HW1: Mid-term assignment report

Ana Alexandra Antunes [876543], v2025-10-22

1 Introduction

1.1 Overview of the work

Este relatório apresenta o projeto individual intermédio da unidade curricular de TQS, abordando as funcionalidades do produto de software e a estratégia de garantia de qualidade adotada. O objetivo foi desenvolver uma web app - ZeroMonos - capaz de simplificar e tornar mais eficiente o processo de agendamento e gestão de recolha de resíduos volumosos, que respondem a necessidades reais dos municípios e dos cidadãos.

A solução desenvolvida oferece uma interface simples e agradável de utilizar. Através dela, os utilizadores podem criar pedidos de recolha, acompanhar o estado das suas reservas e aceder às funcionalidades próprias do seu perfil — seja como cidadão, seja como gestor de staff.

Para garantir a qualidade do software, foi adotada uma arquitetura composta por um backend em Spring Boot e um frontend em React, complementada por uma estratégia de testes que inclui testes unitários, de integração e testes orientados a comportamento (BDD). Além disso, foram utilizadas métricas de cobertura, processos de integração contínua e uma

auditoria da interface para assegurar estabilidade, consistência e fiabilidade ao longo do desenvolvimento.

A abordagem seguida teve como prioridade criar um sistema modular, fácil de testar e capaz de crescer de forma consistente. Deste modo, assegurou-se também o cumprimento de todos os requisitos definidos no enunciado do trabalho.

Ao longo deste relatório, serão apresentadas as principais funcionalidades do sistema, a arquitetura desenvolvida, a metodologia de testes e a análise da qualidade realizada, bem como a forma como o trabalho se articula com os objetivos pedagógicos da unidade curricular.

1.2 Current implementation (faults & extras)

Foram identificados alguns pontos que podem ser melhorados em desenvolvimentos futuros:

- Não existe, ainda, separação de perfis entre staff e cliente, nem autenticação de utilizadores. Isto reduz o controlo sobre permissões e limita a personalização da experiência.
- O cancelamento automático de reservas expiradas não foi implementado, sendo necessária intervenção manual.
- Ainda não existem validações para impedir reservas duplicadas, o que pode originar inconsistências.
- O histórico de alterações de estado funciona atualmente de forma simples e não pode ser exportado para consulta externa.

Para além dos requisitos principais, foram incluídas funcionalidades extra que melhoram a usabilidade e flexibilidade do sistema:

- Possibilidade de filtrar reservas por município e data, tornando a pesquisa mais eficiente.
- A lista de municípios é carregada dinamicamente a partir de uma API, assegurando que os dados se mantêm atualizados.
- O frontend foi auditado em acessibilidade e performance através do Lighthouse.
- O sistema foi preparado para extensibilidade em cenários de teste BDD utilizando Cucumber.

De forma global, o sistema implementa corretamente os fluxos essenciais definidos no enunciado e encontra-se preparado para evoluir, tanto em termos de funcionalidades, como na melhoria contínua da qualidade do código.

1.3 Use of generative AI

Durante o desenvolvimento do projeto, a inteligência artificial generativa foi utilizada apenas como apoio, com o objetivo de agilizar algumas tarefas tanto no backend (Spring Boot) como

no frontend (React). Nunca descartando a minha revisão do código, e correções posteriormente feitas a esse código.

No entanto, todos os testes — unitários, de integração e BDD — foram desenvolvidos manualmente. Não foi utilizada IA para criar casos de teste, definir critérios de validação ou tratar cenários de erro. Esta opção teve como propósito assegurar uma compreensão completa dos fluxos de negócio, fortalecer o domínio das ferramentas de teste exigidas na unidade curricular e manter o trabalho coerente com os objetivos pedagógicos definidos.

2 Product specification

2.1 Functional scope and supported interactions

Cidadão

- O cidadão utiliza a aplicação para agendar a recolha de resíduos volumosos, quer ao domicílio quer no município.
- Para efetuar o pedido, preenche um formulário indicando o município, a data desejada, o período e uma breve descrição dos resíduos a recolher.
- Após o registo, recebe um **token de reserva**, que lhe permite consultar e acompanhar o estado do pedido ao longo do tempo.
- O cidadão pode ainda pesquisar as suas reservas, ver os detalhes associados e cancelar o pedido antes da recolha, caso seja necessário.

