

HW1: Mid-term assignment report

João Manuel Vieira Roldão [113920], v2025-04-09

1 Introduction	1
1.1 Overview of the work	1
1.2 Current limitations	1
2 Product specification	2
2.1 Functional scope and supported interactions.	2
2.2 System implementation architecture	2
2.3 API for developers	4
3 Quality assurance	5
3.1 Overall strategy for testing	5
3.2 Unit testing	5
3.3 Integration testing	5
3.4 Functional testing	5
3.5 Non functional testing	6
3.6 Code quality analysis	6
4 References & resources	7

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 main goal of this web application is to facilitate meal reservations at various restaurants on the university campus. Students can check available meal plans for different restaurants, make bookings for their preferred meals, and monitor reservation details. The weather integration ensures that students can make informed decisions based on the forecasted weather.

1.2 Current limitations

Currently there are a lot of features still not fully implemented in the project.

Weather API setup exists but it isn't fully implemented.

Reservations are implemented just not fully working in the frontend side.

2 Product specification

2.1 Functional scope and supported interactions.

The Meal Booking Web Application will be used by several types of actors, each with specific roles and interactions with the system. Below is an explanation of the main actors, their interactions with the system, and their experience in using the application, followed by a visual summary of these interactions.

Main Actors:

- Students
 - Goal: Students use the application to view available meals and book reservations
 - Main Interactions:
 - Browse Meals: Students can view meal options for various campus restaurants.
 - Book a Reservation: Based on meal availability and weather conditions, students can reserve a meal at their chosen restaurant.
 - Cancel Reservation: If plans change, students can cancel their reservation ahead of time.
- Restaurant Staff
 - Goal: Restaurant staff use the application to verify and mark student reservations as used when the students arrive at the restaurant.
 - Main Interactions:
 - Mark Reservation as Used: After a student arrives and receives their meal, staff mark the reservation as used, which helps in tracking attendance and availability.
- System Administrators
 - Goal: Admins manage the backend system, monitor user activity, ensure system performance, and handle any issues with the application.
 - Main Interactions:
 - Delete Potencial Hazardous Users or Staff: If any user is found to be engaging in inappropriate, harmful, or suspicious activities, administrators are authorized to delete their accounts

2.2 System implementation architecture

The Meal Booking Web Application follows a microservices architecture that divides the system into discrete services, enabling scalability, flexibility, and easier maintenance. The architecture is designed to efficiently manage meal reservations, user data, and integrate weather forecasts to guide students in their dining decisions.

At the frontend, the application is built with Vite/React with the DaisyUI and Tailwind libraries.

The backend of the system is structured around two key services, both of which are built using Spring Boot. The Meal Service manages meal reservations and restaurant information, storing and retrieving data from a MySQL database specifically designed for meal-related information. It also communicates with an external Visual Crossing Weather API to provide real-time weather data, influencing meal reservation decisions. The User Service handles authentication and user

management, connecting to a separate MySQL database dedicated to user data. This separation of concerns ensures that the application is modular and can be scaled independently.

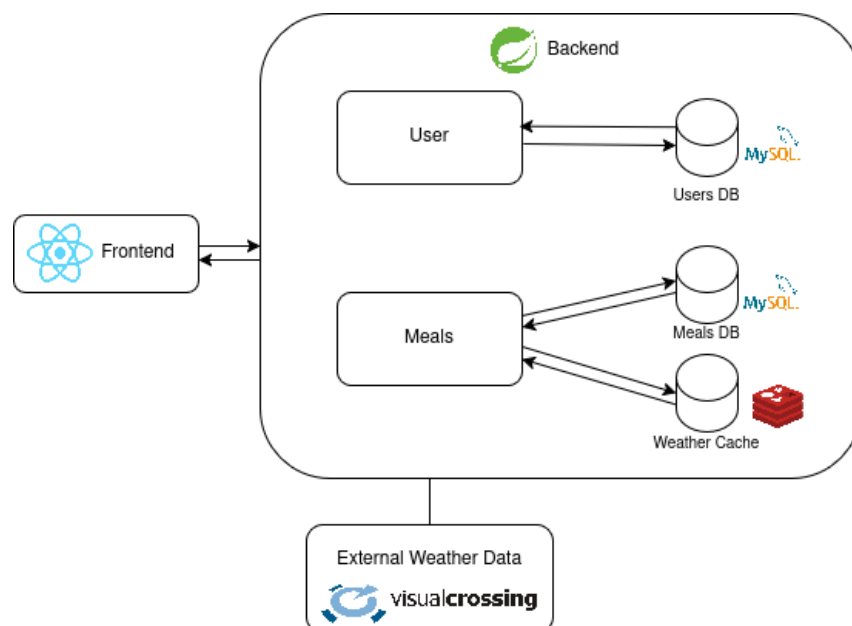
The system also integrates Redis to cache weather data, which enhances performance by reducing the frequency of external API calls. By storing weather data temporarily, Redis optimizes the overall responsiveness of the application, especially during peak usage times.

