universidade de aveiro
deti departamento de eletrónica, telecomunicações e informática

Teste e Qualidade de Software

HW1: Mid-term assignment report

Diogo Henrique Costa Duarte

2025-11-03

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introducão

1.1 Visão Geral do projeto

ZeroMonos é uma aplicação web full-stack desenvolvida para digitalizar e otimizar o processo de agendamento de recolha de resíduos volumosos (monos) em municípios portugueses. Permite que cidadãos criem, consultem e cancelem reservas de forma autônoma, enquanto funcionários municipais gerenciam essas solicitações através de um painel administrativo.

O projeto visa substituir processos manuais e presenciais por uma solução digital eficiente, melhorando a satisfação do cidadão e a produtividade municipal.

1.1.1 Funcionalidades principais

- Criação de marcações com token UUID;
- Consulta de marcações por token;
- Gestão administrativa (visualizar, filtrar, alterar status, cancelar);
- Regras de negócio: validação de datas (sem hoje/passado/domingos) e limite de 100 marcações por município.

1.2 Implementação Atual (falhas & extras)

A implementação cobre o fluxo principal e inclui extras focados em qualidade. A seguir um resumo dos pontos mais relevantes.

Tabela 1.1: Resumo de funcionalidades e estado

Item	Estado
Adicionar mais entidades a base de dados	Não implementado
Implementar uma "operations dashboard"para o staff	Não implementado
Quality Gate em CI	Extra

1.3 Uso de IA generativa

Ferramentas como GitHub Copilot e ChatGPT foram utilizadas para maior eficiencia de geração de código. Todo o código gerado por IA foi revisto manualmente

Capítulo 2

Especificação do Produto

2.1 Escopo Funcional e interações suportadas

2.1.1 Atores

- **Cidadão**: criar, consultar e cancelar marcações via token;
- **Staff Municipal**: visualizar e gerir marcações (atualizar status, cancelar);
- **Sistema / Serviços**: importação de municípios, validações e geração de tokens.

2.1.2 Casos de uso principais

UC1 — Criar Marcação.

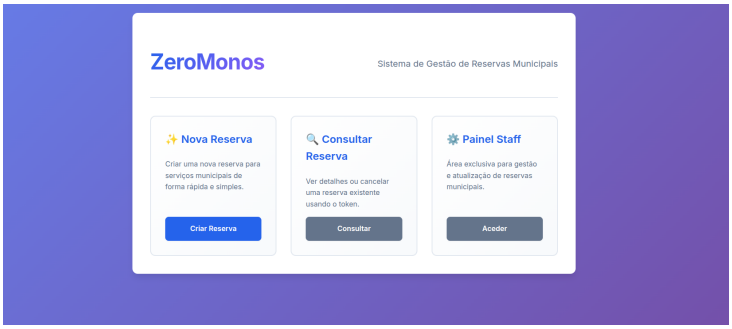
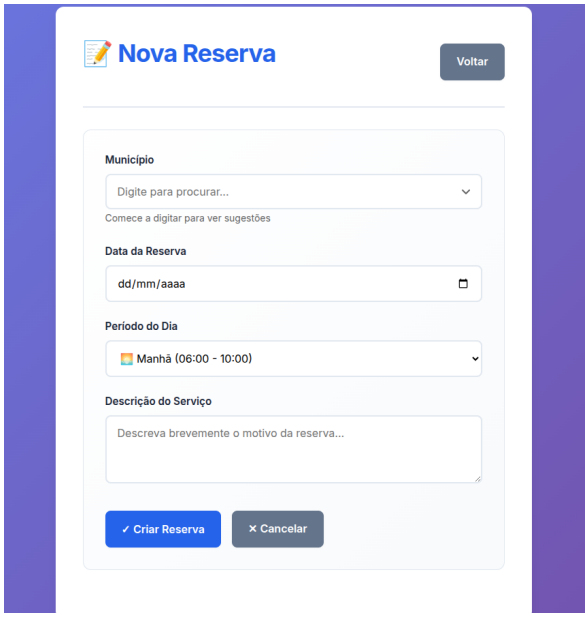


Figura 2.1: Página inicial (ponto de entrada do utilizador) — passo inicial antes de criar uma marcação.