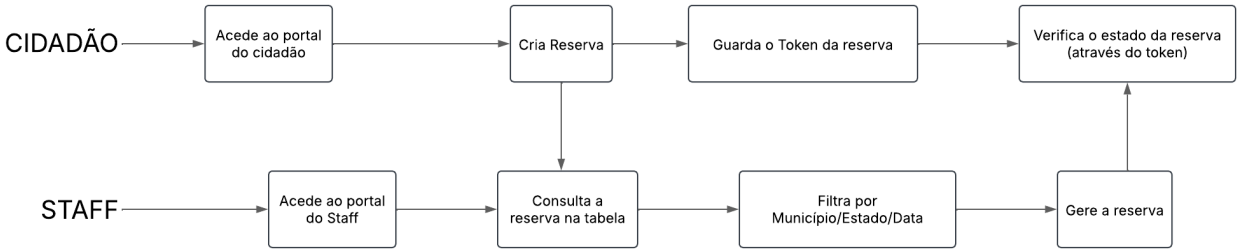
Staff

- O staff tem acesso a todas as reservas submetidas pelos cidadãos.
- Pode filtrar e gerir os pedidos por município, data ou estado, garantindo assim uma organização eficiente das recolhas.
- É responsável por atualizar o estado das reservas (por exemplo: *recebido*, *em processamento*, *concluído*) e consultar rapidamente o histórico de alterações de cada pedido.
- Além disso, assegura o bom funcionamento do sistema e presta apoio ao cidadão quando solicitado.

Interações Principais

Ator	Ações Disponíveis
Cidadão	- Criar nova reserva - Consultar reserva existente através do token - Cancelar reserva antes da recolha - Acompanhar o estado atual do pedido

Staff	- Visualizar lista completa de reservas - Filtrar pedidos por município e/ou data - Atualizar o estado de uma reserva - Consultar histórico de alterações associado a cada pedido



2.2 System implementation architecture

O sistema foi desenvolvido seguindo uma arquitetura moderna em camadas, separando claramente o backend, o frontend e a comunicação entre ambos através de uma API RESTful. Esta abordagem garante uma boa organização lógica, facilita a manutenção e permite que cada parte evolua de forma independente.

Backend

- O backend foi implementado em Java, utilizando Spring Boot como framework principal. A estrutura segue o padrão clássico de camadas:
- Controller – recebe os pedidos via HTTP e encaminha para os serviços.
- Service – contém a lógica de negócio e validações aplicadas aos pedidos.
- Repository – responsável pelo acesso à base de dados através de JPA/Hibernate.
- A persistência é gerida com Spring Data JPA, simplificando operações CRUD sobre entidades como Booking, StateChange e Municipality.
- A API REST foi definida de forma clara, com endpoints bem organizados e suporte à serialização JSON através de Jackson.
- A gestão de dependências é feita pelo ficheiro pom.xml, incluindo os módulos necessários (Web, JPA, Testes, entre outros).

Frontend

- O frontend foi desenvolvido em React, com componentes modulares e páginas dedicadas às principais funcionalidades (por exemplo: BookingForm, BookingTable, HomePage).
- A comunicação com o backend é feita através de Axios, seguindo o formato JSON.
- A interface é responsiva e focada em clareza e facilidade de uso, incluindo filtros de pesquisa para facilitar a gestão dos pedidos.
- O sistema de build utiliza Vite, garantindo tempos de desenvolvimento rápidos e boa performance em produção.

Integração entre Frontend e Backend

- A interação entre as duas camadas é feita através de chamadas HTTP à API REST do backend.
- Este desacoplamento permite testar e manter cada componente de forma independente, melhorando a escalabilidade e a robustez do sistema.

Testes e Ferramentas de Qualidade

- Foram implementados testes:
- Unitários
- Integração
- BDD (Cucumber)

Todos desenvolvidos em Java, com JUnit e Mockito a suportarem a lógica de validação.

- A interface foi analisada com Lighthouse, garantindo boas métricas de acessibilidade e desempenho, e a qualidade do código foi acompanhada via SonarCloud.
- Documentação adicional encontra-se disponível em README e páginas de apoio (HELP).

