HW1: Mid-term assignment report

Carolina Sofia Pereira da Silva [113475], v2025-04-08

1 Introduction

1.1 Overview of the work

This report presents the midterm individual project required for TQS, covering both the software product features and the adopted quality assurance strategy.

This report also outlines the development process and key features of my meals booking based on weather forecasts application. The primary objective was to create a robust services API facilitating the reservation of meals tickets, focusing on two core functionalities: view and reserve meals planned for a selected restaurant unit for the upcoming days and get a code after the reservation is done and also show the weather forecast for each day. Besides that functionalities, I also implemented functionalities like check details for an active reservation (and optionally cancel it) and allow food services workers to verify a reservation (and mark the code as used).

To showcase the application's capabilities, it was also developed a simplified web app demonstrating these essential use cases. The product’s name is Moliceiro University Restaurants.

To ensure the reliability and quality of the application, I incorporated various types of tests:

A) Unit tests: Implemented for critical components such as booking logic, cache behavior, and utility functions like validators and converters.

B) Service level tests: Conducted with dependency isolation using mocks, particularly testing scenarios detached from external data providers.

C) Integration tests: Utilized Spring Boot MockMvc and REST-Assured with Flyway Migration to verify the functionality of the API endpoints.

D) Functional testing: Employed BDD with Selenium WebDriver to validate the web interface's behavior.

Furthermore, I integrated code quality metrics into the development process by utilizing SonarQube for analysis. This integration ensures adherence to coding standards and identifies areas for improvement. Integration with SonarQube (or Codacy) was established, enabling continuous analysis of the project's codebase. For public Git projects, SonarCloud was utilized for analysis.

1.2 Current limitations

There exists some problems with the functional testing in the front-end, like for them to work I need to frist insert a Restaurant, a Meal and a reservation in the backend because I couldn't implement a system to insert automatically that specific data when que docker compose is runned.

I tried to implement the SonarQube for analysis but I didn't work like I expected.

I also tried to implement the Implement Quality Gate enforcement in a continuous integration (CI) workflow but I think it didn't work like it was expected.

1.3 Functional scope and supported interactions

The application targets two primary categories of users: students and food services workers.

Students
Actors: Individuals enrolled at the Moliceiro University who wish to plan their meals on campus.
Purpose: Students use the application to view the meals available in different food facilities and make reservations based on their preferences and weather conditions.
Main Usage Scenarios:

- A student accesses the homepage, selects a restaurant unit from a dropdown menu, chooses a specific date, and confirms their selection. The application displays the planned meals for that day, along with the weather forecast for the selected location.

- If interested in a meal, the student clicks on it to start a reservation. A confirmation modal appears, and upon confirming the reservation, the student receives a success message with a unique reservation code.

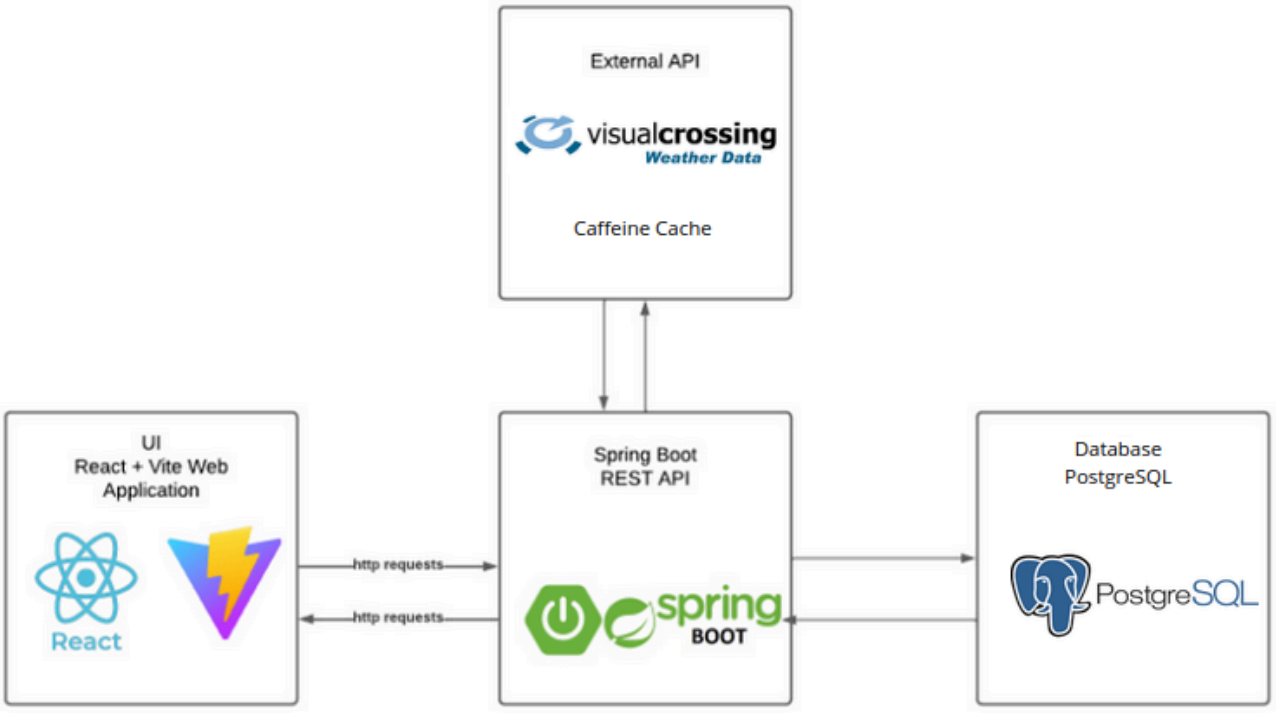
- Later, the student can visit the "My Reservations" page to view active bookings. If needed, they can cancel a reservation by selecting it and confirming the cancellation, after which the reservation is removed and a success message is shown.