CIDADÃO: a partir da página inicial (figura 2.1) acede ao formulário de criação de marcação. No formulário o utilizador selecciona município, data e horário, descreve o serviço pretendido e submete o pedido. As principais validações aplicadas no momento da criação são: (i) o município existe; (ii) a data não pode ser no passado, no dia actual ou num domingo; (iii) não exceder o limite de 100 marcações por município. Após submissão bem sucedida, o sistema gera um token UUID que identifica a marcação e o apresenta ao utilizador.



(a) Formulário de criação de marcação (booking).

(b) Token UUID apresentado ao utilizador após criação da marcação.

Figura 2.2: Fluxo de criação: (esq.) formulário de booking; (dir.) token gerado após submissão.

UC2 — Consultar Marcação.

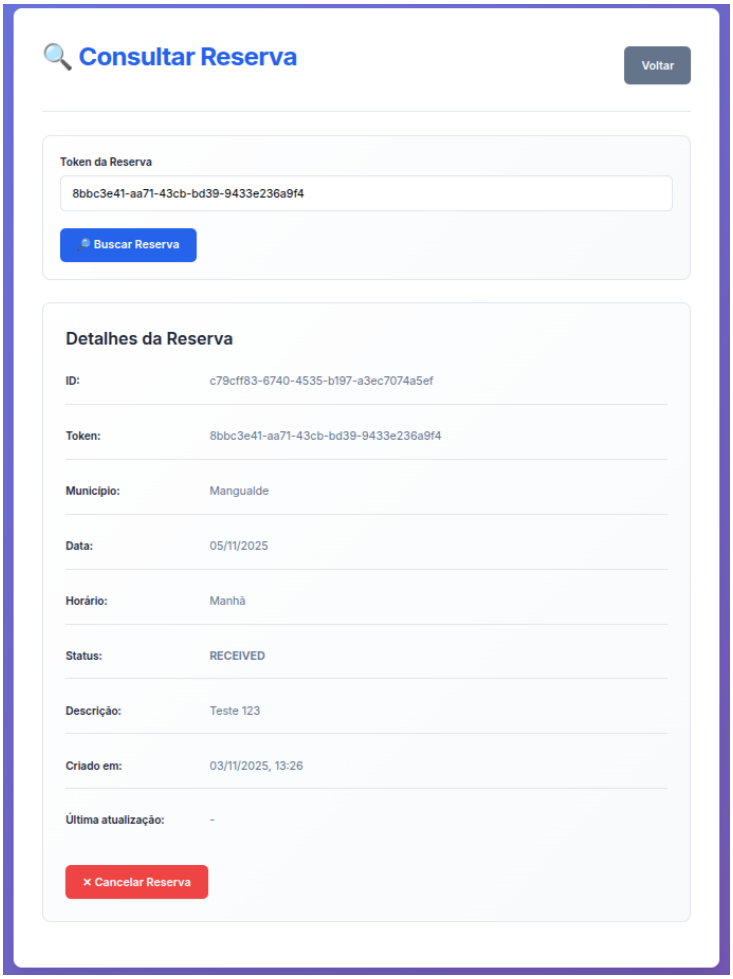


Figura 2.3: Página de consulta de marcação por token — o cidadão insere o token para visualizar os detalhes (município, data, horário, estado, descrição e atualizações).

Descrição: o cidadão acede à página de consulta (figura 2.3), insere o token recebido e visualiza os detalhes da marcação. Se o estado da marcação permitir, é disponibilizada a opção de cancelamento.

UC3 — Cancelar Marcação.

Figura 2.4: Mensagem de cancelamento (cidadão) — o utilizador fornece o token e confirma o cancelamento quando aplicável.

Descrição: cidadão ou staff pode iniciar o cancelamento. O sistema valida se o estado actual permite o cancelamento (apenas `RECEIVED`). Em caso afirmativo, o estado é actualizado para `CANCELLED`.

UC4 — Gestão por Staff.

(a) Painel de staff: listagem e filtros de marcações.

Descrição: staff municipal acede ao painel de gestão (figura 2.5a) para filtrar marcações por município, pesquisar por token e realizar operações administrativas (atribuir, completar ou cancelar marcações). Abaixo exemplifica-se a acção de actualizar o estado de uma marcação.

