HW1: Mid-term assignment report

Rodrigo Marques de Jesus [113526], v2025-04-09-04-09

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.
The application, named “Meal Booking Manager”, is a full-stack web solution developed for managing meal bookings on the Moliceiro University campus. The application allows students to view available meals (with corresponding weather forecasts), create reservations (issuing a unique token), and check reservation status. Additionally, food service staff can verify a reservation and mark it as used.

1.2 Current limitations

- The list of restaurants is not dynamically generated from the database. Currently, users must manually enter a restaurant ID in the form.
- Some advanced booking features (e.g., group bookings or selection from a detailed menu) are not implemented.
- User authentication has not been implemented, as it is not a mandatory requirement.

2 Product specification

2.1 Functional scope and supported interactions

Actors:

a) **Students:** They can view upcoming meal options for a given restaurant unit (including weather forecasts), book a meal in advance, and check the status of their reservation using a generated token.

b) **Food Service Staff:** They can verify an existing reservation by entering the token, and subsequently mark the reservation as used.

User Experience Summary:

c) Upon accessing the application, users are presented with a simple page where they can enter the restaurant ID and the desired date range.

d) After submission, the application lists the available meals along with the weather forecast for the selected dates.
(Insert screenshot of the restaurant selection page)

e) Users can then select a meal and proceed to book it through a dedicated reservation form.

f) Once a reservation is created, the system returns a unique token that can be used later to consult the reservation status or for check-in by the staff.

g) A separate support page is provided for staff to validate and mark reservations as used.

2.2 System implementation architecture

h) **Backend:**
The backend is built with Spring Boot using JPA/Hibernate for ORM and H2 as an in-memory database for development purposes. A REST API is exposed for external integration and for internal service communication.

i) **Frontend:**
A minimalist user interface is implemented using Thymeleaf. The HTML pages are rendered on the server and handle user inputs for restaurant selection, booking, and reservation verification.

j) **API Exposure:**
Two main groups of endpoints are available:

k) **Domain Endpoints:**
For listing meal options, booking a meal, checking reservation details, and performing check-in.

l) **Cache Monitoring Endpoints:**
For monitoring the performance and usage of the external weather API cache (e.g., returning hit/miss counts).

Technology Stack:

m) **Backend:** Spring Boot, JPA, H2

n) **Frontend:** Thymeleaf

o) **Testing:** JUnit 5, Mockito, Spring Boot Test (MockMvc), Selenium WebDriver, k6

p) **Code Quality:** SonarQube (or Codacy)

2.3 API for developers

The application exposes two groups of REST endpoints that allow developers to interact with the system:

q) **Problem Domain Endpoints**
These endpoints provide core functionalities related to the meal booking process. They include:

r) **List Meals:**

s) **Endpoint:** GET /api/meals

t) **Description:** Returns a list of meals (or menu options) planned for the specified restaurant unit within the provided date range. The response also includes weather forecast details for each day.

u) **Book a Meal:**

v) **Endpoint:** POST /api/reservations

w) **Description:** Creates a new reservation. The response contains a generated token (reservationCode) that uniquely identifies the reservation.

x) Check Reservation Details:

y) **Endpoint:** GET /api/reservations/{reservationCode}

z) **Path Variable:** reservationCode: the token returned when a reservation is created

aa) **Description:** Returns the details of the reservation including the status (whether it has been used) and other relevant information.

bb) Reservation Check-in (Mark as Used):

cc) **Endpoint:** PATCH /api/reservations/{reservationCode}/use (or PUT, as per implementation)

dd) **Path Variable:** reservationCode: the reservation token

ee) **Description:** Verifies if the reservation is valid and marks it as used (check-in for the restaurant staff).

ff) Optional – Cancel Reservation:

gg) **Endpoint:** DELETE /api/reservations/{reservationCode}

hh) **Path Variable:** reservationCode: the reservation token

ii) **Description:** Cancels the reservation, if cancellation is supported.

jj) **Cache Usage Statistics Endpoints**
These endpoints provide information about the performance and usage of the weather forecast caching mechanism:

kk) Weather Cache Statistics:

ll) **Endpoint:** GET /api/weather/cache

mm) **Description:** Returns statistics regarding the weather cache including:

nn) Total requests to the weather API

oo) Number of cache hits

pp) Number of cache misses

This allows developers to monitor the efficiency of the cache, ensuring that repeated requests for the same date do not unnecessarily call the external API.