Food Services Workers
Actors: Employees responsible for managing and verifying meal reservations at the university's food facilities.
Purpose: Workers use the application to validate reservation codes presented by students at the time of meal collection.

Main Usage Scenario:

- A worker accesses the "admin" page and views the list of pending reservations. Upon selecting a reservation, they can mark it as used. Once verified, the reservation status changes to "Used", and the option to verify it again is disabled to prevent duplicate validations.

1.4 System implementation architecture



The application follows a **three-tier architecture**, with clear separation between the frontend, backend, and data/external service layers:

➔ **Frontend (UI Layer):**

a. A web interface built using **React** with **Vite** as the build tool, which provides a fast and modern development environment.

b. The frontend communicates with the backend via HTTP requests, allowing users to interact with the system—select restaurants, view meals and forecasts, make/cancel reservations, etc.

➔ **Backend (Application Layer):**

a. A **Spring Boot REST API** handles the core business logic, processes client requests, manages reservations and meals, and communicates with the database and external APIs.

b. It acts as the central coordinator between the UI, database, and external weather service.

c. To improve performance and reduce unnecessary external API calls, the backend uses **Caffeine Cache**, a high-performance caching library, to store weather forecast results temporarily.

➔ **External API Integration:**

a. The backend integrates with the **Visual Crossing Weather API** to fetch weather forecasts.

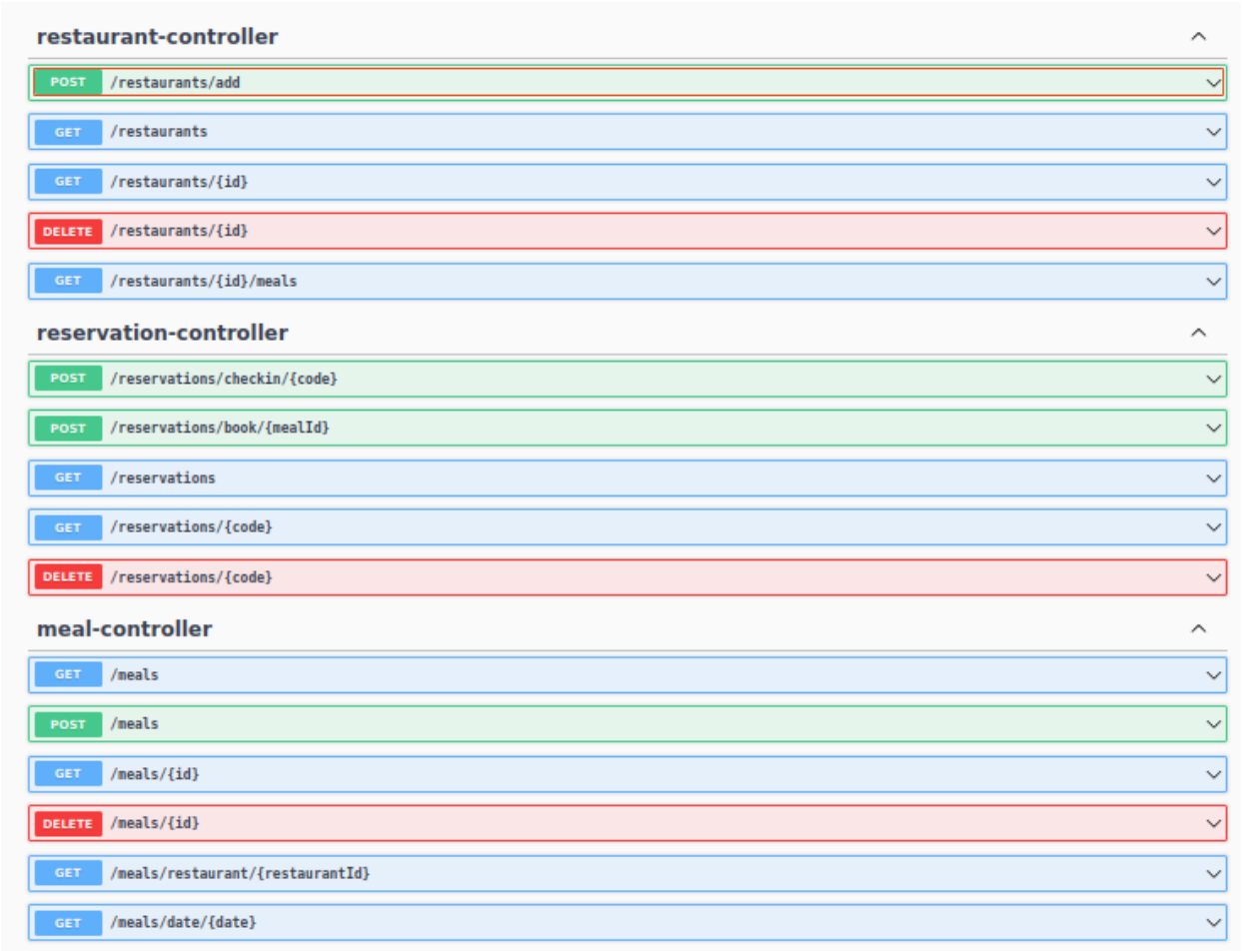
b. This data is used to enhance the meal planning experience by informing students of the weather conditions for selected dates.