2.3 API for developers

A API disponibilizada pelo sistema segue o padrão RESTful, permitindo que outros serviços ou aplicações possam integrar, automatizar ou estender as funcionalidades de gestão de recolha de resíduos volumosos. A estrutura dos endpoints foi pensada para ser clara, consistente e simples de utilizar, facilitando o desenvolvimento de integrações externas.

Principais Serviços Disponíveis na API

A API expõe operações que cobrem todo o ciclo de vida de uma reserva:

- Criar um pedido de recolha
  Permite ao utilizador submeter uma nova reserva, indicando município, data, período do dia e descrição dos resíduos.

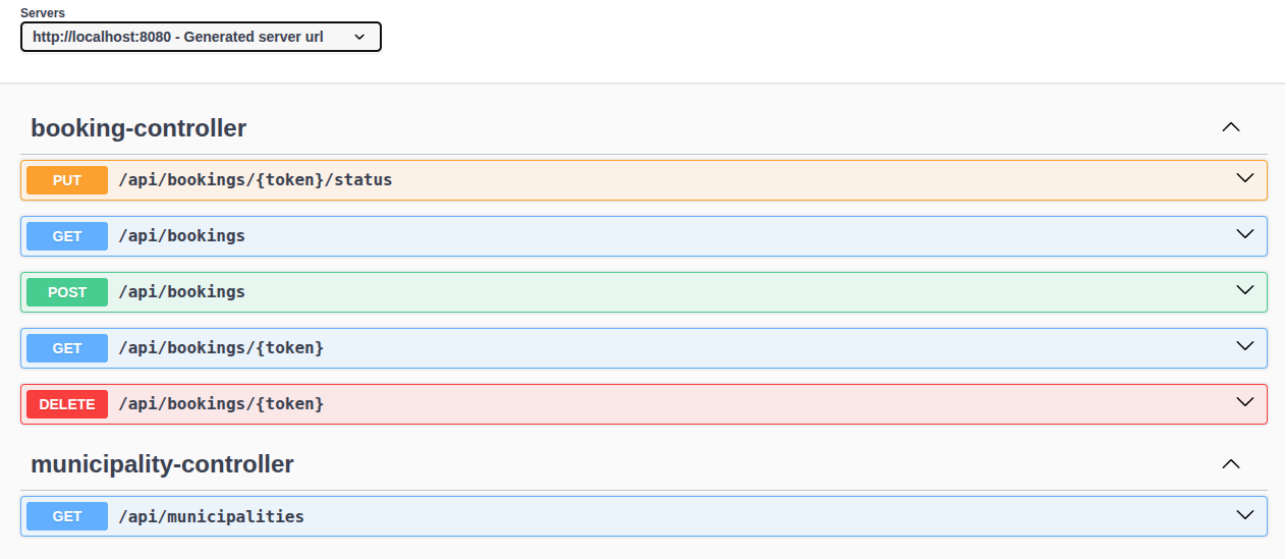
- Consultar uma reserva através do token
  Cada pedido gera um identificador único, que pode ser usado para acompanhar o estado ou obter detalhes.

- Pesquisar reservas com filtros avançados
  Possibilidade de listar reservas por município, data ou estado.

- Atualizar o estado de um pedido
  Utilizado pelo staff para marcar pedidos como recebidos, em processamento ou concluídos, podendo adicionar notas internas.

- Cancelar reservas
  Permite ao cidadão cancelar o pedido antes da recolha, caso já não seja necessário.

- Obter lista de municípios disponíveis
  A lista é carregada dinamicamente, garantindo sincronização com os dados reais do serviço.



3 Quality assurance

3.1 Overall strategy for testing

A estratégia de testes adotada no projeto seguiu uma abordagem equilibrada, orientada para garantir clareza, cobertura e fiabilidade nas diferentes camadas da aplicação. O objetivo principal foi assegurar que as funcionalidades essenciais funcionam conforme o esperado, tanto isoladamente como em conjunto.

Testes Unitários

- Foram desenvolvidos testes unitários para os métodos de lógica de negócio e funções utilitárias, utilizando JUnit e Mockito.
- A utilização de mocks permitiu isolar componentes e validar o comportamento de serviços como o BookingService sem dependência da base de dados.
- O foco esteve na verificação de regras de negócio e na deteção precoce de falhas lógicas.