The entire application is containerized using Docker and Docker Compose. Docker allows the application to be packaged into containers, ensuring that it runs consistently across different environments. Docker Compose is used to manage multiple services—frontend, backend, databases, and Redis.

For testing, the application employs several frameworks and tools. JUnit is used for unit testing the backend services, ensuring that individual components function correctly. Spring Boot Test is used for integration testing, verifying that different parts of the system work together as expected. For API testing, REST Assured is utilized to test the endpoints of the REST API, ensuring that the backend services handle requests and responses correctly. Mockito is used for mocking dependencies in tests, allowing for isolated unit testing. WireMock is employed to simulate the behavior of external services, such as the weather API, for more realistic testing scenarios. Additionally, Jacoco provides code coverage reports, helping ensure that the code is well-tested.

By leveraging SonarQube for continuous integration and static code analysis, the application ensures code quality throughout its development lifecycle. The integration of these testing and analysis tools provides a comprehensive quality assurance strategy, helping to identify potential issues early and ensuring that the system operates efficiently and securely.

In summary, the architecture of the Meal Booking Web Application employs modern technologies to create a robust and scalable system that ensures a seamless experience for students, staff, and administrators alike, all while maintaining high standards for code quality and system performance.




2.3 API for developers

Meals Service Endpoints (<http://localhost:8080/api/swagger-ui/index.html>)

Restaurant Restaurant management APIs			^
GET	/restaurants	Get all restaurants	▼
GET	/restaurants/{id}	Get restaurant by ID	▼
GET	/restaurants/{id}/meals	Get restaurant meals	▼
Meals Meal management APIs			^
GET	/meals	Get all meals	▼
GET	/meals/restaurant/{restaurantId}	Get meals by restaurant	▼
GET	/meals/restaurant/{restaurantId}/category/{category}	Get meals by restaurant and category	▼
GET	/meals/category/{category}	Get meals by category	▼
Reservations Reservation management APIs			^
PUT	/reservations/{id}/status	Update reservation status	▼
GET	/reservations	Get all reservations	▼
POST	/reservations	Create a new reservation	▼
GET	/reservations/{id}	Get reservation by ID	▼
GET	/reservations/token/{token}	Get reservation by token	▼

User Service Endpoints (<http://localhost:8081/api/swagger-ui/index.html>)

Authentication Authentication management APIs			^
POST	/auth/register	User registration	▼ 
POST	/auth/login	User login	▼
GET	/auth/test	Test endpoint	▼
Admin Admin management APIs			^
GET	/admin/users	Get all users	▼
DELETE	/admin/users/{userId}	Delete user	▼

3 Quality assurance

3.1 Overall strategy for testing

The project employs a mixed testing strategy using BDD and TDD. Cucumber, JUnit, Mockito, REST Assured, TestContainers and JMeter are and were used along project development.

3.2 Unit testing

JUnit 5 with Mockito for mocking dependencies.
Tests cover CRUD operations, validation logic, and error cases.
Full coverage of service layer behavior in isolation.

```
@ExtendWith(MockitoExtension.class)
class RestaurantServiceTest {
    @Mock
    private RestaurantRepository restaurantRepository;

    @InjectMocks
    private RestaurantService restaurantService;
}
```

3.3 Integration testing

API Integration: REST Assured with TestContainers.
Tests validate API endpoints with real database connections.
Ensures correct behavior of the complete request-response cycle.

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@Testcontainers
class RestaurantControllerIT {
    @Container
    static MySQLContainer<?> mySQLContainer = new MySQLContainer<>("mysql:8.0.26");
}
```

3.4 Functional testing

BDD with Cucumber: Feature files define user-facing scenarios
Implemented user stories:

- Restaurant management (CRUD operations)
- Reservation creation and management

Step definitions connect scenarios to actual API calls

...