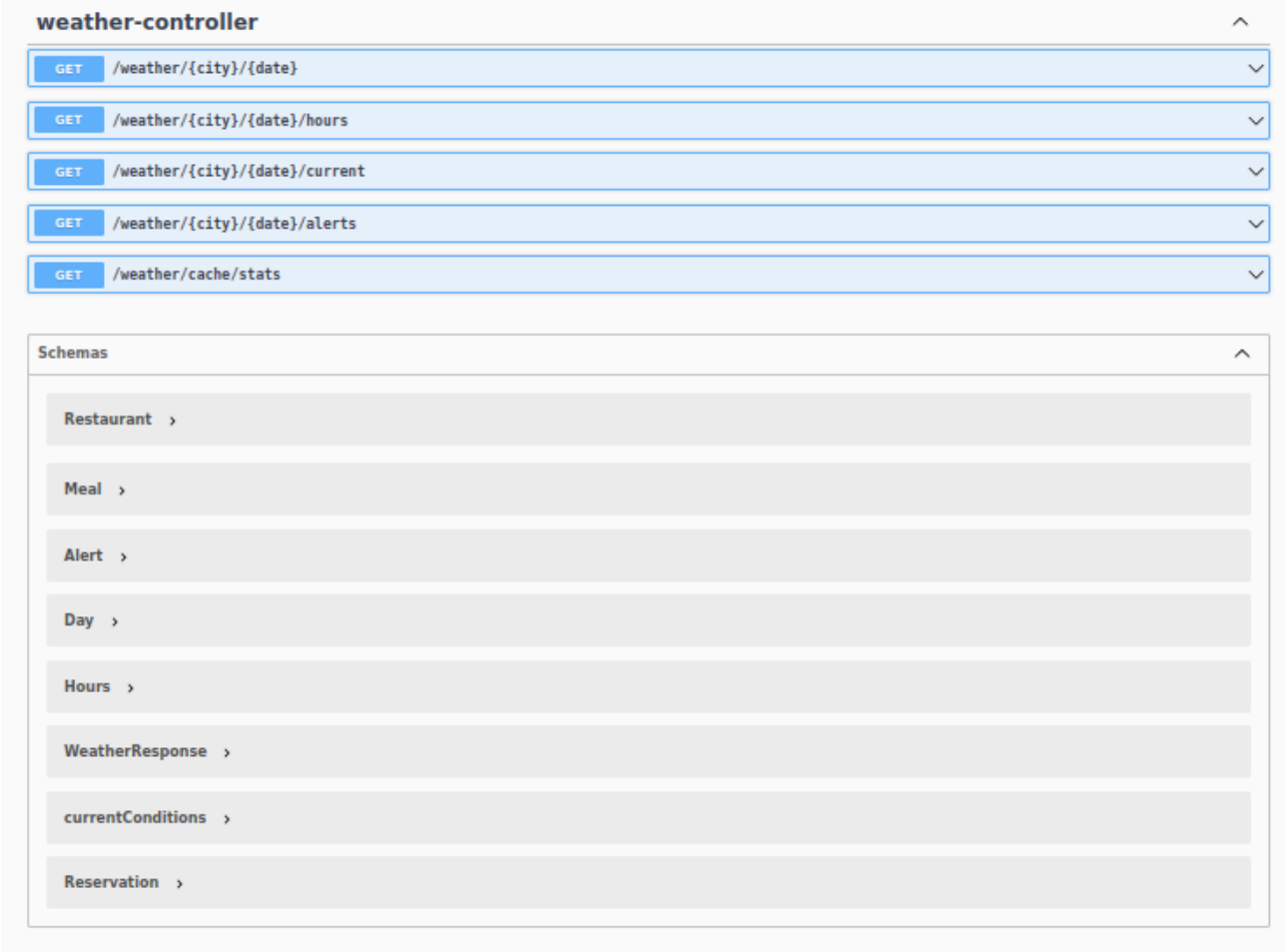
➔ **Database Layer:**

a. A **PostgreSQL** database stores structured data such as restaurant information, planned meals, reservations, and user interactions.

b. The backend accesses this database to perform CRUD operations and maintain data consistency.

1.5 API for developers





2 Quality assurance

2.1 Overall strategy for testing

The overall test development strategy employed in the project was centered around Behavior-Driven Development (BDD) using several tools like Cucumber, Junit, Mockito, etc. The strategy Test-Driven Development (TDD) was not adopted for this project.

For service level testing, I utilized Mockito to create mocks for isolating dependencies, particularly to test scenarios detached from external data providers.

In terms of integration testing, I relied on Spring Boot MockMvc and also integration with and REST-Assured with Flyway to perform tests on my API endpoints.

2.2 Unit and integration testing

In my project, I implemented both unit and integration tests to ensure the functionality and reliability of the application.

Unit tests were created for critical components such as input validation and some cache

behaviors. These tests validate the behavior of individual units of code in isolation to ensure they function as expected.

