deti universidade de aveiro
departamento de eletrónica, telecomunicações e informática

HW1: Mid-term assignment report

Hugo Gonçalo Lopes Castro [113889], v2025-03-26

1 Introduction

1.1 Overview of the work

Este relatório apresenta o trabalho individual para a disciplina de TQS (Teste e Qualidade de Software). O projeto consiste numa aplicação web para gestão de refeições num restaurante universitário, desenvolvida utilizando Spring Boot para o backend e uma interface web moderna e responsiva para o frontend.

A aplicação permite que os utilizadores visualizem o menu semanal, façam reservas de refeições, gerem as suas reservas e visualizem estatísticas de utilização do sistema. O sistema foi desenvolvido com foco na qualidade, testabilidade e desempenho, seguindo boas práticas de desenvolvimento de software.

1.2 Current limitations

Atualmente, o sistema apresenta algumas limitações conhecidas:

- Cobertura de Testes
    - Apesar da implementação abrangente de testes unitários, de integração e funcionais, o sistema apresenta 0% de cobertura de código reportada. Este é um

problema conhecido que está a ser investigado, possivelmente relacionado com a configuração do SonarQube.

- Funcionalidades Avançadas de Reserva
  - O sistema não implementa ainda funcionalidades avançadas de reserva como:
    - Reserva de opções específicas de menu (com limitações de quantidade)
    - Reservas em grupo
    - Gestão de preferências alimentares
    - Sistema de fila de espera para refeições populares
- Quality Gates em CI/CD
  - O pipeline de integração contínua não implementa ainda:
    - Enforçamento automático de quality gates
    - Bloqueio de merge em caso de falha de qualidade
    - Análise automática de vulnerabilidades
    - Verificação automática de dependências desatualizadas

2 Product specification

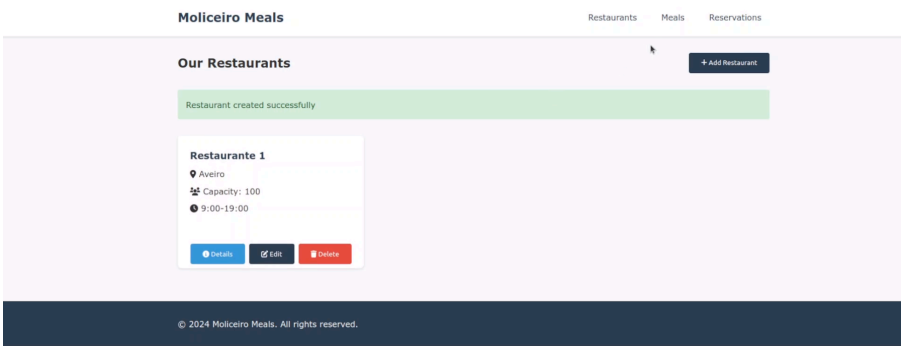
2.1 Functional scope and supported interactions

O sistema é utilizado por dois tipos principais de atores:

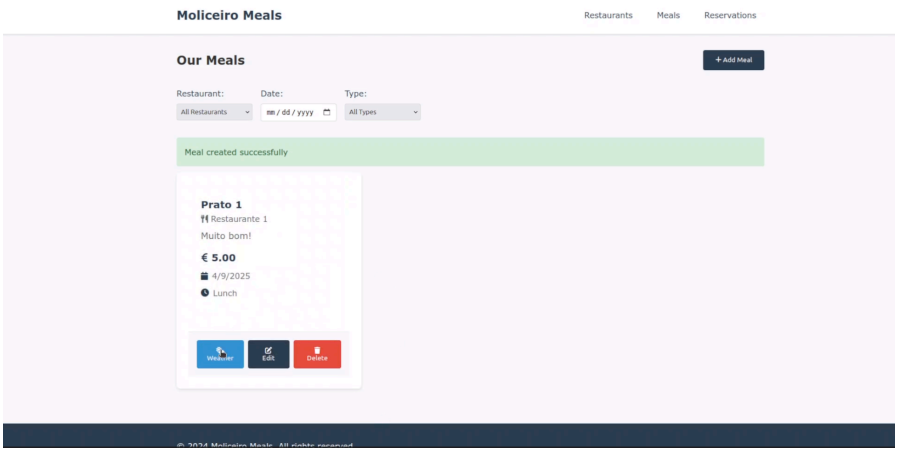
- Utilizadores do Restaurante:
  - Visualizar o menu semanal
  - Fazer reservas de refeições
  - Gerir as suas reservas existentes
  - Visualizar o estado das suas reservas
- Administradores do Sistema:
  - Gerir o menu semanal
  - Visualizar estatísticas de utilização do sistema
  - Monitorar o desempenho da cache

