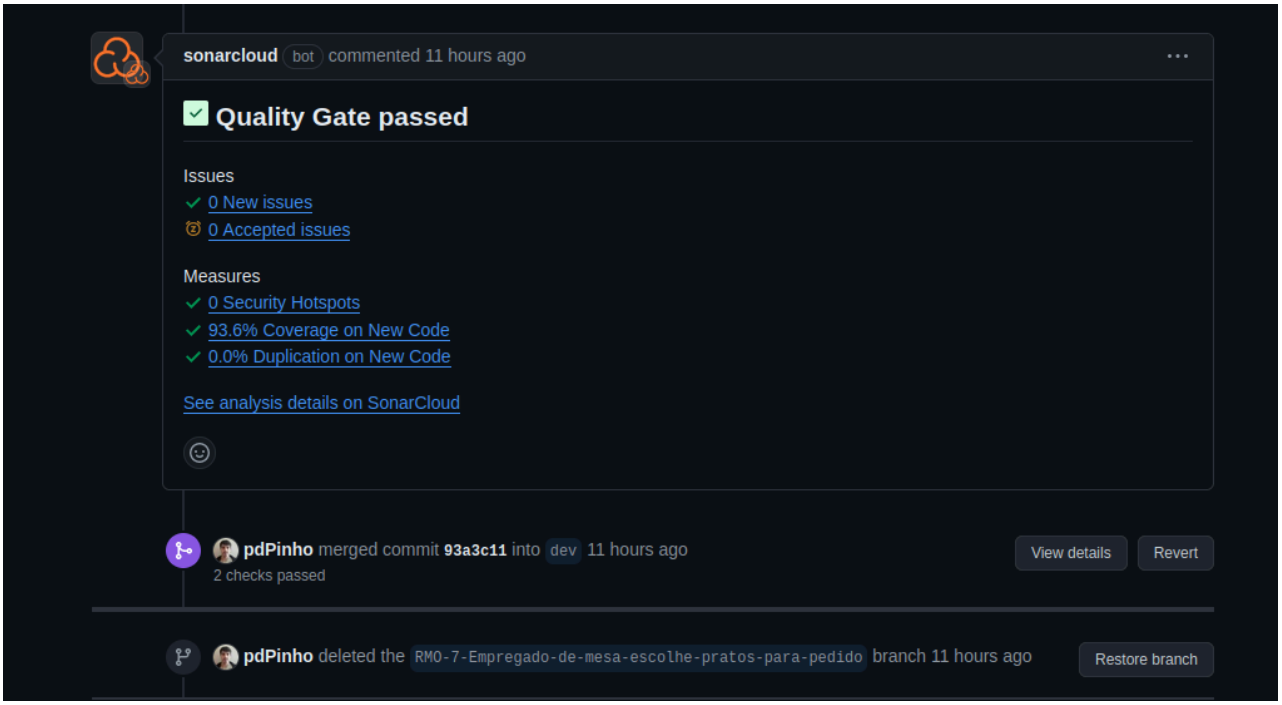


Pull Request - branch protection example

For a user story to be considered done it has to have unitary tests that pass the quality standards set by SonarCloud. Not only that but they must also be reviewed by other developers.

3.2 CI/CD pipeline and tools

In terms of Continuous Integration we have github workflows that run the tests everytime there is a push to either the main or dev branches or when there is a pull request. Not only this but we also have a workflow for running static code analysis with Sonar Cloud.



Pull Request - quality gate example

For Continuous Deployment we have a github runner in the provided Virtual Machine that updates the repository and runs the dockers, deploying the system everytime there is a push to the main branch. While we didn't have time to implement this, a good idea would be to also add some sort of github action that would call this deployment action everytime there is an update to the submodules.

We also have an action that when the mobile frontend repository receives any updates in the main branch a new apk is generated.

3.3 System observability

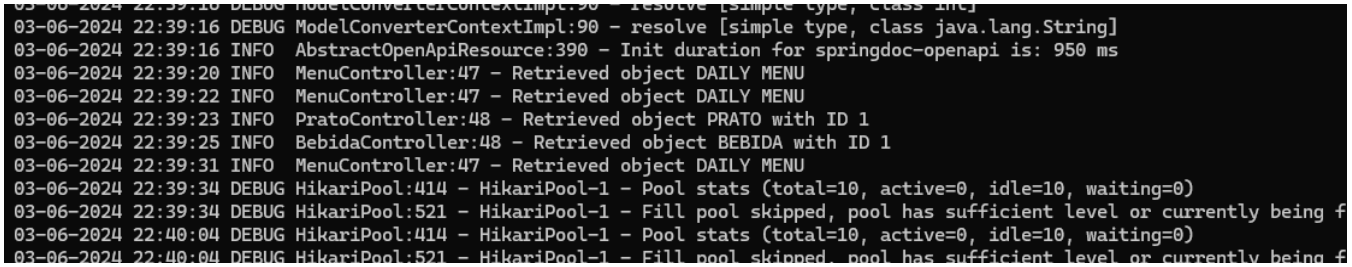
To better debug the application and understand why certain actions failed or issues occurred, we have implemented logging using SLF4J within our Controllers and Services. These logs provide information about the application's behavior to help us understand problems efficiently.