Examples:

In the **ReservationServiceTest** class implements unit tests to ensure the correct functioning of the reservation service in a meal management system. Using Mockito, the tests simulate the behavior of the ReservationRepository, isolating the service from the database and controller layers.

The tests are set up using @Mock and @InjectMocks annotations, with mocks initialized in the setUp() method before each test. Key scenarios include verifying the correct creation of reservations, handling valid check-ins, and managing error cases like attempting to check-in a used reservation or creating a reservation when the restaurant is at capacity.

The tests also cover the deletion of reservations, ensuring proper functionality when a valid reservation code is provided. These unit tests validate critical business rules, enhance system reliability, and reduce the likelihood of production failures.

```
You, 3 hours ago | 1 author (You)
public class ReservationServiceTest {

    @Mock
    private ReservationRepository reservationRepository;

    @InjectMocks
    private ReservationService reservationService;

    @BeforeEach
    void setUp() {
        MockitoAnnotations.openMocks(this);
    }

    // Teste de reserva criada corretamente
    @Test
    void shouldCreateReservation() {
        Restaurant restaurant = new Restaurant("Rest A");
        Meal meal = new Meal("Fish & Chips", LocalDate.now(), restaurant);
        Reservation savedReservation = new Reservation(UUID.randomUUID().toString(), LocalDate.now().atStartOfDay(), false, meal);

        when(reservationRepository.save(any(Reservation.class))).thenReturn(savedReservation);

        Reservation reservation = reservationService.createReservation(meal);
        assertThat(reservation.getMeal().getDescription()).isEqualTo("Fish & Chips");
        assertThat(reservation.isUsed()).isFalse();
    }

    // Teste de verificação de reserva (check-in)
    @Test
    void shouldCheckInReservation() {
        Reservation reservation = new Reservation("CHECK123", LocalDate.now().atStartOfDay(), false, null);
        when(reservationRepository.findByCode("CHECK123")).thenReturn(Optional.of(reservation));

        boolean checkedIn = reservationService.checkInReservation("CHECK123");
        assertThat(checkedIn).isTrue();
        verify(reservationRepository).save(reservation);
    }

    // Teste para verificar se o ticket já foi utilizado
    @Test
    void shouldRejectCheckInIfReservationAlreadyUsed() {
        Reservation reservation = new Reservation("CHECK123", LocalDate.now().atStartOfDay(), true, null); // Já usado
        when(reservationRepository.findByCode("CHECK123")).thenReturn(Optional.of(reservation));

        assertThrows(IllegalStateException.class, () -> {
            reservationService.checkInReservation("CHECK123");
        });

        verify(reservationRepository, never()).save(any(Reservation.class)); // Garante que não tentou salvar
    }
```

```java
// Teste para garantir que a reserva não é permitida se o limite de capacidade foi atingido
@Test
void shouldNotAllowReservationIfRestaurantIsFull() {
    Restaurant restaurant = new Restaurant("Rest A");
    restaurant.setId(1L);
    Meal meal = new Meal("Steak", LocalDate.now(), restaurant);

    when(reservationRepository.countByMeal_DateAndMeal_Restaurant_Id(LocalDate.now(), 1L)).thenReturn(50L);

    assertThrows(IllegalStateException.class, () -> {
        reservationService.createReservation(meal);
    });
    You, yesterday • alterações no modo de reservation service do ba…
    verify(reservationRepository, never()).save(any());
}


// Teste para garantir que a reserva não seja feita em um dia sem serviço (sem refeições)
@Test
void shouldNotAllowReservationOnDayWithoutService() {
    Restaurant restaurant = new Restaurant("Rest A");
    restaurant.setId(2L);
    Meal meal = new Meal("Pasta", LocalDate.now().plusDays(1), restaurant);

    when(reservationRepository.countByMeal_DateAndMeal_Restaurant_Id(meal.getDate(), 2L)).thenReturn(50L);

    assertThrows(IllegalStateException.class, () -> {
        reservationService.createReservation(meal);
    });

    verify(reservationRepository, never()).save(any());
}

// Teste para garantir que a deleção de reserva funciona corretamente
@Test
void shouldDeleteReservation() {
    Reservation reservation = new Reservation("DELETE123", LocalDate.now().atStartOfDay(), false, null);
    when(reservationRepository.findByCode("DELETE123")).thenReturn(Optional.of(reservation));
    doNothing().when(reservationRepository).delete(any(Reservation.class));

    boolean deleted = reservationService.deleteReservationByCode("DELETE123");
    assertThat(deleted).isTrue();
    verify(reservationRepository).delete(reservation); // Verificando se o delete foi chamado no repositório
}

// Teste para garantir que não é possível deletar uma reserva que não existe
@Test
void shouldReturnFalseIfReservationCodeNotFoundOnDelete() {
    when(reservationRepository.findByCode("INVALID")).thenReturn(Optional.empty());
    boolean deleted = reservationService.deleteReservationByCode("INVALID");
    assertThat(deleted).isFalse();
    verify(reservationRepository, never()).delete(any(Reservation.class));
}
```

In the **WeatherControllerTest** class test, one test focuses specifically on validating cache statistics: testGetCacheStats. This test ensures that the system is correctly keeping track of cache-related data, such as the number of total requests, cache hits, and cache misses.

