HW1: Mid-term assignment report

Abel José Enes Teixeira [113655], v2025-03-26

Introdução

Visão geral

Este relatório apresenta o Mid-term Assignement necessário para o TQS, abrangendo tanto os recursos do produto de software quanto a estratégia de garantia de qualidade adotada.
A aplicação desenvlovida chama-se Moliceiro University e baseia-se numa plataforma web de reservas de refeições em várias cantinas (restaurantes).
A aplicação também é capaz de fornecer previsões do tempo, até 7 dias, juntamente com os detalhes de cada refeição para ajudar os utilizadores na escolha do restaurante.

Limitações atuais

A aplicação apresenta as seguintes limitações:

a) Não existe autenticação
b) Não há controle do número de reservas por restaurante
c) O utilizador só pode reservar uma refeição por dia Almoço ou Jantar.

Especificação do produto

Escopo funcional e interações suportadas

A aplicação Moliceiro University é destinada a qualquer pessoa que frequente a Universidade de Aveiro, que tencionem reservar refeições com antecedência nos restaurantes/cantinas disponíveis.

Os principais atores são:

d) Utilizador Comum (Aluno, Funcionário, etc.):

e) Ver Restaurantes disponíveis;

f) Por cada restaurante ver a lista de refeições, com Almoços, Jantares e previsão do tempo;

g) Fazer a reserva de refeições, escolhendo o dia e tipo de refeição;

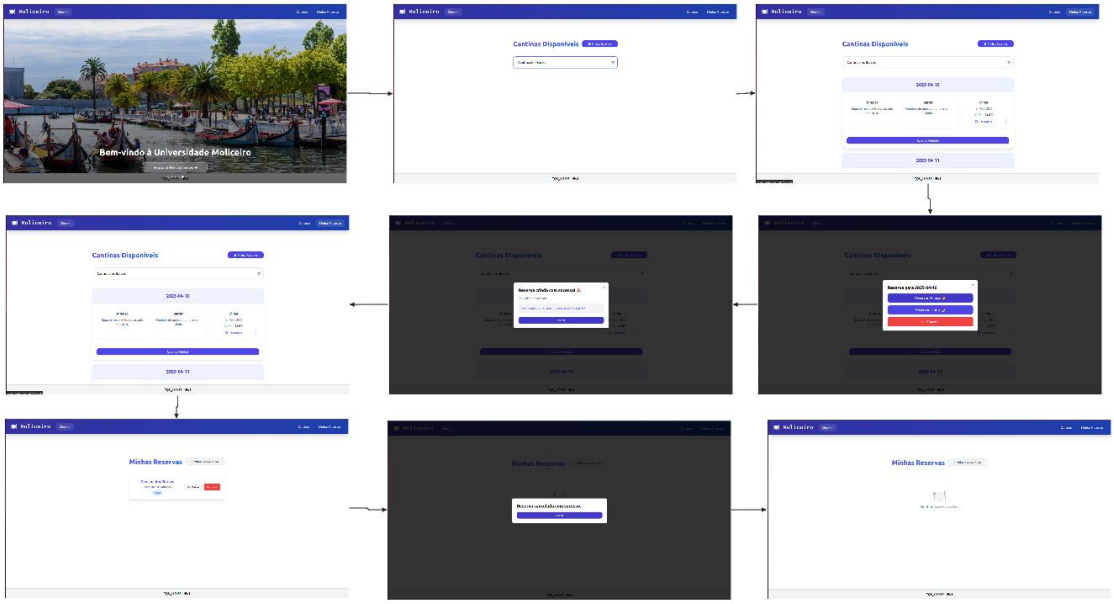
h) Consultar as reservas realizadas;

i) Poder cancelar as reservas ativas e que não foram usadas;