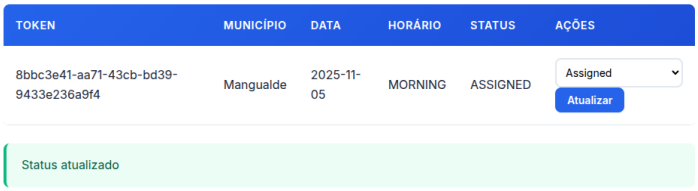


Figura 2.6: Exemplo de atualização de estado de uma marcação pelo staff (alteração de `RECEIVED` para `ASSIGNED`, etc.).

2.2 Arquitetura implementado no sistema

Arquitectura: monolito em camadas. A aplicação usa Spring Boot, Spring Data JPA, suporta H2 para testes. Um serviço de importação de municípios é executado ao arranque com fallback para lista interna.

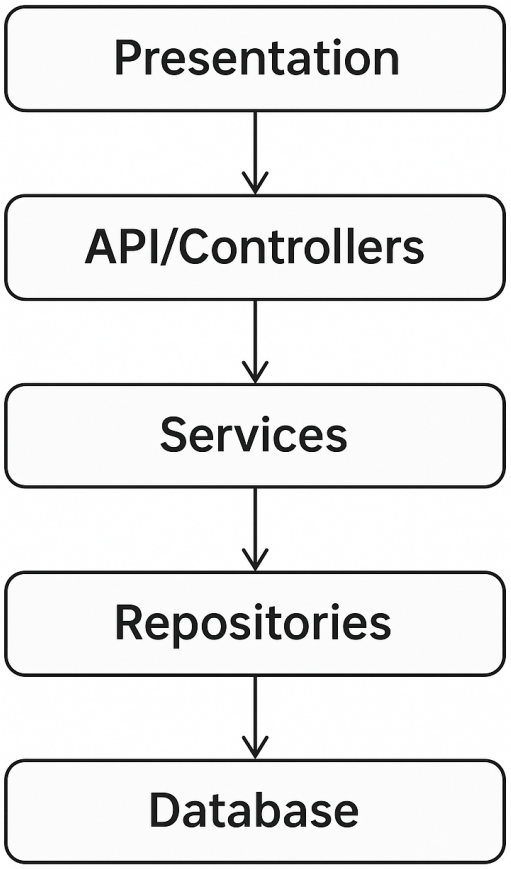


Figura 2.7: Arquitetura em camadas

2.3 API para developers

Principais endpoints do sistema, refletindo os requisitos funcionais do projeto:

- `POST /api/bookings` — cria nova marcação (body: `BookingRequest`); gera e retorna `token` UUID.
- `GET /api/bookings/municipalities` — devolve lista fechada de municípios válidos.
- `GET /api/bookings/{token}` — consulta uma marcação existente através do token.
- `PUT /api/bookings/{token}/cancel` — permite ao cidadão cancelar a marcação.
- `GET /api/staff/bookings` — lista todas as marcações (acesso restrito ao staff).
- `PATCH /api/staff/bookings/{token}/status` — staff pode atualizar o estado de uma marcação (e.g. `RECEIVED` → `ASSIGNED`).

Os principais **DTOs** usados são:

Tabela 2.1: Principais estruturas de dados trocadas via API

BookingRequest	`municipalityName, rquestDate, timeSlot, description`
BookingResponse	`token, municipalityName, requestDate, description, status, statusHistory...`

O histórico de estados (`statusHistory`) regista cada alteração de estado com `timestamp`, permitindo rastrear a evolução completa da marcação.

Capítulo 3

Garantia de Qualidade

3.1 Estratégia geral de testes

O projeto **Zeromonos** adotou uma estratégia de testes multinível e híbrida. A abordagem combinou desenvolvimento orientado a comportamento (BDD) usando `Cucumber` para testes funcionais end-to-end, com testes unitários tradicionais em `JUnit` e `Mockito`. Para testes de integração, utilizou-se `H2` como banco de dados em memória, garantindo velocidade e isolamento sem dependências externas. O `REST-Assured` foi usado em testes específicos de API (`RestAssureTest`), enquanto o `Selenium WebDriver` foi integrado com `Cucumber` para automação da interface web. Nos testes não funcionais, `K6` foi utilizado para *performance testing* (load, stress e spike tests) e `Lighthouse` para análise de performance frontend. Esta abordagem multicamada assegurou cobertura desde unidades individuais até fluxos end-to-end completos com interface gráfica.