Testes de Integração

- Recorreu-se ao Spring Boot Test para validar o funcionamento conjunto das camadas Controller → Service → Repository, incluindo persistência real.
- Estes testes verificam o ciclo completo de uma operação, garantindo que os endpoints REST devolvem as respostas esperadas e que os dados são persistidos corretamente.
- A abordagem assegura que a aplicação funciona como um todo e não apenas em módulos isolados.

Testes de Aceitação / BDD

- A nível funcional, foi utilizado o Cucumber, permitindo descrever cenários em linguagem natural.
- Os ficheiros .feature cobrem os principais fluxos de interação do cidadão e do staff, validando o comportamento da aplicação ao nível do utilizador final.
- Esta abordagem facilita o alinhamento com os requisitos do enunciado e a rastreabilidade dos comportamentos testados.

Ferramentas e Práticas Complementares

- A API foi testada e validada manualmente durante o desenvolvimento com Postman.
- A qualidade do código e cobertura de testes foram monitorizadas com SonarCloud.
- Embora não tenha sido aplicado TDD de forma estrita, os testes foram desenvolvidos em paralelo com as funcionalidades, reduzindo regressões e erros em fase tardia.

3.2 Unit and integration testing

Testes Unitários

Os testes unitários foram utilizados para validar a lógica de negócio do serviço de reservas, garantindo que cada método funciona corretamente de forma isolada. Para isso, recorreu-se ao JUnit em conjunto com Mockito, permitindo simular dependências externas como repositórios e serviços relacionados com municípios, sem necessidade de acesso real à base de dados.

Principais cenários:

- Impedir a criação de reservas com datas já passadas.
- Bloquear reservas quando a capacidade diária do município está esgotada.
- Aceitar pedidos válidos e rejeitar casos inválidos, tais como:
  - Município inexistente
  - Dados obrigatórios em falta
  - Limite de marcações atingido
- Cancelar reservas, verificando:
  - Erro ao tentar cancelar uma reserva já cancelada
  - Erro ao tentar cancelar uma reserva inexistente

Exemplos do ficheiro BookingServiceUnitTest.java:

```java
@Test
void naoPermiteBookingComDataPassada() {
    Booking booking = new Booking();
    booking.setMunicipality(municipality: "Braga");
    booking.setDate(LocalDate.now().minusDays(1)); // data passada
    booking.setTimeslot(timeslot: "Manhã");
    booking.setDescription(description: "Teste");

    when(municipalityService.getAllMunicipalities()).thenReturn(List.of("Braga"));

    Exception e = assertThrows(expectedType: IllegalArgumentException.class, () ->
        service.createBooking(booking)
    );
    assertTrue(e.getMessage().contains("Date cannot be in the past"));
}
```

```java
@Test
void naoPermiteCancelarBookingJaCancelado() {
    Booking booking = new Booking();
    booking.setToken(token: "test-token-456");
    booking.setStatus(BookingStatus.CANCELLED);

    when(bookingRepository.findByToken(token: "test-token-456"))
        .thenReturn(java.util.Optional.of(booking));

    Exception e = assertThrows(expectedType: IllegalArgumentException.class, () ->
        service.cancelBooking(token: "test-token-456")
    );
    assertTrue(e.getMessage().contains("already cancelled"));
}
```

Testes de Integração

Os testes de integração foram utilizados para validar o funcionamento conjunto das diferentes camadas da aplicação, desde a interface HTTP até à persistência de dados. Para tal, recorreu-se ao Spring Boot Test em combinação com MockMvc, permitindo simular pedidos reais à API sem necessidade de execução manual do servidor.
Estes testes verificam o ciclo completo da operação:

HTTP Request → Controller → Service → Repository → Base de Dados

Aspetos validados:

- Criação de reservas através de pedidos POST.
- Consulta de reservas utilizando o token único.
- Pesquisa com filtros por município, data ou estado.
- Atualização do estado de pedidos pelo staff.
- Cancelamento de reservas, garantindo regras de negócio.
- Verificação do comportamento do sistema perante:
  - Dados válidos (fluxo esperado)
  - Dados inválidos (erros e mensagens adequadas)