The test begins by mocking the cache statistics that the **WeatherService** should return. Specifically, the *getTotalRequests*(), *getCacheHits*(), and *getCacheMisses*() methods are mocked to return values representing the total number of requests, the number of successful cache lookups (cache hits), and the number of times the requested data wasn't found in the cache (cache misses). The test then simulates an HTTP GET request to the /weather/cache/stats endpoint. This endpoint is expected to return the cache statistics that were mocked earlier.

After the request is made, the test checks the response for the correct cache statistics. It validates that the response contains the expected values for total requests, cache hits, and cache misses. The assertions ensure that the response contains the relevant data as a string.

By using this approach, the test ensures that the cache statistics are accurately tracked and displayed through the appropriate endpoint. This verifies that the application is properly interacting with the cache layer and maintaining its internal counters for cache hits and misses.

```java
@Test
@DisplayName("GET /weather/cache/stats returns cache statistics")
void testGetCacheStats() throws Exception {
    System.out.println("🔍 Iniciando teste: testGetCacheStats");

    when(weatherService.getTotalRequests()).thenReturn(5);
    when(weatherService.getCacheHits()).thenReturn(3);
    when(weatherService.getCacheMisses()).thenReturn(2);

    System.out.println("📡 A fazer chamada GET para /weather/cache/stats");
    mvc.perform(get("/weather/cache/stats"))
            .andExpect(status().isOk())
            .andExpect(content().string(containsString("Total Requests: 5")))
            .andExpect(content().string(containsString("Cache Hits: 3")))
            .andExpect(content().string(containsString("Cache Misses: 2")));

    System.out.println("✅ Estatísticas de cache verificadas corretamente");
}
```

Lastly, integration tests can also ensure that database interactions work correctly. Let's first see the integration tests in the context of a Spring Boot application, using the MockMvc framework to simulate HTTP requests and verify the interaction between the controller and the services.

The print below shows an **integration test** focused on the *DELETE /reservations/{code}* endpoint of a Spring Boot application. The test verifies the controller's behavior when attempting to delete a reservation using its reservation code.

This is a **controller integration test**. It checks if the controller correctly handles HTTP DELETE requests, interacts with the reservationService, and responds with the expected outcomes. The reservationService is mocked to simulate interactions without using a real database

The test evaluates two scenarios:

1. **Successful Deletion:** When a valid reservation code (e.g., "ABC123") is provided, the system should successfully delete the reservation and return a 200 OK response.

2. **Failure on Reservation Not Found:** If a non-existent reservation code (e.g., "XYZ999") is provided, the system should return a 404 Not Found response with the message "Reservation not found."

Steps:

- **Mock Setup:** For the successful case, reservationService is mocked to return true for code "ABC123". For the failure case, it's mocked to return false for code "XYZ999".

- **Simulated HTTP Requests:** The first test sends a DELETE request with the valid code, while the second uses an invalid code.

- **Validation:** In the successful case, the response is validated as 200 OK, confirming the deletion. In the failure case, the response is 404 Not Found, with the appropriate error message.

```java
@Test
@DisplayName("DELETE /reservations/{code} - Deve deletar uma reserva com sucesso")
void whenDeleteReservation_thenReturnSuccess() throws Exception {
    when(reservationService.deleteReservationByCode("ABC123")).thenReturn(true);

    mvc.perform(delete("/reservations/ABC123")
                    .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(content().string("Reservation deleted successfully"));
}

@Test
@DisplayName("DELETE /reservations/{code} - Deve retornar erro quando reserva não for encontrada")
void whenDeleteReservation_thenReturnNotFound() throws Exception {
    when(reservationService.deleteReservationByCode("XYZ999")).thenReturn(false);

    mvc.perform(delete("/reservations/XYZ999")
                    .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isNotFound())
            .andExpect(content().string("Reservation not found"));
}
```

The tests below in the **ReservationControllerRestTemplateIT** with *Flyway migration* class are focused on testing the *ReservationController* endpoints using integration tests with TestRestTemplate. These tests verify the behavior of the reservation system through various scenarios, ensuring that the system handles requests and responses correctly, both for successful operations and error cases.

The scenario below focus on the **POST /reservations/checkin/{code}** endpoint in a Spring Boot application, verifying the check-in process for reservations. These tests ensure the correct behavior when marking a reservation as checked-in and handle cases where the reservation is not found.

This is a controller integration test that checks:

1. **Successful Check-in:** When a valid reservation code (e.g., "ABC123") is provided, the system should successfully mark the reservation as checked in and return a 200 OK response with a message confirming the success ("Check-in successful").

2. **Failure on Reservation Not Found:** If a non-existent reservation code (e.g., "XYZ999") is provided, the system should return a 404 Not Found response, along with an error message ("Reservation not found or already used").

Steps:

- **Mock Setup:** The test uses TestRestTemplate to simulate HTTP requests. For the successful case, a valid reservation code is passed. For the failure case, a non-existent reservation code is tested.

- **Simulated HTTP Requests:** The first test sends a POST request to check in a reservation with a valid code, and the second test attempts the same with an invalid reservation code.