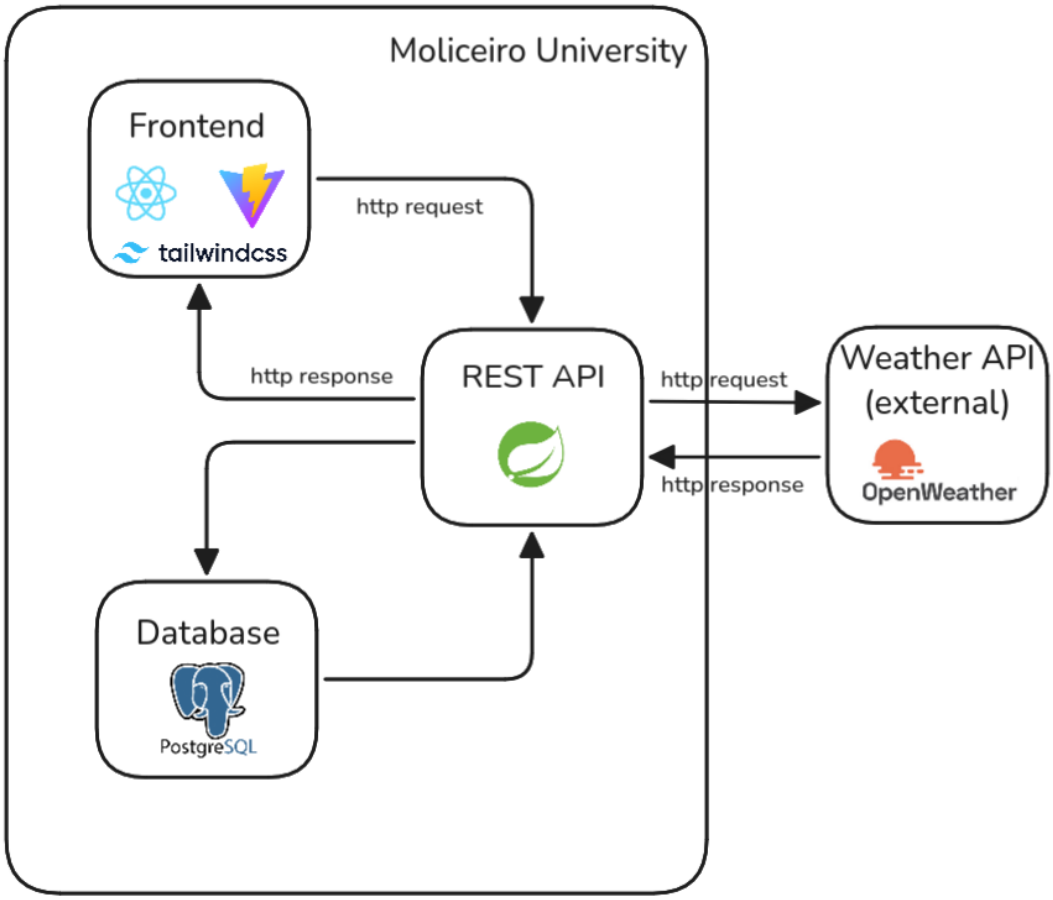
j) Admin (Funcionário do Restaurante/Cantina):

k) Utilizar o token da reserva para poder dar checkin ao utilizador comum;

Vamos seguir o fluxo do utilizador comum:



Arquitetura da implementação do sistema



Como representado no diagrama acima, a arquitetura está dividida em 3 camadas principais:

l) Frontend: O frontend foi desenvolvido recorrendo ao uso de React, Vite, para estrutura, Tailwind CSS para os estilos e a framework Shadcn UI para componentes.
O frontend comunica com a API através de pedidos HTTP.