Exemplos do ficheiro BookingControllerITTest.java:

```java
@Test
@DisplayName("POST /api/bookings - BAD REQUEST (data passada)")
void whenCreateBookingWithPastDate_thenReturnsBadRequest() throws Exception {
    String bookingJson = """
        {
            "municipality": "Braga",
            "date": "2020-01-01",
            "timeslot": "Manhã",
            "description": "Data passada"
        }
        """;

    mvc.perform(post("/api/bookings")
            .contentType(MediaType.APPLICATION_JSON)
            .content(bookingJson))
            .andExpect(status().isBadRequest());
}
```

```java
@Test
@DisplayName("POST /api/bookings - OK")
void whenCreateValidBooking_thenReturnsOk() throws Exception {
    String bookingJson = """
        {
            "municipality": "Braga",
            "date": "2025-12-01",
            "timeslot": "Manhã",
            "description": "Teste integração"
        }
        """;

    mvc.perform(post("/api/bookings")
            .contentType(MediaType.APPLICATION_JSON)
            .content(bookingJson))
            .andExpect(status().isCreated())
            .andExpect(jsonPath("$.token").exists())
            .andExpect(jsonPath("$.municipality", is(value: "Braga")))
            .andExpect(jsonPath("$.timeslot", is(value: "Manhã")));
}
```

3.3 Acceptance testing

A validação final do sistema foi realizada através de testes de aceitação, seguindo a abordagem Behaviour-Driven Development (BDD) com recurso ao Cucumber. Estes testes complementam os unitários e de integração, garantindo que a aplicação funciona corretamente do ponto de vista do utilizador real.

Estratégia Geral

Os testes de aceitação simulam o fluxo completo de utilização do sistema, tanto para o perfil cidadão como para o staff, interagindo com a interface como num ambiente real.
O objetivo é confirmar que as funcionalidades cumprem os requisitos funcionais e que a experiência do utilizador é consistente.

Principais cenários cobertos:

- Submissão de um novo pedido de recolha, preenchendo município, data, período e descrição.
- Acompanhamento e consulta do estado de uma reserva através do token.
- Atualização e cancelamento de pedidos conforme as regras de negócio.
- Validação de mensagens e feedback do sistema em situações de erro:
  - Dados inválidos
  - Município sem vagas
  - Campos incompletos

Implementação e Automação

Os cenários de aceitação foram descritos em ficheiros .feature do Cucumber, utilizando linguagem natural para facilitar a leitura e ligação com os requisitos.
As step definitions foram implementadas em Java — por exemplo, em classes como CitizenBookingSteps.java — traduzindo cada passo dos cenários para ações executáveis.
A automação da interface foi realizada com Playwright, que executa as interações visuais no browser:

- Preenchimento de formulários
- Cliques em botões
- Navegação entre páginas
- Submissão e verificação de resultados

Cada cenário valida automaticamente o comportamento e o feedback exibido, garantindo correspondência com os critérios de aceitação definidos no enunciado.

Exemplo de CitizenBookingSteps.java e citizen-booking.feature:

```java
@Given("the citizen is on the homepage")
public void citizenOnHomepage() {
    page = PlaywrightContext.getPage();
    homePage = new HomePage(page);
    homePage.navigate();
    log.info(msg: "Citizen on homepage");
}

@When("the citizen navigates to the citizen page")
public void navigateToCitizenPage() {
    homePage.clickCitizenLink();
    citizenPage = new CitizenPage(page);
    assertTrue(page.url().contains("/citizen"));
    log.info(msg: "Navigated to citizen page");
}

@And("the citizen selects municipality {string}")
public void selectMunicipality(String municipality) {
    citizenPage.selectMunicipality(municipality);
    log.info(format: "Selected municipality: {}", municipality);
}
```

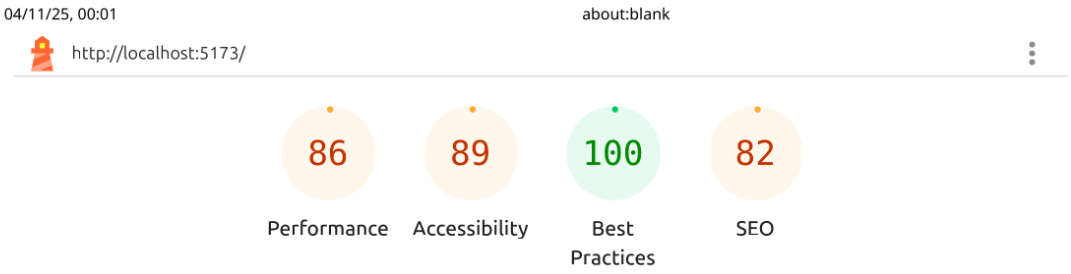
```
Feature: Citizen Booking Management
  As a citizen
  I want to create waste collection bookings
  So that I can schedule collection services

  Scenario: Citizen creates a booking successfully
    Given the citizen is on the homepage
    When the citizen navigates to the citizen page
    And the citizen selects municipality "Aveiro"
    And the citizen enters date "2025-11-20"
    And the citizen selects timeslot "Manhã"
    And the citizen enters description "Colchão e mobília velha"
    When the citizen submits the booking form
    Then the booking is created successfully
```