Our application uses two types of logs: INFO and WARN.

- **INFO Logs**: We use these logs to notify us when a user successfully retrieves a product by its ID, when a booking has been updated, when anything goes as planned.

- **WARN Logs**: We use these to indicate problems such as failed authentication attempts, attempts to access non-existent resources, or failed operations due to insufficient stock. For example, when a user tries to book a table with more people than the restaurant can fit, or when a request asks for more stock than it exists.

  These logs are saved within a file named **tqslogs.log**

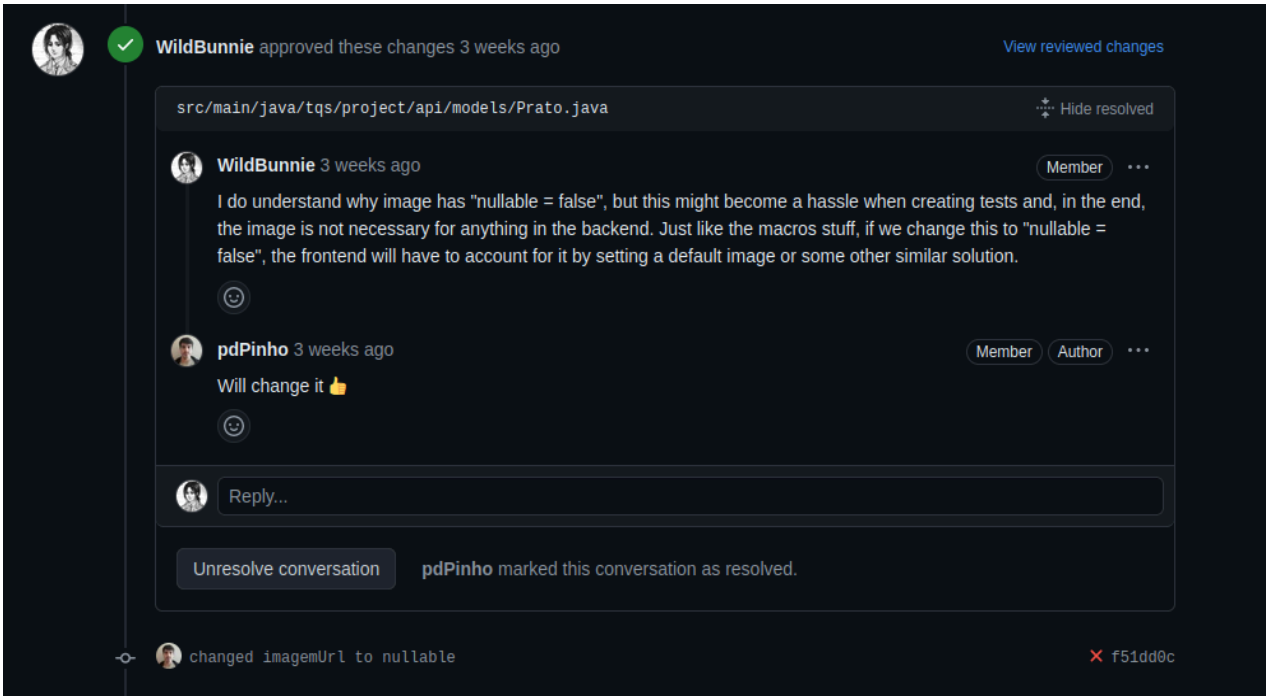


tqslogs.log - log example

4 Software testing

4.1 Overall strategy for testing

The test development strategy was test-driven-development throughout the entire project. This made it so that logic implementations were not needed to be fully implemented before actually testing logic. Such as using null returns at the beginning. Tests are necessary to be added with a pull request to be peer reviewed and here the code coverage is tracked with Sonar cloud.



Pull Request - PR example review

REST-Assured was used to mock http requests, making it easy to not only read but also test. In the frontend we mostly use jest for mocking purposes.

4.2 Functional testing/acceptance

Due to time restrictions it was not possible to implement any functional tests.

4.3 Unit tests

Unit testing was used throughout our application be it frontend and/or backend, guaranteeing that individual components functioned correctly. This helped to quickly debug certain logic as these areas were isolated and well protected from others, by the usage of mocks.
Even though most of the backend's code contains unit testing, login aspects were not tested due to even after having tried to test with filters and userMocks, it did not yield quick results and time was of the essence.