- **Validation:** In the successful case, the response is validated as 200 OK, and the body contains the "Check-in successful" message. In the failure case, the response is validated as 404 Not Found, with the appropriate error message indicating that the reservation is not

found or has already been used.

```java
@Test
@DisplayName("POST /reservations/checkin/{code} - Deve marcar check-in com sucesso")
void whenCheckIn_thenReturnSuccess() {
    logger.info("Testing POST /reservations/checkin/{code}...");
    String url = "http://localhost:" + randomServerPort + "/reservations/checkin/" + reservation.getCode();

    ResponseEntity<String> response = testRestTemplate.postForEntity(url, null, String.class);

    logger.info("Received response status: {}", response.getStatusCode());
    assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
    assertThat(response.getBody()).contains("Check-in successful");
}

@Test
@DisplayName("POST /reservations/checkin/{code} - Deve retornar erro quando reserva não for encontrada")
void whenCheckIn_thenReturnNotFound() {
    logger.info("Testing POST /reservations/checkin/{code} for non-existent reservation...");
    String url = "http://localhost:" + randomServerPort + "/reservations/checkin/XYZ999";

    ResponseEntity<String> response = testRestTemplate.postForEntity(url, null, String.class);

    logger.info("Received response status: {}", response.getStatusCode());
    assertThat(response.getStatusCode()).isEqualTo(HttpStatus.NOT_FOUND);
    assertThat(response.getBody()).contains("Reservation not found or already used");
}
```

2.3 Functional testing

For the user-facing test cases, multiple scenarios were considered for the reservation system, focusing on the user's ability to select a restaurant, make a reservation, and manage their reservations. These scenarios were designed to ensure the system works correctly without errors. The following feature file was defined:

```
Feature: View meals and weather forecast for selected restaurant

  Scenario: User selects a restaurant and date to view planned meals and weather forecast
    Given the user navigates to the homepage
    When the user selects a restaurant from the dropdown
    And the user selects a date for the meals
    And the user confirms the selection
    Then the meals for the selected restaurant and date should be displayed
    And the weather forecast for the selected date should be shown for the chosen city

  Scenario: User makes a reservation for a meal
    Given the user navigates to the homepage
    When the user selects a restaurant from the dropdown
    And the user selects a date for the meals
    And the user confirms the selection
    And the user clicks on a meal to make a reservation
    Then a confirmation modal should be displayed
    When the user confirms the reservation
    Then the success modal should be displayed with a reservation code

  Scenario: Food services worker verifies a reservation
    Given the admin navigates to the "admin" page
    And there is at least one reservation with status "Pendente"
    When the admin clicks to mark the reservation as used
    Then the reservation status should change to "Usada"
    And the verify button should no longer be visible

  Scenario: The user checks the active reservation and cancels it
    Given the user navigates to the "My Reservations" page
    When the user sees the list of active reservations
    And the user chooses to cancel a reservation
    And the user confirms the cancellation
    Then the reservation should be removed from the list
    And a success message should be displayed
```

2.4 Non functional testing

For non-functional testing, I focused on **performance testing of the backend** using **K6**. The goal was to evaluate the system's ability to handle various levels of concurrent requests and ensure that the performance remains within expected limits under real-world usage scenarios. The test scenarios were designed to simulate user interactions with the system and assess how well the backend responds under different loads.

The following test scenarios were defined and executed during the load testing:

1. **Viewing Meals and Weather Forecast**: The goal of this scenario was to check if the system can provide meals for a specific restaurant and weather forecast for a selected date and city without compromising performance.

2. **Making a Reservation**: This scenario simulates the process of making a reservation for a meal, ensuring the system responds correctly and includes a valid reservation code in the response.

3. **Checking an Existing Reservation**: This scenario tests the system's ability to return accurate information about an existing reservation when provided with the reservation code.

4. **Marking a Reservation as Used**: After checking a reservation, this scenario simulates the process of marking a reservation as "used" by the admin, ensuring the reservation's status is correctly updated.

5. **Deleting a Reservation**: This scenario tests the system's ability to delete a reservation successfully, ensuring the response confirms the reservation has been deleted.

The load testing configuration was designed based on **performance limits** with clearly defined thresholds. The following metrics were used to measure the system's effectiveness:

- **Response Time**: At least 95% of requests should be responded to in under 1 second.

- **Error Rate**: The failure rate of requests should be less than 5%.

- **Success Rate of Checks**: At least 83% of check tests (such as verifying successful responses) should pass.

Additionally, the test was structured into different **load stages** to simulate varying traffic scenarios and observe how the system behaves under different levels of demand. The test stages were as follows:

- **Stage 1**: Gradually ramp up to 1 user over 10 seconds.

- **Stage 2**: Maintain 1 user for 30 seconds.

- **Stage 3**: Gradually scale down to 0 users over 10 seconds.