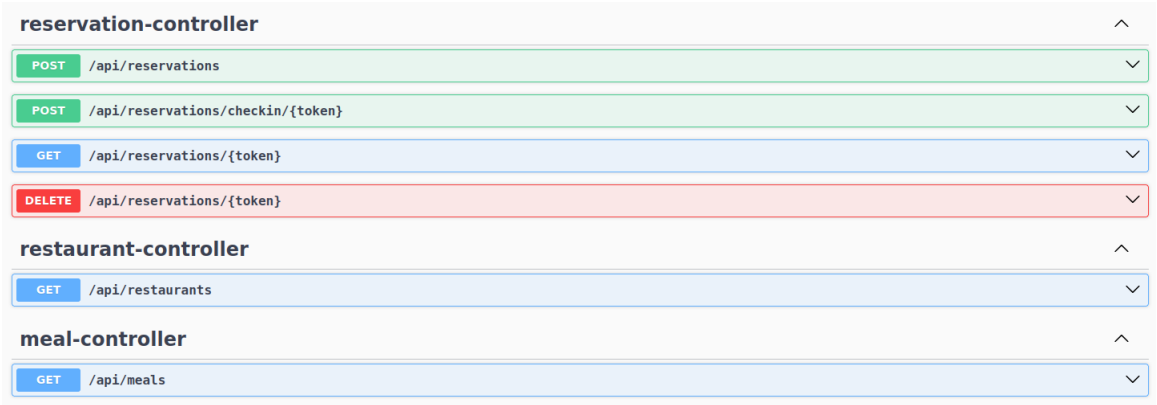
m) Backend: O backend é uma REST API criada com Spring Boot. O backend comunica com a API externa através de pedidos HTTP.

n) Base de dados: Foi utilizado PostgreSQL para armazenamento dos dados dos restaurantes, refeições e reservas.

o) API Externa: O backend faz requisições a uma API externa, a Open-Meteo, para obter as previsões do estado do tempo para associar às refeições.

API para desenvolvedores

Esta API foi implementada para permitir a comunicação na aplicação de forma fácil e eficaz. Abaixo estão listados os endpoints implementados:



Lista de Endpoints da API:

Reservation Controller:

p) POST /api/reservations
   - Cria uma nova reserva num determinado restaurante.

q) POST /api/reservations/checkin/{token}
   - Altera o estado da reserva quando passado um token de reserva.

r) GET /api/reservations/{token}
   - Retorna os detalhes de uma reserva quando passado um token.

s) DELETE /api/reservations/{token}
   - Cancela a reserva quando passado um token.

Restaurant Controller:

t) GET /api/restaurants
   - Retorna uma lista com todos os restaurantes.

Meal Controller

u) GET /api/meals
   - Retorna todas as refeições de um restauante.

Garantia de Qualidade

Estratégia geral para testes

A estratégia adotada para o projeto foi orientada a testes unitários e testes de integração, que cobrem a maior parte do código desenvolvido no backend.

No início, os testes foram desenvolvidos juntamente com os serviços, para garantir que a aplicação estava a funcionar de forma correta, com a utilização do Junit5 e do Mockito.

Para testar a ligação entre as diferentes camadas do backend (controllers, serviços e repositories) foram criados testes de integração recorrendo ao Spring Boot Test e ao MockMvc.

Por fim foram realizados os testes funcionais de frontend com o Selenium WebDriver e os testes de desempenho do backend com o Grafana k6, que permitiu avaliar a performance da aplicação quando submetida a uma intensa carga de utilização.

Testes Realizados:

v) Testes Unitários:

w) MealWithWeatherDTOTest.java: Testa a criação da MealWithWeatherDTO.

x) ReservationRequestDTO.java: Testa a criação da ReservationRequestDTO.

y) ReservationResponseDTO.java: Testa a criação da ReservationResponseDTO

z) ReservationServiceTest.java: Testa a criação, consulta, checkin e cancelamento de reservas.

aa) WeatherServiceTest.java: Testa se o serviço obtém a previsão do tempo corretamente para uma determinada data.

bb) Testes de Integração:

cc) ReservationControllerTest.java: Testa a criação, cancelamento e checkin de reservas.

dd) RestaurantControllerTest.java: Testa o retorno das informações dos restaurantes.

ee) Testes Funcionais:

ff) BDDHw1Test: Testa o frontend.

gg) Testes de Desempenho:

hh) PerformanceGetTest.js: Avalia a capacidade e tempo de resposta da API quando submetida a vários pedidos GET.

ii) PerformancePostTest.js: Avalia a capacidade da API concluir várias reservas com vários Virtual Users em simultâneo.