As principais interações incluem:

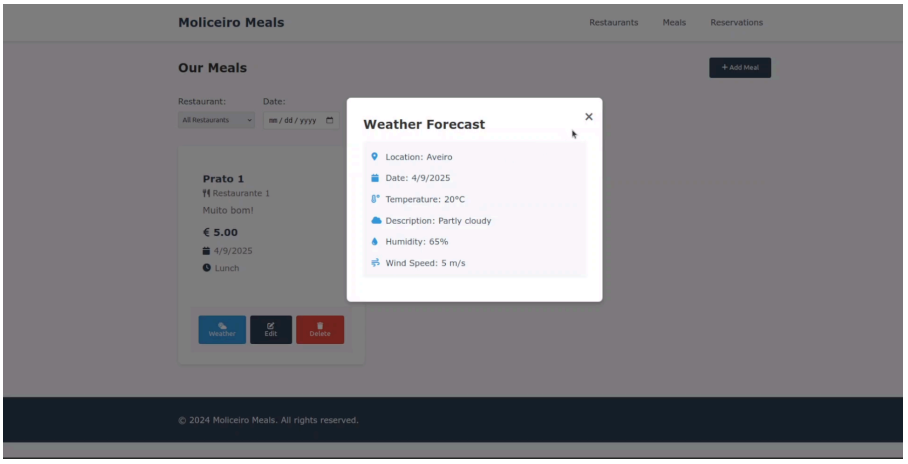
- Navegação pelo menu semanal
- Seleção de refeições para reserva
- Visualização e gestão de reservas
- Acesso a estatísticas de utilização do sistema



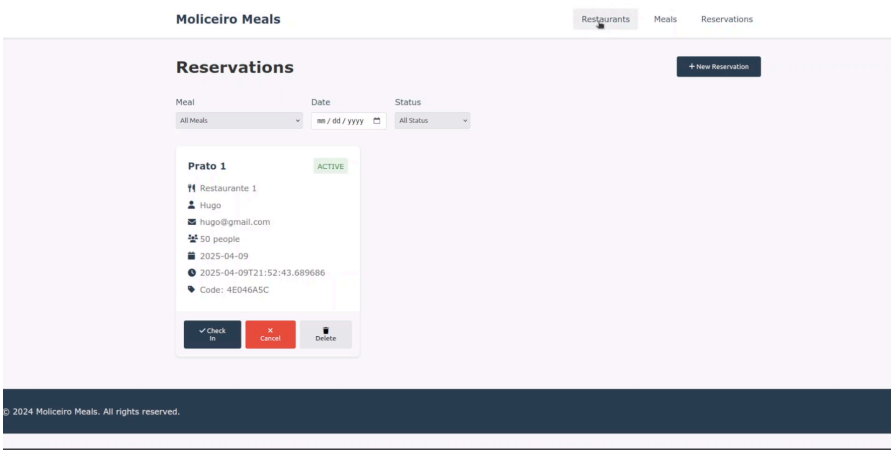
Página de restaurantes



Página de refeições



Página de meteorologia da refeição



Página de reservas

Video demo available at: https://github.com/hujuc/TQS_HW1/blob/main/docs/videoDemo.mp4

2.2 System implementation architecture

A aplicação foi desenvolvida seguindo uma arquitetura em camadas, com uma clara separação de responsabilidades e componentes. No frontend, a aplicação utiliza HTML5, CSS3 e JavaScript para criar uma interface web moderna e responsiva, garantindo uma boa experiência de utilizador em diferentes dispositivos. A interface foi desenhada para ser intuitiva e acessível, seguindo as melhores práticas de design web.