3.2 Testes Unitários e de Integração

Testes Unitários foram implementados em `BookingServiceImplUnitTest`, focando na lógica de negócio isolada do `BookingService`. Utilizou-se `Mockito` para simular dependências de repositórios e outros serviços, permitindo testes rápidos e determinísticos sem I/O externo.

Testes de Serviço foram organizados no pacote `isolationtests` usando `H2 database` em memória:

- `BookingControllerTest`: testa a camada de controladores REST com Spring `MockMvc`;
- `BookingRepositoryTest`: valida operações de persistência com H2, testando queries e relacionamentos JPA;
- `BookingServiceTest`: testa integração entre serviço e repositório;

- `MunicipalityImportServiceTest`: valida importação de dados de municípios;
- `StaffControllerTest`: testa endpoints de gestão de staff.

O teste de integração principal é o `RestAssureTest`, que valida a aplicação completa (*end-to-end*) usando `REST-Assured`. Ele executa num servidor Spring Boot real, iniciado em uma porta, testando todos os endpoints REST — criação, busca, cancelamento e atualização de reservas, listagem de municípios e operações de staff. A persistência é feita com `H2` em memória, configurada com `@TestPropertySource` para garantir isolamento total entre execuções.

A estratégia envolveu `H2` configurado automaticamente pelo Spring Boot Test para criar schemas efêmeros a cada execução, garantindo isolamento total e velocidade. Não foram usados containers Docker, mantendo os testes leves e rápidos.

3.3 Testes de Aceitação

Os testes de aceitação foram implementados usando `Cucumber BDD` com `Selenium WebDriver`, organizados em três *features* principais:

- **1-booking.feature** — cenários de criação de reservas através da interface web;
- **2-searchReserve.feature** — busca e consulta de reservas pela interface;
- **3-changeStateStaff.feature** — alteração de status de reservas pela equipa de staff.

Os *step definitions* (`BookingSteps`, `BookingViewSteps`, `StaffBookingSteps`) implementam os cenários usando Selenium para interação real com elementos HTML (formulários, botões, inputs). `BookingViewSteps` demonstra o uso de Selenium WebDriver com esperas explícitas, manipulação de tokens compartilhados entre cenários e validação de conteúdo dinâmico. `TestContext` mantém estado compartilhado (incluindo WebDriver) entre steps, enquanto `CommonSteps` fornece steps reutilizáveis. Os testes são executados via `RunCucumberTest` com integração `JUnit`, validando fluxos completos da interface web até persistência em H2. Esta abordagem permitiu especificações em linguagem natural (Gherkin) compreensíveis por stakeholders não técnicos, com automação completa das interações do utilizador.

3.4 Testes não funcionais

Performance Testing com K6: três tipos de testes foram implementados:

- `performance.js`: *load testing* com carga sustentada para avaliar comportamento sob uso normal;
- `stress-test.js`: testes de stress para identificar limites do sistema e pontos de quebra;
- `spike-test.js`: simulação de picos súbitos de tráfego.

Os resultados mostraram excelente estabilidade e desempenho em todos os cenários. Durante o teste de carga normal, a latência média foi de apenas **6.8 ms** com **p95 = 13 ms**, sem erros (**0% de falhas**) e throughput de cerca de **25 requisições/s**. Nos testes de *spike*, mesmo com 200 utilizadores simultâneos, o sistema manteve p95 abaixo de **30 ms** e nenhuma requisição falhou. No *stress test* com até 1 000 utilizadores, o tempo p95 manteve-se em torno de **1.6 s**, confirmando resiliência e escalabilidade do backend sem degradação crítica.

Frontend Performance com Lighthouse: o script `lighthouse-test.js` automatizou a análise de todas as páginas principais do sistema (`index`, `booking-form`, `booking-view`, `staff-bookings`). Os resultados ficaram consistentemente acima dos thresholds definidos (performance >=70, acessibilidade >=80). Todas as páginas atingiram **performance entre 84 e 92**, **acessibilidade 100/100**, e **boas práticas 96/100**, com **SEO 91/100**. Isto demonstra excelente otimização e acessibilidade, com apenas pequenas oportunidades de melhoria em otimização de imagens e scripts estáticos.