Testes unitários e de integração

A estratégia adotada foi separar os Testes de Integração dos Testes Unitários, permitindo assim o correto funcionamento da lógica interna da aplicação e da aplicação como um todo. Para realizar os testes unitários, foram utilizados mocks (Mockito) para isolar as dependências e os repositórios.
Os testes de integração, foram realizados recorrendo ao Spring Boot Test e ao MockMvc, que garantem o bom funcionamento entre controllers, serviços e base de dados.

Testes Unitários

Onde foram utilizados:

jj) Serviços: *Business Logic* (ReservationService, WeatherService).

kk) DTOs: Validação do mapeamento de dados (MealWithWeatherDTO).

Exemplo de Implementação (ReservationServiceTest.java):

```java
@Test
void testCreateReservation_Success() {
    ReservationRequestDTO request = new ReservationRequestDTO();
    request.setRestaurantId(1L);
    request.setDate(LocalDate.now());
    request.setType("ALMOCO");

    when(restaurantRepository.findById(1L)).thenReturn(Optional.of(mockRestaurant));

    ReservationResponseDTO response = reservationService.createReservation(request);

    assertNotNull(response.getToken());
    assertEquals( expected: "Cantina do Barcos", response.getRestaurantName());
    assertEquals( expected: "ALMOCO", response.getType());
    assertFalse(response.isCheckedIn());

    verify(restaurantRepository).findById(1L);
    verify(reservationRepository).save(any(Reservation.class));
}
```

Testes de Integração

Onde foram utilizados:
II) Controllers: Endpoints da API (ReservationController e MealController).

Exemplo de Implementação (trecho de ReservationControllerTest.java):

```java
@Test
void testCheckInReservation_Success() throws Exception {
    ReservationRequestDTO request = new ReservationRequestDTO();
    request.setRestaurantId(1L);
    request.setDate(LocalDate.now().plusDays( daysToAdd: 3));
    request.setType(MealType.ALMOCO.name());

    String response = mockMvc.perform(post( uriTemplate: "/api/reservations")
                    .contentType(MediaType.APPLICATION_JSON)
                    .content(objectMapper.writeValueAsString(request)))
            .andExpect(status().isOk())
            .andReturn().getResponse().getContentAsString();

    String token = objectMapper.readTree(response).get("token").asText();

    mockMvc.perform(post( uriTemplate: "/api/reservations/checkin/" + token))
            .andExpect(status().isOk())
            .andExpect(content().string(containsString( substring: "Check-in realizado com sucesso")));
}
```

Testes funcionais

Os testes funcionais foram desenvolvidos utilizando o **Selenium WebDriver**, simulando de forma realista a interação de um utilizador com a interface gráfica da aplicação. O objetivo principal foi validar o fluxo do utilizador, garantindo que todas as funcionalidades estão a funcionar conforme esperado.

Casos de Teste Considerados:

Reserva de Refeição:

mm) Seleção de cantina, data e tipo de refeição.

nn) Geração de token de confirmação.

Cancelamento de Reserva:

oo) Acesso à lista de reservas ativas.

pp) Validação de cancelamento bem-sucedido.

Check-in pelo Admin:

qq) Validação de token na interface do admin.