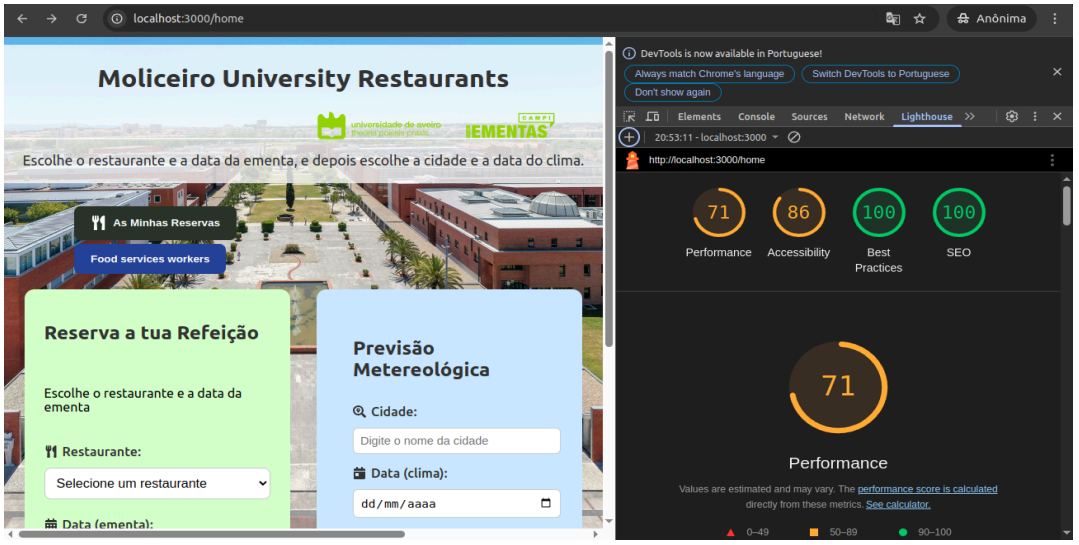
These load tests provide valuable insights into the performance of the system under real-world conditions, ensuring that the backend can handle user traffic efficiently and without failure, maintaining service quality and adhering to the established response times and error limits.

```
TOTAL RESULTS

  checks_total.....................: 2860    57.119176/s
  checks_succeeded.................: 100.00% 2860 out of 2860
  checks_failed....................: 0.00%   0 out of 2860

  ✓ Refeições carregadas com sucesso
  ✓ Verifica se o restaurante tem refeições
  ✓ Previsão do tempo carregada com sucesso
  ✓ Reserva realizada com sucesso
  ✓ Verifica código de reserva
  ✓ Reserva verificada com sucesso
  ✓ Verifica código da reserva
  ✓ Reserva 'Pendente' verificada com sucesso
  ✓ Reserva marcada como usada com sucesso
  ✓ Verifica o status da reserva como 'Usada'
  ✓ Obter reservas com sucesso
  ✓ Reserva deletada com sucesso
  ✓ Verifica a mensagem de sucesso

  HTTP
  http_req_duration.....: avg=27.99ms  min=4.62ms  med=21.85ms  max=3.24s p(90)=43.55ms  p(95)=56.1ms
    { expected_response:true }...: avg=27.99ms  min=4.62ms  med=21.85ms  max=3.24s p(90)=43.55ms  p(95)=56.1ms
  http_req_failed..........: 0.00%  0 out of 1760
  http_reqs................: 1760   35.150262/s

  EXECUTION
  iteration_duration.....: avg=227.55ms min=97.83ms med=183.39ms max=4.74s p(90)=315.38ms p(95)=362.74ms
  iterations.....: 220    4.393783/s
  vus.....: 1      min=1        max=1
  vus_max.....: 1      min=1        max=1

  NETWORK
  data_received.....: 6.2 MB 123 kB/s
  data_sent.....: 217 kB 4.3 kB/s


running (0m50.1s), 0/1 VUs, 220 complete and 0 interrupted iterations
default ✓ [======================================] 0/1 VUs  50s
```

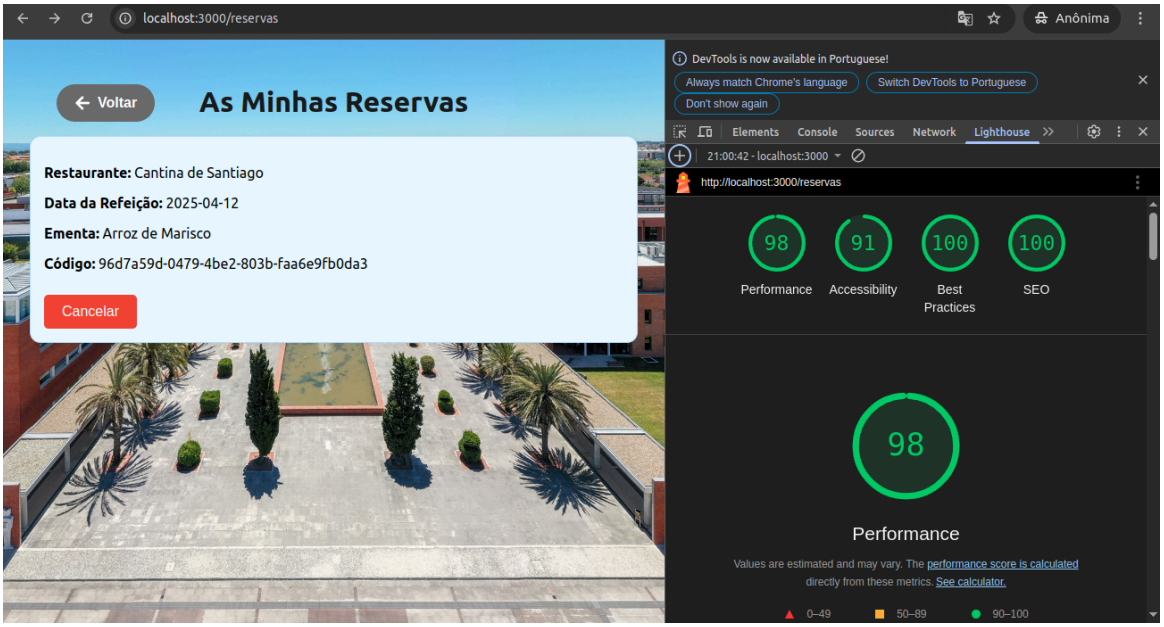
For the **frontend performance testing**, I used **Lighthouse** in Google Chrome to analyze the performance of the three main pages of the application: **Home**, **Reservations**, and **Admin**. The goal was to ensure that the frontend pages load efficiently, providing a smooth and responsive user experience, and to identify areas for improvement.