3 Quality assurance

3.1 Overall strategy for testing

The overall testing strategy was designed to ensure a robust, high-quality application covering both functional and non-functional aspects. In our approach we combined several testing techniques and tools:

qq) **Test-Driven Development (TDD):**
Although we did not fully embrace TDD, many critical business scenarios—especially those involving the reservation logic and cache behavior—were first outlined through tests before being implemented or refactored.

rr) **Behavior-Driven Development (BDD):**
For user-facing features, we considered a BDD approach using tools such as Cucumber (although not fully integrated, we aligned our functional tests in the spirit of BDD by simulating real user interactions through Selenium).

ss) **Mixed Testing Tools:** For API integration and service-level testing, we used **MockMvc**.

tt) **Unit tests** were written with JUnit 5 and Mockito to isolate logic and dependencies.

uu) **Functional (UI) tests** used Selenium WebDriver to simulate end-user interactions.

vv) For performance, we chose **k6** for load testing.

3.2 Unit and integration testing

Unit Testing:

ww) **Scope:**

Unit tests were implemented for core services such as MealService, ReservationService, and WeatherService. These tests focused on validating the business logic: for example, the token generation during reservation, the correct handling of reservation status (e.g., marking a reservation as used), and the behavior of the weather cache (hits/misses logic).

xx) **Strategy:**

Mocking was extensively used (via Mockito) to simulate interactions with repository interfaces and external services (like the weather API). This allowed us to isolate business logic and ensure that unit tests were fast and repeatable.

```
@Test
void testCreateReservation() {
    Long mealId = 10L;
    LocalDate date = LocalDate.of(2025, 4, 15);
    String studentName = "Joao";

    ArgumentCaptor<Reservation> captor = ArgumentCaptor.forClass(clazz:Reservation.class);
    when(reservationRepository.save(any(type:Reservation.class))).thenAnswer(inv -> inv.getArgument(index:0));

    Reservation r = reservationService.createReservation(mealId, date, studentName);

    assertNotNull(r.getReservationCode());
    assertEquals(mealId, r.getMealId());
    assertEquals(date, r.getDate());
    assertFalse(r.isUsed());

    verify(reservationRepository, times(wantedNumberOfInvocations:1)).save(captor.capture());
    Reservation saved = captor.getValue();
    assertEquals(studentName, saved.getStudentName());
}
```

Integration Testing:

yy) **Scope:**

Integration tests were written to test the end-to-end behavior of the REST endpoints. For example, we validated the complete process of creating a reservation via the /api/reservations endpoint, and checked the response, including the generated token and correct JSON structure.

zz) **Strategy:**

We used Spring Boot Test with **MockMvc** to simulate HTTP requests to the API and validate the responses against an in-memory H2 database.

3.3 Functional testing

User-Facing Test Cases:

aaa) **Scope:**

Functional testing focused on the key user journeys. For example, we validated that a user can select a restaurant and date range, fill out the reservation form, and subsequently view the list of available meals. Additionally, tests were created to verify that the staff page correctly validates and marks a reservation as used.

bbb) **Implementation:**

We used **Selenium WebDriver** to simulate user interactions with the web interface. The tests mimic a real user by filling out the input fields (restaurant ID, date fields), clicking the "Ver refeições" button, and then checking that the user is redirected to the correct page.