Exemplo de utilização no código:

```
@Test
public void bDDHw1Test() {
  // Test name: BDDHw1Test
  // Step # | name | target | value
  // 1 | open | / |
  driver.get("http://localhost:3000/");
  // 2 | setWindowSize | 1850x1053 |
  driver.manage().window().setSize(new Dimension( width: 1850,  height: 1053));
  // 3 | click | css=.inline-flex |
  driver.findElement(By.cssSelector(".inline-flex")).click();
  // 4 | click | css=.p-2 |
  driver.findElement(By.cssSelector(".p-2")).click();
  // 5 | select | css=.p-2 | label=Cantina dos Barcos
  {
    WebElement dropdown = driver.findElement(By.cssSelector(".p-2"));
    dropdown.findElement(By.xpath( xpathExpression: "//option[. = 'Cantina dos Barcos']")).click();
  }
```

Testes não funcionais

Os testes não funcionais baseiam-se em avaliar a **capacidade de resposta**, **estabilidade** e **escalabilidade** do sistema sob condições de carga intensiva. Para isso, utilizou-se o **Grafana k6**, que permitiu simular cenários realistas de tráfego intenso e mostrar indicadores críticos de desempenho. O principal objetivo destes testes é verificar a robustez da API.

Testes de Carga com k6

rr) **Configuração**:

ss) **GET /meals**: 100 VUs por 10 segundos.

tt) **POST /reservations**: 20 VUs por 10 segundos.

uu) **Resultados**:

Métrica	GET /meals	POST /reservations
Pedidos	5883	3278
Taxa de sucesso	99.94%	89.27%
Latência	44.07ms	9.53ms

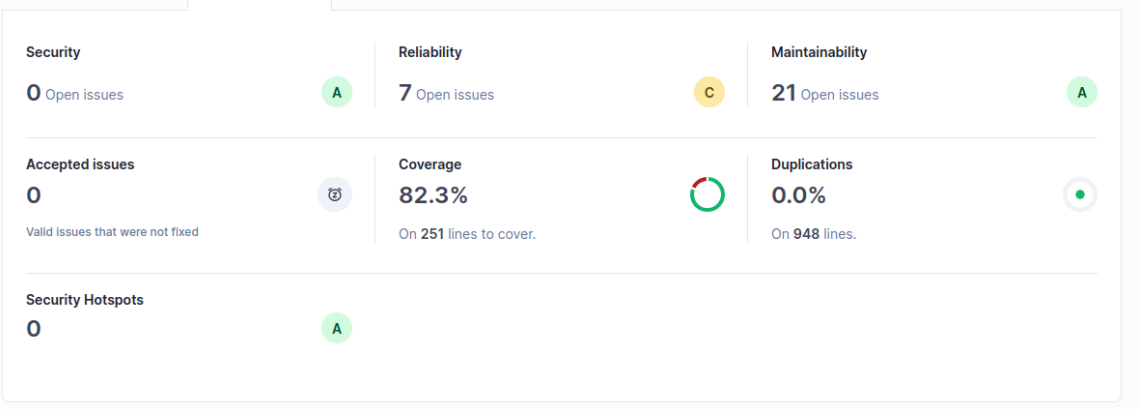
vv) **Análise**:

ww) O sistema mostrou alta escalabilidade para operações de leitura (GET).

xx) Operações de escrita (POST) geram confilitos (409) devido à limitação de apenas poder haver uma reserva por dia.

Análise de qualidade do código

O código foi analizado com suporte ao SonarQube, que separa em categorias a cobertura de testes, confiabilidade, manutenibilidade e segurança. A cobertura de testes ficou a 82.3%, o que significa que temos uma boa cobertura de testes na aplicação.



References & resources

Project resources

Resource:	URL/location:
Git repository	https://github.com/ttabelhaxd/TQS_113655/tree/main/hw1
Video demo	https://github.com/ttabelhaxd/TQS_113655/tree/main/hw1/demo
QA dashboard (online)	---
CI/CD pipeline	---
Deployment ready to use	---

Reference materials

https://site.mockito.org/
https://junit.org/junit5/
https://www.selenium.dev/