No backend, a aplicação foi construída utilizando Spring Boot 3.2.3, que fornece uma base robusta para o desenvolvimento de aplicações web. A arquitetura REST foi adotada para a API, permitindo uma comunicação clara e padronizada entre o frontend e o backend. Para melhorar o desempenho, foi implementado um sistema de cache distribuído utilizando Redis, que ajuda a reduzir a carga na base de dados e a melhorar os tempos de resposta.

A persistência de dados é gerida através do JPA/Hibernate, que fornece uma camada de abstração sobre a base de dados PostgreSQL. Esta escolha permite uma gestão eficiente dos dados, com suporte para transações, relacionamentos e consultas complexas. A base de dados foi desenhada para suportar as necessidades do sistema, com tabelas para refeições, reservas, restaurantes e previsões meteorológicas.

A infraestrutura da aplicação foi containerizada utilizando Docker, o que facilita a implantação e escalabilidade do sistema. O Docker Compose é utilizado para orquestrar os diferentes serviços, incluindo a aplicação Spring Boot, a base de dados PostgreSQL, o Redis para cache e o Prometheus para monitorização. Esta abordagem permite uma fácil configuração do ambiente de desenvolvimento e produção.

A gestão de dependências é feita através do Maven, que garante que todas as bibliotecas e frameworks necessários estão corretamente versionados e disponíveis. O controlo de versão é gerido através do Git, permitindo um desenvolvimento colaborativo e um histórico claro das alterações.

Para garantir a qualidade e fiabilidade do sistema, foram implementados testes em diferentes níveis, incluindo testes unitários, testes de integração e testes de performance. A monitorização do sistema é feita através do Prometheus e Grafana, que permitem visualizar métricas importantes como tempos de resposta, utilização de recursos e estatísticas de cache.

Esta arquitetura foi desenhada para ser escalável, mantendo um bom equilíbrio entre desempenho, manutenibilidade e facilidade de desenvolvimento. A separação clara entre componentes permite uma evolução independente de cada parte do sistema, facilitando futuras melhorias e adaptações.

2.3 API for developers

A API do sistema foi desenvolvida seguindo os princípios REST e está documentada utilizando o OpenAPI (Swagger). A API está organizada em dois grupos principais de endpoints:

1. **Problem Domain Endpoints**

Estes endpoints permitem a gestão do domínio principal da aplicação - a gestão de refeições e reservas no restaurante universitário. Incluem operações para:

- Gestão de refeições (criação, consulta, atualização e remoção)
- Gestão de reservas (criação, consulta, cancelamento e check-in)
- Consulta de disponibilidade e menus
- Gestão de restaurantes

2. **Cache Statistics Endpoints**

Estes endpoints fornecem métricas sobre a utilização do sistema de cache, permitindo monitorizar:

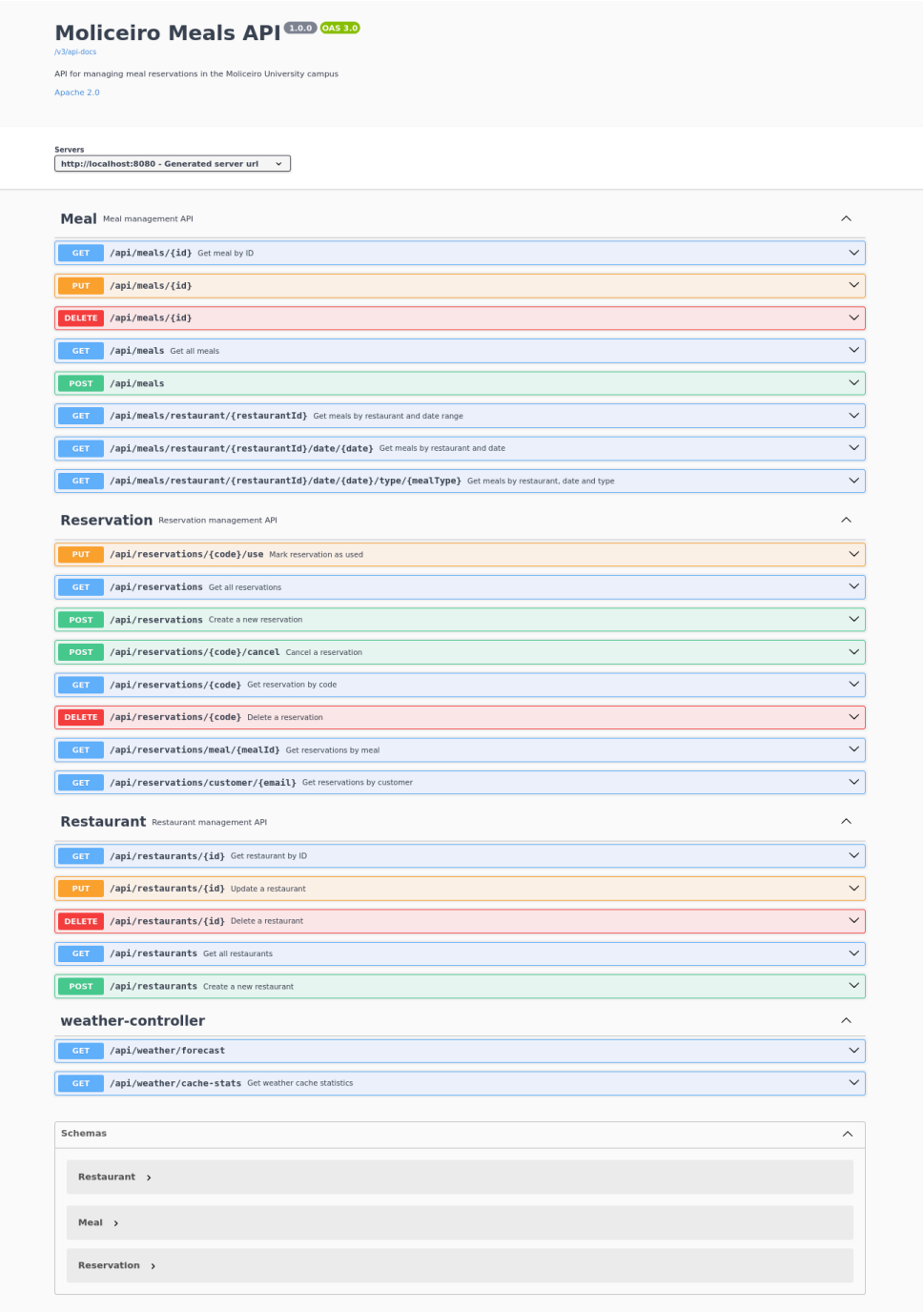
- Taxa de hit/miss do cache
- Tempo médio de resposta
- Estatísticas de utilização por tipo de recurso
- Métricas de desempenho

A documentação completa da API está disponível através do Swagger UI em /swagger-ui.html, onde os desenvolvedores podem:

- Explorar todos os endpoints disponíveis
- Testar as operações diretamente na interface
- Ver exemplos de requisições e respostas
- Consultar os modelos de dados utilizados

A API segue as melhores práticas de REST, incluindo:

- Uso adequado de verbos HTTP (GET, POST, PUT, DELETE)
- Respostas HTTP apropriadas (200, 201, 400, 404, etc.)
- Validação de inputs
- Documentação clara e consistente