De forma geral, os testes não funcionais comprovam que o sistema apresenta **alto desempenho, estabilidade e experiência de utilizador consistente**, cumprindo plenamente os requisitos de qualidade estabelecidos.

3.5 Análise da qualidade de código

A análise de qualidade de código foi conduzida através de ferramentas integradas ao processo de desenvolvimento (SonarQube e verificações Maven). O foco esteve na identificação de *code smells*, vulnerabilidades de segurança e melhorias de manutenibilidade. De forma geral, o código apresentou boa estrutura de pacotes (`boundary/data/services/dto`), separação clara de responsabilidades e alta legibilidade. Ainda assim, foram identificados pontos de melhoria pontuais, posteriormente corrigidos, que resultaram em melhor qualidade técnica e conformidade com as boas práticas Java.

Principais métricas e correções (SonarQube)

1. **Uso de `Stream.toList()` vs `Collectors.toList()`**
   Problema: Utilização de APIs antigas e verbosas para conversão de streams em listas.
   Correção: Atualização para `Stream.toList()` (Java 16+).
   Melhoria: Código mais conciso, legível e geração de listas imutáveis por padrão.

2. **Log de dados controlados pelo utilizador**
   Problema: Potencial risco de *Log Injection* e exposição de dados sensíveis nos logs.
   Correção: Sanitização e remoção de valores controlados pelo utilizador antes do registo em log.
   Melhoria: Aumento da segurança e redução de ruído em logs de produção.

3. **Tratamento desnecessário de exceções**
   Problema: Blocos `try-catch` que apenas registavam o erro e relançavam a mesma exceção, sem contexto adicional.
   Correção: Simplificação ou remoção de blocos redundantes, adicionando contexto informativo quando necessário.
   Melhoria: Código mais limpo e mensagens de erro mais úteis.

4. **Melhoria do coverage**
   Melhoria: Implementação de testes adicionais para validações e serviços que não estavam previamente cobertos.

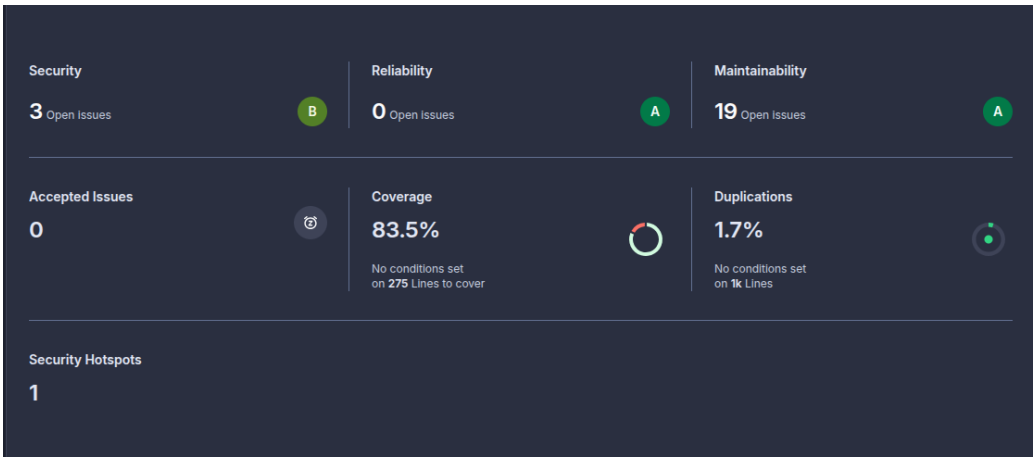


Figura 3.1: Relatório SonarQube antes de algumas correções.