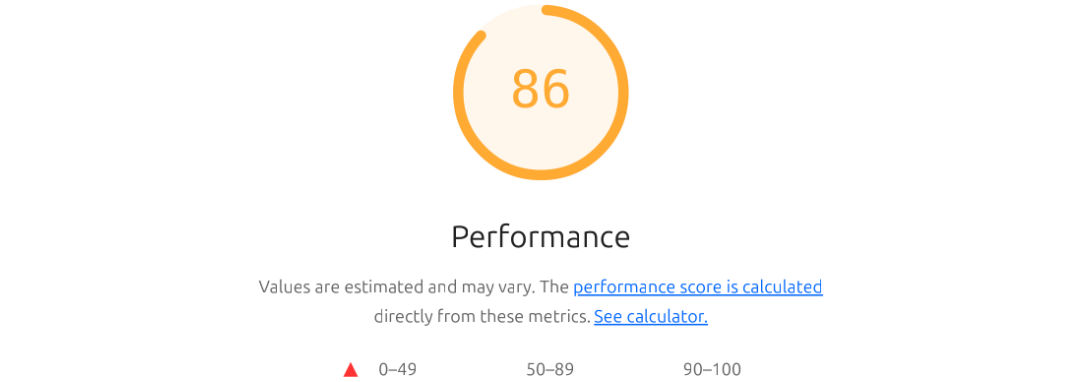
3.4 Non-functional testing

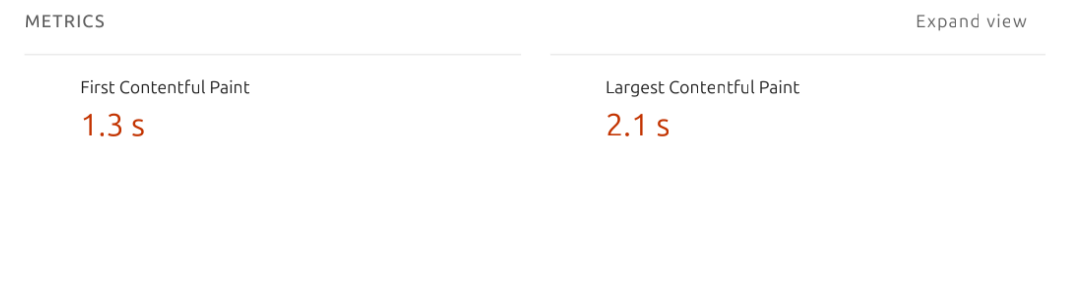
Para garantir a qualidade de experiência do utilizador e a conformidade com boas práticas web, foi utilizada a ferramenta Lighthouse para auditar a aplicação frontend (em http://localhost:5173/). O teste foi realizado usando emulação desktop e produziu métricas reais da aplicação numa sessão single-page.