API Endpoints usando Swagger

3 Quality assurance

3.1 Overall strategy for testing

A estratégia de testes implementada neste projeto segue uma abordagem em camadas, combinando diferentes tipos de testes para garantir uma cobertura abrangente da aplicação. A estratégia foi desenvolvida considerando os seguintes princípios:

1. **Testes Unitários**

- Foco em componentes individuais (serviços e controladores)

- Utilização de JUnit 5 e Mockito para isolamento
- Cobertura de casos de sucesso e erro
- Validação de regras de negócio

2. **Testes de Integração**

- Foco na interação com a base de dados
- Utilização de TestContainers para ambiente real
- Validação de fluxos de persistência
- Teste de transações e relacionamentos

3. **Testes Funcionais**

- Testes end-to-end com Selenium
- Validação de fluxos completos de negócio
- Simulação de interação do utilizador
- Verificação de interface web

4. **Testes de Performance**

- Testes de carga com k6
- Análise de tempos de resposta
- Teste de concorrência
- Monitorização com Prometheus

3.2 Unit and integration testing

Os testes foram implementados em diferentes camadas da aplicação, seguindo uma estratégia de teste em camadas:

Testes Unitários

Os testes unitários foram implementados em duas camadas principais:

1. **Camada de Serviço**

Testes como MealServiceTest.java validam a lógica de negócio isoladamente, utilizando mocks para as dependências:

```
@Test
void whenGetAllMeals_thenReturnAllMeals() {
        // Arrange
        List<Meal> expectedMeals = Arrays.asList(testMeal);
        when(mealRepository.findAll()).thenReturn(expectedMeals);

        // Act
        List<Meal> actualMeals = mealService.getAllMeals();
```

```
        // Assert
        assertThat(actualMeals).hasSize(1);
        assertThat(actualMeals.get(0)).isEqualTo(testMeal);
        verify(mealRepository, times(1)).findAll();
}
```

2. Camada de Controlador

Testes como MealControllerTest.java validam o comportamento da API REST:

```
@Test
void whenGetAllMeals_thenReturnMeals() {
        // Arrange
        when(mealService.getAllMeals()).thenReturn(testMeals);

        // Act
        ResponseEntity<List<Meal>> response = mealController.getAllMeals();

        // Assert
        assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
        assertThat(response.getBody()).isEqualTo(testMeals);
        verify(mealService, times(1)).getAllMeals();
}
```

Testes de Integração

Os testes de integração focam-se na interação com a base de dados, utilizando TestContainers para garantir um ambiente real de teste. Por exemplo, no MealRepositoryTest.java:

```
@SpringBootTest
@Testcontainers
class MealRepositoryTest {
        @Container
        static PostgreSQLContainer<?> postgres = new
PostgreSQLContainer<>("postgres:15-alpine")
                .withDatabaseName("testdb")
                .withUsername("test")
                .withPassword("test");

        @Test
        void whenFindByRestaurantIdAndDateBetween_thenReturnMeals() {
        // Save the meal
        mealRepository.save(testMeal);

        // Create another meal for the same restaurant
        Meal anotherMeal = new Meal();
        anotherMeal.setRestaurant(testRestaurant);
        anotherMeal.setName("Another Meal");
        anotherMeal.setDescription("Another Description");
        anotherMeal.setPrice(15.0);
        anotherMeal.setDate(today);
        anotherMeal.setMealType("dinner");
        mealRepository.save(anotherMeal);

        // Find meals by restaurant ID and date range
        List<Meal> foundMeals = mealRepository.findByRestaurantIdAndDateBetween(
                testRestaurant.getId(),
```

```
                today,
                today.plusDays(1)
        );

        // Verify results
        assertThat(foundMeals).hasSize(2);
        assertThat(foundMeals).extracting(Meal::getName)
                .containsExactlyInAnyOrder("Test Meal", "Another Meal");
        }
}
```

3.3 Functional testing

Os testes funcionais foram implementados em dois níveis:

- Testes de Controlador

Os testes de controlador validam o comportamento da API REST, garantindo que os endpoints respondem corretamente. Por exemplo, no MealControllerTest.java:

```
@Test
void whenGetAllMeals_thenReturnMeals() {
        // Arrange
        when(mealService.getAllMeals()).thenReturn(testMeals);

        // Act
        ResponseEntity<List<Meal>> response = mealController.getAllMeals();

        // Assert
        assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
        assertThat(response.getBody()).isEqualTo(testMeals);
        verify(mealService, times(1)).getAllMeals();
}
```

- Testes End-to-End com Selenium

Os testes end-to-end simulam a interação real do utilizador com a aplicação web. O TestFullMoliceiroMealsTest.java implementa um fluxo completo de negócio:

```
@Test
void testFullMoliceiroMeals() {
        // Navigate to home page
        driver.get("http://localhost:8080/");

        // Navigate to Restaurants page

wait.until(ExpectedConditions.elementToBeClickable(By.linkText("Restaurants"))).click();

        // Add new restaurant
        wait.until(ExpectedConditions.elementToBeClickable(By.id("addRestaurantBtn"))).click();

        // Fill restaurant details
```

```
        driver.findElement(By.id("name")).sendKeys("Burguer King");
        driver.findElement(By.id("location")).sendKeys("Espinho");
        driver.findElement(By.id("capacity")).sendKeys("100");
        driver.findElement(By.id("operatingHours")).sendKeys("9:00-19:00");
        driver.findElement(By.cssSelector(".btn:nth-child(5)")).click();
```

3.4 Non functional testing

Os testes não funcionais foram implementados utilizando k6 para testes de carga e performance. Foram considerados vários cenários de teste, com diferentes níveis de carga e duração:

Testes de Carga Progressiva
Implementados em test_stages.js, este teste simula um cenário de carga progressiva:

- Rampa de subida: 0 a 20 utilizadores virtuais em 5 segundos
- Pico de carga: 20 utilizadores virtuais durante 10 segundos
- Rampa de descida: 20 a 0 utilizadores virtuais em 5 segundos

Testes de Carga Sustentada
Implementados em test_stages2.js, este teste simula uma carga mais prolongada:

- Rampa de subida: 0 a 20 utilizadores virtuais em 30 segundos
- Pico de carga: 20 utilizadores virtuais durante 30 segundos
- Rampa de descida: 20 a 0 utilizadores virtuais em 30 segundos

Testes de Carga Extrema
Implementados em test_stages3.js, este teste simula uma carga mais intensa:

- Rampa de subida: 0 a 120 utilizadores virtuais em 30 segundos
- Pico de carga: 120 utilizadores virtuais durante 30 segundos
- Rampa de descida: 120 a 0 utilizadores virtuais em 30 segundos

Métricas e Thresholds
Para todos os testes, foram definidos os seguintes thresholds:

- Tempo de resposta: 95% das requisições devem completar em menos de 1.1 segundos
- Taxa de falha: menos de 1% das requisições podem falhar
- Taxa de sucesso dos checks: mais de 98% dos checks devem passar

Monitorização
Os testes incluem monitorização através do Prometheus, permitindo a análise detalhada de:

- Tempo de resposta por endpoint
- Taxa de sucesso/falha
- Tamanho das respostas
- Utilização de recursos

Esta abordagem abrangente de testes não funcionais garante que a aplicação mantém o desempenho esperado mesmo sob diferentes níveis de carga, desde cenários normais até picos de utilização intensa.

Os resultados dos testes de performance mostram que a aplicação consegue lidar eficientemente com cargas significativas, mantendo tempos de resposta consistentes e baixas taxas de erro. A implementação de cache com Redis e a otimização das consultas à base de dados contribuem para este bom desempenho.

A monitorização através do Prometheus e Grafana permite visualizar em tempo real o comportamento da aplicação durante os testes de carga, facilitando a identificação de gargalos e a otimização do sistema.