3.4 Non functional testing

Performance Testing with k6:

ccc) **Scope:**

Non-functional tests were conducted to evaluate the performance of the API endpoints under load. Specifically, we focused on scenarios such as simulating a ramp-up of users, maintaining a constant load, and then ramping down, while measuring critical performance metrics.

ddd) **Implementation:**

We developed k6 test scripts to simulate multiple virtual users (VUs) performing actions such as booking a reservation or retrieving meal lists. Thresholds were defined so that 95% of requests must be completed within 1.1 seconds and the failure rate stays below 1%.

3.5 Code quality analysis

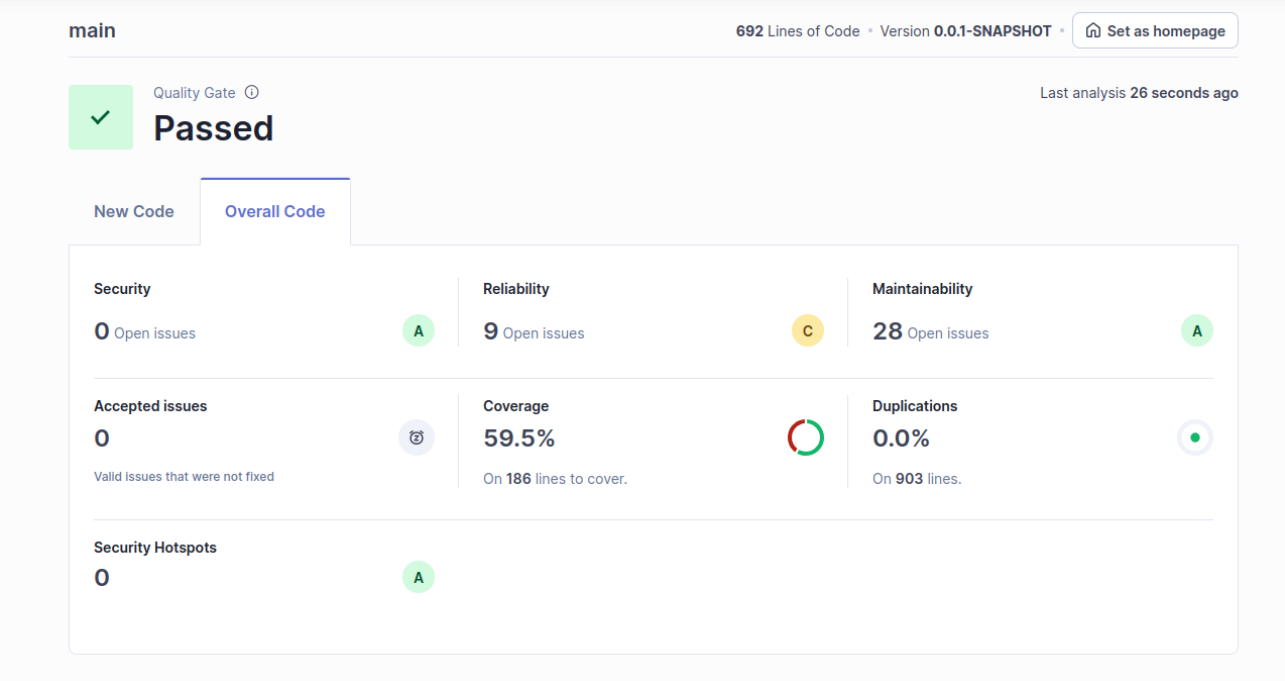
Static Code Analysis:

eee) **Tool Integration:**

The project was integrated with SonarQube (or Codacy) to continuously monitor various code quality metrics including test coverage, code duplication, complexity, and potential vulnerabilities.

fff) **Results:**

The analysis confirmed a high level of test coverage and minimal code duplications. Some areas, such as complex business logic in the booking services, were identified as potential candidates for further refactoring to improve maintainability.



4 References & resources

Project resources

Resource:	URL/location:
Git repository	https://github.com/R0drigOO/tqs_113526
Video demo	HW1/docs