Resultados do Lighthouse

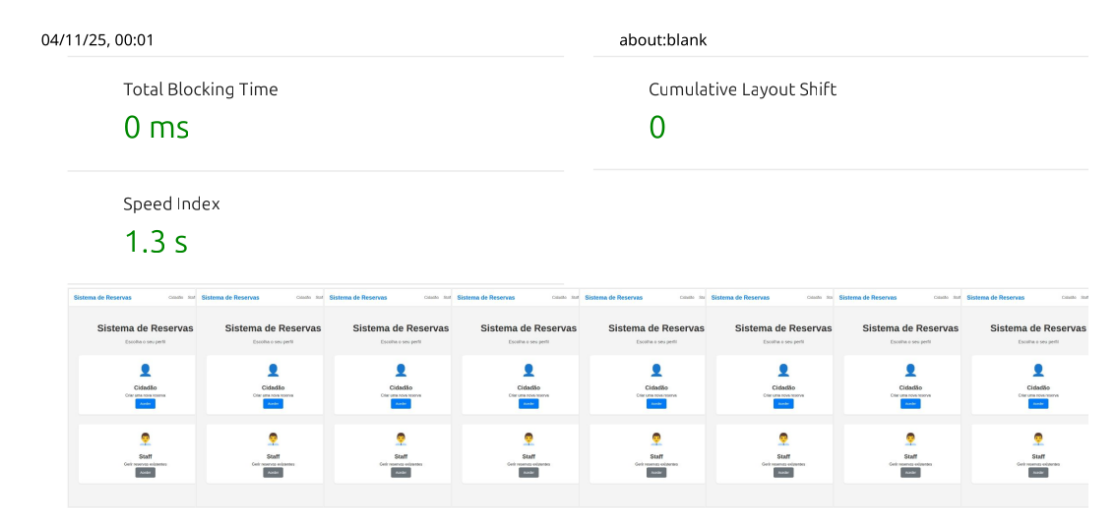
- Performance: 86/100
  - *First Contentful Paint:* 1.3 s
  - *Largest Contentful Paint:* 2.1 s
  - *Speed Index:* 1.3 s
  - *Total Blocking Time:* 0 ms
  - *Cumulative Layout Shift:* 0
- Acessibilidade: 89/100
  - Pequenas oportunidades de melhoria, como contraste insuficiente em algumas cores para legibilidade.
- Best Practices: 100/100
  - Cumprimento total das recomendações: políticas CSP/HSTS, isolamento de origem e padrões modernos.
- SEO: 82/100
  - Falta de meta description e erros no robots.txt, ambos facilmente ajustáveis para otimização do ranking.


04/11/25, 00:01
about:blank
http://localhost:5173/
86
Performance
89
Accessibility
100
Best Practices
82
SEO


86
Performance
Values are estimated and may vary. The performance score is calculated directly from these metrics. See calculator.
0–49
50–89
90–100


METRICS
Expand view
First Contentful Paint
1.3 s
Largest Contentful Paint
2.1 s

about:blank
1/5


04/11/25, 00:01
about:blank
Total Blocking Time
0 ms
Cumulative Layout Shift
0
Speed Index
1.3 s

3.5 Code quality analysis

A qualidade do backend foi analisada utilizando o SonarCloud, uma ferramenta de referência para análise estática de código em projetos Java. A integração foi configurada através de GitHub Actions, permitindo que cada atualização na branch principal fosse automaticamente auditada quanto a segurança, fiabilidade, manutenibilidade e cobertura de testes.

Resultados principais

- Quality Gate: Passed
- Security: 5 issues abertos, nível B
- Reliability: 0 issues, nível A
- Maintainability: 2 issues abertos, nível A
- Test Coverage: 84.4%
- Duplications: 0%
- Security Hotspots: 2