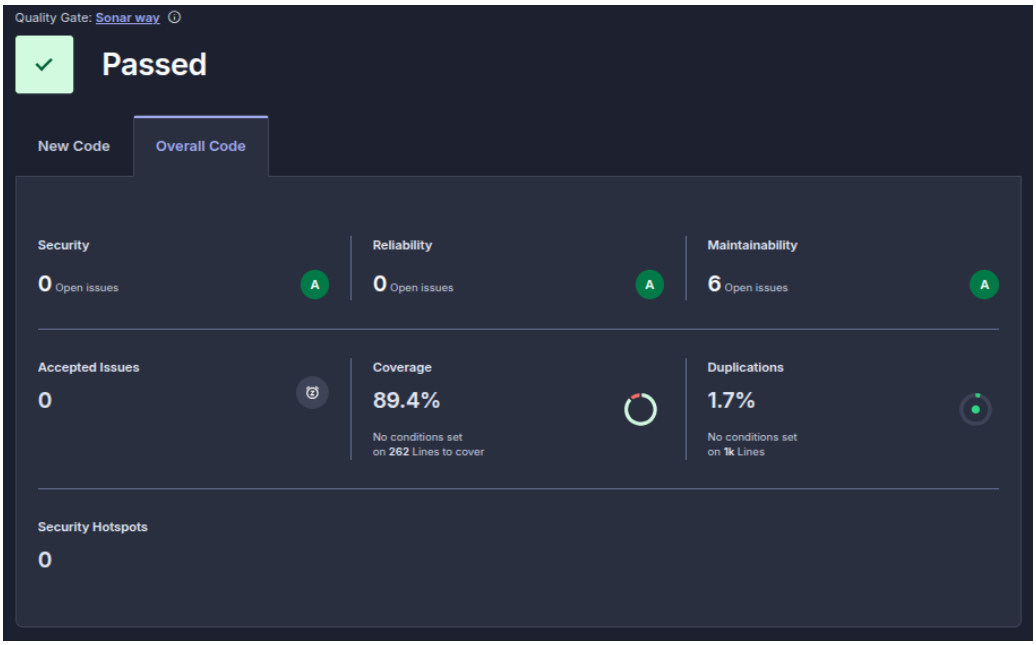


Figura 3.2: Relatório SonarQube após correções — melhorias aplicadas.

Figura: Comparação das métricas de qualidade de código antes e depois das melhorias implementadas.

Com estas melhorias, o código do **Zeromonos** tornou-se mais seguro, eficiente e sustentável, refletindo boas práticas de engenharia de software e reforçando a confiabilidade do sistema.

3.6 Integração Contínua (CI)

O projeto implementa uma **pipeline CI/CD** automatizada que é executada em cada `push` ou `pull request`. A pipeline está configurada para:

1. **Build**: compilação Maven com dependências;
2. **Unit Tests**: execução rápida com JUnit/Mockito;
3. **Service Tests**: testes com H2 em memória (sem Docker);
4. **Integration Tests**: validação de endpoints com REST-Assured;
5. **Functional Tests**: execução completa de features Cucumber com Selenium (modo *headless* em CI);
6. **Code Quality**: análise estática e relatórios de cobertura (SonarQube);
7. **Performance Tests**: execução de K6 tests;
8. **Frontend Tests**: auditoria Lighthouse para métricas de UX.

A pipeline executa *jobs* em paralelo para otimizar tempo. Os testes unitários executam primeiro (*fast feedback*), seguidos de testes funcionais e não funcionais. O uso do banco H2 elimina dependências externas e acelera a execução. Relatórios de cobertura e qualidade são gerados automaticamente, e o *build* falha caso testes críticos falhem, prevenindo regressões. Esta configuração garante qualidade contínua sem overhead de containers ou bases de dados externas.

Capítulo 4

Referências & recursos

4.1 Recursos do Projeto

Tabela 4.1: Tabela Recursos

Resource	URL / Location
Video demo	*Dentro da pasta docs no repositório*
QA dashboard (online)	https://sonarcloud.io/project/overview?id=diogoh-11_ZeroMonos
CI/CD pipeline	GitHub Actions pipeline in ZeroMonos repository

4.2 Referências

- **Spring Boot Documentation** — https://docs.spring.io/spring-boot/docs/current/reference/html
- **JUnit 5 User Guide** — https://junit.org/junit5/docs/current/user-guide/
- **Mockito Documentation** — https://site.mockito.org/
- **Cucumber BDD Documentation** — https://cucumber.io/docs/guides/
- **Selenium WebDriver Documentation** — https://www.selenium.dev/documentation/
- **SonarCloud Documentation** — https://docs.sonarcloud.io/
- **GitHub Actions Documentation** — https://docs.github.com/en/actions
- **Zeromonos GitHub Repository** — https://github.com/diogoh-11/ZeroMonos
- **Zeromonos SonarCloud Dashboard** — https://sonarcloud.io/project/overview?id=diogoh-11_ZeroMonos