```
Feature: Restaurant Management
    Scenario: Create a new restaurant
```

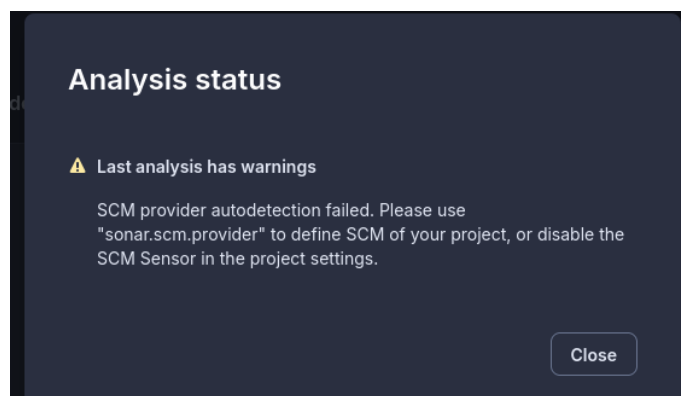
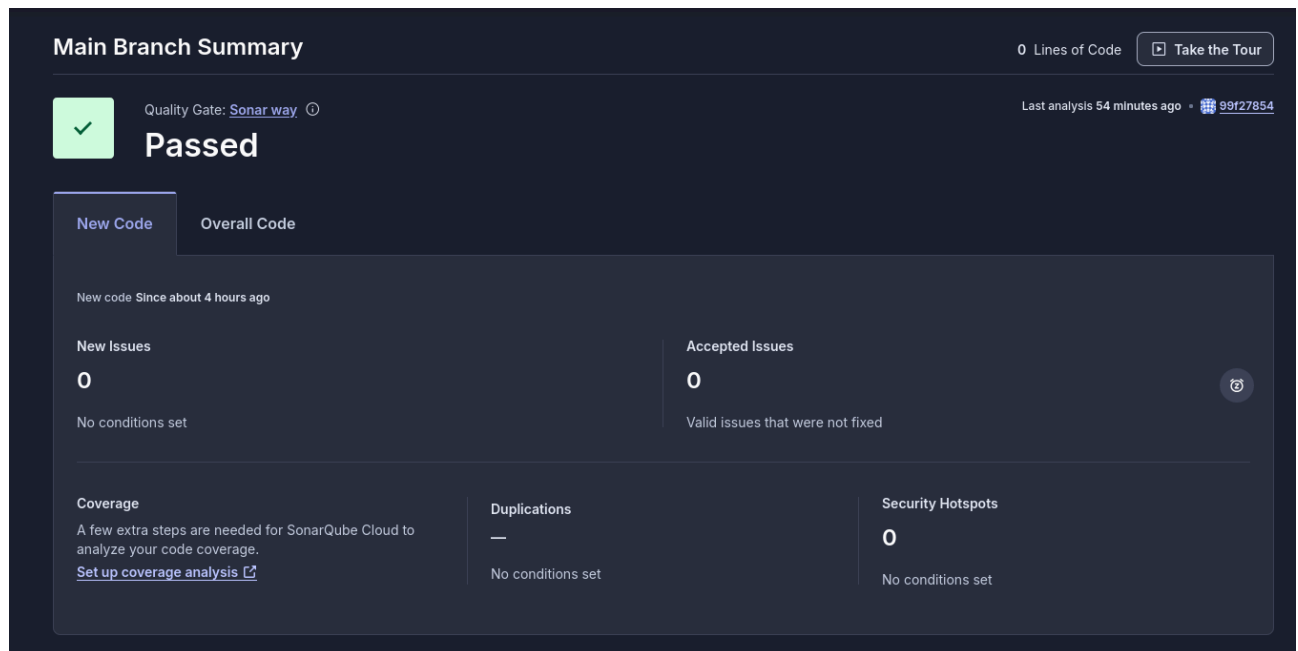
```
When I create a restaurant with the following details:
| name          | Moliceiro Restaurant      |
| description   | Traditional Portuguese food |
Then the restaurant should be created successfully
..
```

3.5 Non functional testing

Performance testing and load testing with k6.
Tests in /test/resources/k6 in each service.

3.6 Code quality analysis

For code quality I have a basic integration with SonarCloud.
Unfortunately I didn't have enough time to really implement SonarQube in my project, it is connected to my rep and there is a workflow (build.yml) and files(sonar-project.properties and) indicating its initial setup.



4 References & resources

Project resources

Resource:	URL/location:
Git repository	https://github.com/roldao04/TQS_113920
Video demo	https://github.com/roldao04/TQS_113920/blob/main/HW1/docs/Demo.webm