Interpretação dos Resultados

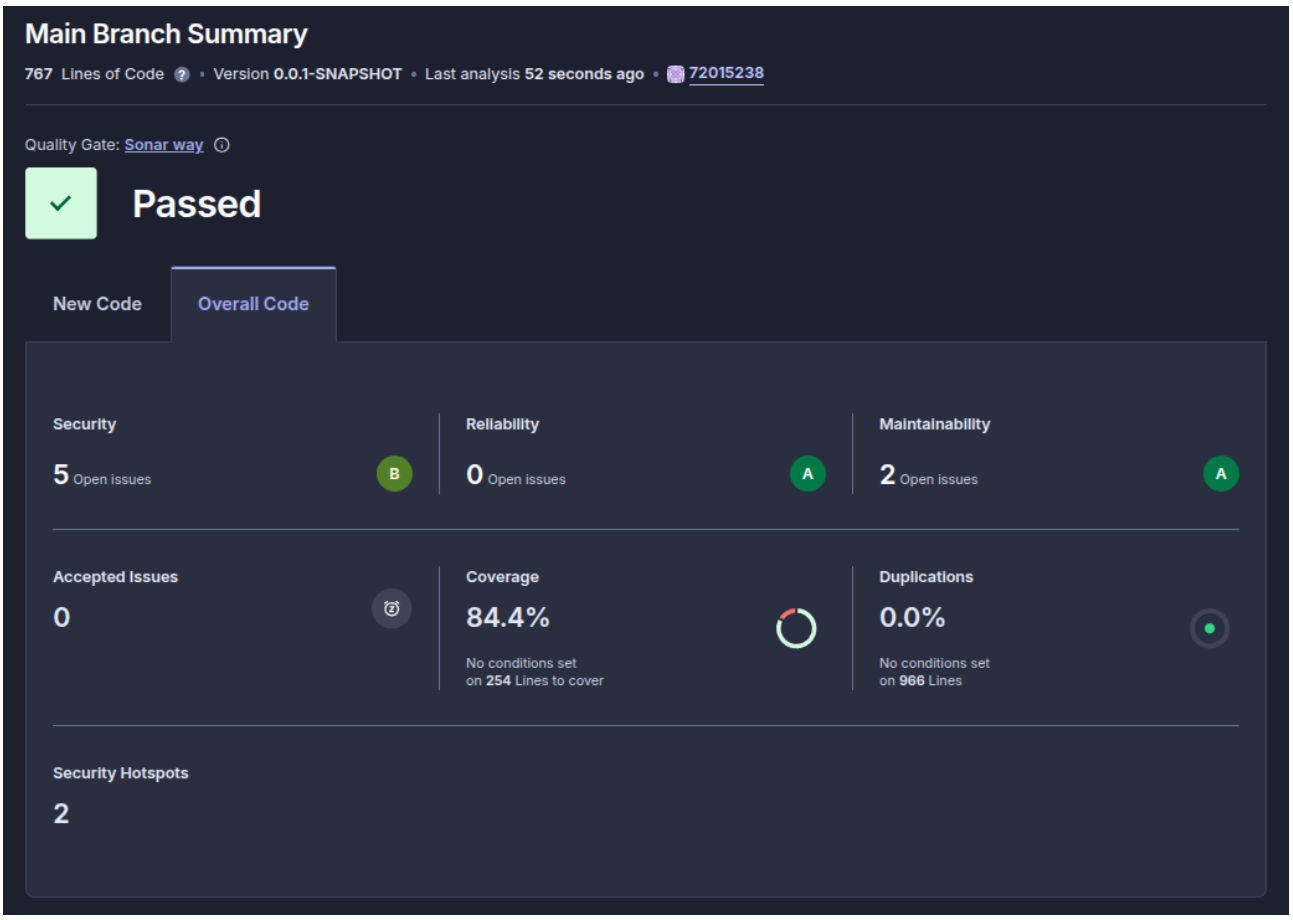
O projeto apresenta uma cobertura de testes elevada (84.4%), garantindo que a grande maioria da lógica de negócio está protegida contra regressões.
Não foram identificados problemas de fiabilidade, e o índice de manutenibilidade é favorável, refletindo um código organizado e sustentável.
Foram detetados:

- 5 alertas de segurança
- 2 security hotspots

Estes casos não representam vulnerabilidades críticas, mas indicam pontos onde podem ser reforçadas validações de entrada ou práticas de segurança nos endpoints — aspetos que muitas vezes não são imediatamente evidentes numa revisão manual de código.
A ausência total de código duplicado demonstra uma boa disciplina de reutilização e limpeza estrutural.

Lições Aprendidas

A análise automática permitiu identificar possíveis melhorias em pontos ligados à validação de dados e segurança de API, que não eram explícitos durante a revisão manual. Isto reforça o valor da ferramenta como suporte contínuo à qualidade, ajudando a antecipar riscos e manter padrões elevados ao longo da evolução do sistema.



4 References & resources

Project resources

Resource:	URL/location:
Video demo	https://github.com/rodrigosc8/projtqs/blob/main/docs/Demo.mp4
QA dashboard (online)	https://sonarcloud.io/project/overview?id=rodrigosc8_projtqs

Reference materials

- Documentação Spring Boot: https://spring.io/guides/gs/spring-boot/
- Baeldung – Spring/Java: https://www.baeldung.com/
- Documentação oficial React: https://react.dev/
- Springdoc OpenAPI: https://springdoc.org/
- Documentação Cucumber: https://cucumber.io/docs/
- SonarCloud docs: https://docs.sonarcloud.io/