- **Backend:** Utilizing JUnit, Mockito and REST-Assured, we were capable of testing these components in an isolated manner.
    - **Controllers**: To begin with, we started with our controllers, mocking the service and eventually our authenticationfilter, using @WebMvcTest, forcing it to only load a small amount of the application.

```java
@WebMvcTest(BebidaController.class)
@AutoConfigureMockMvc(addFilters = false)
class BebidaControllerTest {

    @Autowired
    MockMvc mvc;

    @MockBean
    BebidaService service;

    @MockBean
    JwtAuthenticationFilter jwtAuthenticationFilter;

    Bebida beverage = new Bebida();

    @BeforeEach
    void setUp(){
        beverage.setId(1L);
        beverage.setNome("Fanta");
    }

    @Test
    void givenBebida_whenGetBebidaById_thenReturnBebida(){
        when(service.getBebida(1L)).thenReturn(beverage);

        RestAssuredMockMvc
            .given()
                .mockMvc(mvc)
                .contentType(ContentType.JSON)
            .when()
                .get("/api/beverages/1")
            .then()
                .statusCode(HttpStatus.SC_OK)
                .assertThat()
                .body("nome", equalTo("Fanta"));

        verify(service, times(1)).getBebida(1L);
    }
```

Controller Test - example

- **Services**: To test our services we mocked the repository and simply tested the service's behavior, ensuring its logic was correct.

```java
@ExtendWith(MockitoExtension.class)
public class BebidaServiceTest {

    @Mock(strictness = Mock.Strictness.LENIENT )
    BebidaRepository repository;

    @InjectMocks
    private BebidaServiceImpl service;

    Bebida beverage = new Bebida();

    @BeforeEach
    void setUp(){
        beverage.setId(1L);
        beverage.setNome("Fanta");
        beverage.setPreco(2.5);
        beverage.setStock(10);

        when(repository.findById(1L)).thenReturn(Optional.of(beverage));
    }

    @Test
    void whenGetBebidaById_thenReturnBebida() {
        Bebida found = service.getBebida(beverage.getId());

        verify(repository, times(1)).findById(Mockito.any());
        assertThat(found.getNome()).isEqualTo("Fanta");
    }
```

Service Test - example

- **Repositories**: To test our repositories @JpaDataTest was used alongside EntityManagers, making it easy to verify the repository's logic. These tests were only aimed at useful, practical ones, avoiding creating unnecessary ones, polluting the test file with things such as CRUD operations in situations where no full CRUD was used (not taking DataInitializer into account).

```java
@DataJpaTest
class BebidaRepositoryTest {
    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private BebidaRepository repository;

    Bebida beverage = new Bebida();

    @BeforeEach
    void setUp(){
        beverage.setNome("Fanta");
        beverage.setPreco(2.5);
        beverage.setStock(10);

        entityManager.persistAndFlush(beverage);
    }

    @Test
    void givenBebida_whenFindBebidaById_thenReturnBebida(){
        Bebida found = repository.findById(beverage.getId()).get();

        assertThat(found.getNome()).isEqualTo("Fanta");
    }
```

Repository Test - example

- **Frontend:** For the pages and components we utilize jest to mock and test their behavior. For this we mock the results of the http requests that are sent to the backend to verify that the pages show the expected results. We also test for edge cases such as a user not being logged in, or not having permission to enter a webpage.

```
jest.mock("next/navigation", () => ({
  useRouter() {
    return {
      route: "/",
      pathname: "",
      query: "",
      asPath: "",
      push: jest.fn(),
      events: {
        on: jest.fn(),
        off: jest.fn(),
      },
      beforePopState: jest.fn(() => null),
      prefetch: jest.fn(() => null),
    };
  },
}));

describe("Prato page Component", () => {
  it("data is received", async () => {

    (global as any).fetch = jest.fn(() =>
      Promise.resolve({
        status: 200,
        json: () => Promise.resolve(bebida),
      })
    );

    let pagina = await BebidaPage({ params: { bebidaID: 1 } });

    render(pagina);

    expect(screen.getByText(`€${bebida.preco.toFixed(2)}`)).toBeInTheDocument();
    expect(screen.getByText(`${bebida.kcal}`)).toBeInTheDocument();
    expect(screen.getByText(`${bebida.hidratosCarbono}g`)).toBeInTheDocument();
    expect(screen.getByText(`${bebida.proteina}g`)).toBeInTheDocument();
    expect(screen.getByAltText(bebida.nome)).toHaveAttribute('src', `${process.env.NEXT_PUBLIC_IP_ADDRESS}${bebida.imagemUrl}`);
  });
```

Jest Test - example

4.4 System and integration testing

Due to time restrictions it was not possible to implement any integration tests.