Lighthouse is an automated tool for improving the quality of web pages. It audits pages for performance, accessibility, SEO, and more. The tool generates a report with various metrics, including **Performance**, **Accessibility**, **Best Practices**, and **SEO**, with scores out of 100 for each category. For this testing, I focused on the **Performance** category, which provides key indicators such as page load time, interactivity, and visual stability.

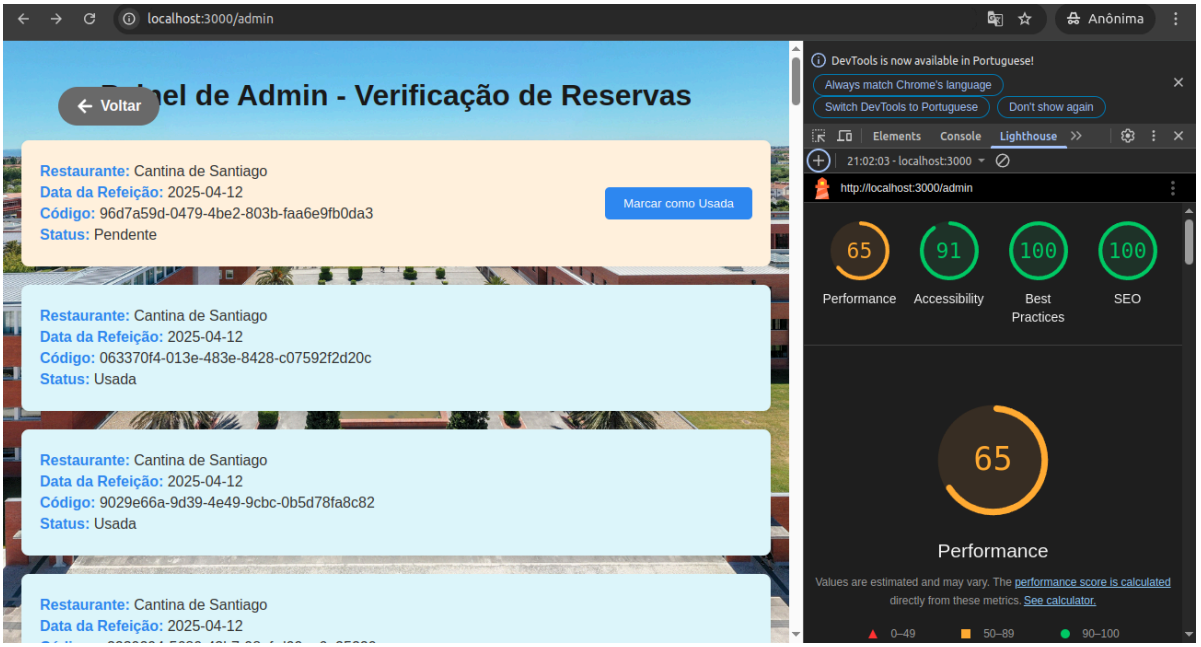
Home:



Reservas



Admin:



Each of the three pages was tested under normal network conditions, and the following insights were gathered from the Lighthouse reports:

- **Home Page**: The **FCP** and **LCP** scores were within the recommended thresholds, but there were areas for improvement in terms of **TBT** and **TTI**, suggesting that some resources or scripts were blocking the page's interactivity.

- **Reservations Page**: This page had a good **FCP** and **LCP**, but the **TTI** score could be improved to make the page interactive more quickly. The **CLS** score was within acceptable

limits, ensuring no unexpected shifts when users interacted with reservation elements.

- **Admin Page**: The **Admin** page had a high **TBT** due to the loading of large datasets, so optimization techniques such as lazy loading or pagination could improve performance. The **CLS** score was also satisfactory, indicating a stable layout.

Based on the findings, I identified areas that need optimization, particularly in reducing **Total Blocking Time** and improving **Time to Interactive**. Further optimization of JavaScript and CSS loading strategies, as well as the use of lazy loading and efficient rendering techniques, will help me enhance the performance of the application and provide a better user experience across all pages.

2.5 Code quality analysis

For static code analysis, it was utilized SonarCloud, a powerful tool that provides automated code review to detect bugs, vulnerabilities, and code smells in a codebase. Interpreting the results from SonarCloud allows us to gain insights into the overall health of our codebase and identify areas for improvement. By addressing the issues reported by SonarCloud, I can enhance the maintainability, reliability, and security of our application. One particular code smell that it wasn't solved and I also didn't understand it well was field injection with the @Autowired element, Sonar Cloud advertise and suggest use constructor injection instead,

3 References & resources

Project resources

Resource:	URL/location:
Git repository	https://github.com/carolinaspsilva2004/TQS
Video demo	o vídeo encontra-se no repositório

Reference materials

https://www.baeldung.com/junit-5
https://www.exchangerate-api.com/
https://docs.cucumber.io/
https://www.selenium.dev/documentation/
https://site.mockito.org/
https://www.baeldung.com/rest-template
https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/