3.5 Code quality analysis

A análise de qualidade do código foi realizada utilizando o SonarQube como ferramenta principal de análise estática. A configuração do SonarQube foi implementada através do plugin Maven e um ficheiro sonar-project.properties dedicado.

Ferramentas e Workflow

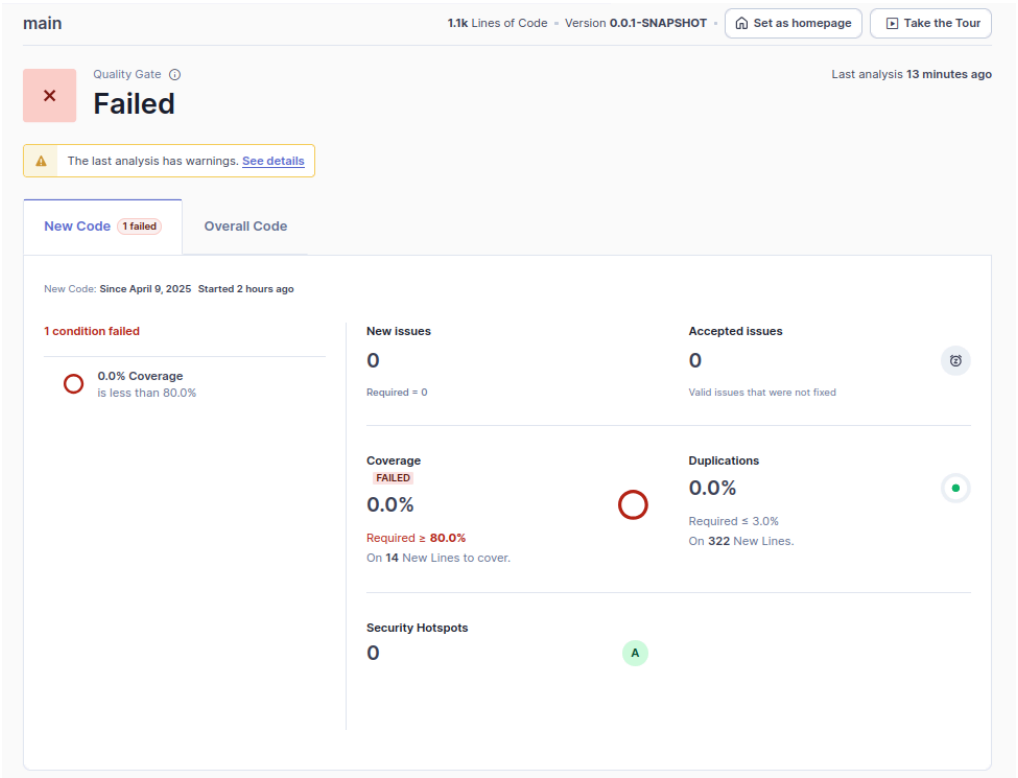
SonarQube

- Configurado através do plugin Maven (sonar-maven-plugin versão 3.10.0.2594)
- Análise automática durante o build do projeto
- Configuração específica no sonar-project.properties:

```
sonar.projectKey=moliceiro
sonar.projectName=Moliceiro
sonar.projectVersion=1.0
sonar.sources=src/main/java
sonar.tests=src/test/java
```

Lombok

- Utilizado para reduzir boilerplate code
- Configurado no pom.xml com processamento de anotações



Interface do dashboard do SonarQube

4 References & resources

Project resources

Resource:	URL/location:
Git repository	https://github.com/hujuc/TQS_HW1
Video demo	https://github.com/hujuc/TQS_HW1/blob/main/docs/videoDemo.mp4

Reference materials

Documentação Spring Boot: https://docs.spring.io/spring-boot/docs/current/reference/html/
Documentação JUnit 5: https://junit.org/junit5/docs/current/user-guide/
Documentação SonarQube: https://docs.sonarqube.org/latest/
Documentação Redis: https://redis.io/documentation