HW1: Mid-term assignment report

Gabriel Martins Silva 113786, v2025-04-8

1 Introduction

1.1 Overview of the work

This report presents the midterm individual project developed as part of the TQS (Testing and Quality Software) course. It aims to provide a comprehensive overview of both the functional aspects of the application and the quality assurance strategies employed during its development. The project focuses on delivering a robust software product while applying testing principles and best practices to ensure reliability, maintainability, and user satisfaction.

The software developed is called **Moliceiro Meals**, a full-featured restaurant reservation platform designed to simplify and enhance the dining experience for customers while providing useful tools for restaurant staff. The platform allows users to browse a list of participating restaurants, view detailed profiles including location and contact information, and explore dynamic daily and future menus. Customers can make reservations online by providing their personal details and party size, and they can track their reservations through a clearly defined lifecycle—ranging from pending and confirmed status to check-in and completion. Features such as special request handling and reservation cancellation are also supported.

In addition to its core reservation system, Moliceiro Meals integrates a weather forecasting component that offers intelligent dining recommendations, such as suggesting indoor or outdoor seating based on local conditions. The platform provides two primary user interfaces: a customer-facing portal for browsing and booking, and a staff dashboard that enables restaurant personnel to manage menus and reservations efficiently.

The system architecture follows the widely adopted Spring Boot framework, leveraging the Model-View-Controller (MVC) pattern to ensure a clean separation of concerns. It is organized into clearly defined layers—including controllers, services, repositories, and models—which contributes to the maintainability and scalability of the application.

This report will delve into the design and implementation of Moliceiro Meals, highlight its key features, and explain the testing techniques and quality control processes applied to ensure a high standard of software quality throughout the development lifecycle.

1.2 Current limitations

While the system provides a solid foundation for restaurant discovery, menu browsing, reservation management, and weather integration, there are a few known limitations that represent planned but currently unimplemented features.

One such limitation is the **restaurant image upload functionality**. Although the restaurant listing pages support the display of visual content, the current implementation does not yet allow staff users to upload images for their respective venues. This feature is expected to enhance the overall user experience by making the browsing interface more visually engaging and informative. Its absence does not affect the core functionality but does limit the richness of the restaurant profiles.

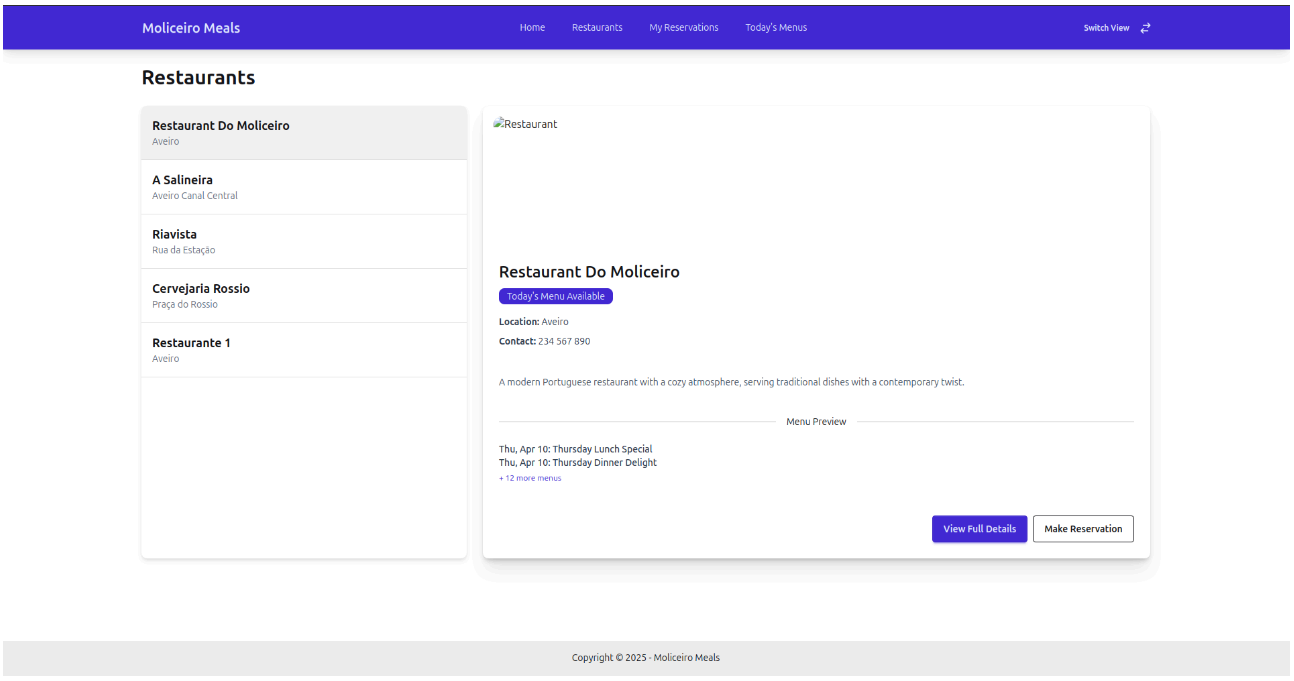
Another anticipated feature that remains under development is **menu recommendations based on daily weather conditions**. While the application already integrates weather forecasts for each restaurant location, this data is not yet used to dynamically suggest specific menu items. The goal is to implement a smart recommendation system that, for example, highlights warm dishes on colder days or promotes lighter options during warm weather - further personalizing the dining experience. For now, weather data is purely informational and not linked to menu logic.

These limitations are well understood and planned for future iterations. Their eventual inclusion will improve user engagement, decision-making, and the overall polish of the application.

2 Product specification

The system has been designed to support two primary user roles: **Customers**, specifically students and professors from Moliceiro University, and **Restaurant Staff**, who are responsible for managing daily operations. Each actor has access to tailored functionalities that align with their needs, ensuring a smooth and intuitive experience across the application.

For **customers**, the journey begins with the ability to **browse restaurants** located in Aveiro. Through a dedicated "Restaurants" page, users can explore a list of available dining options, complete with essential details such as the restaurant's name, location, description, and contact information. Clicking on a restaurant provides further details, allowing users to make informed choices based on their preferences.



Restaurants

Restaurant Do Moliceiro
Aveiro

A Salineira
Aveiro Canal Central

Riavista
Rua da Estação

Cervejaria Rossio
Praça do Rossio

Restaurante 1
Aveiro

Restaurant Do Moliceiro

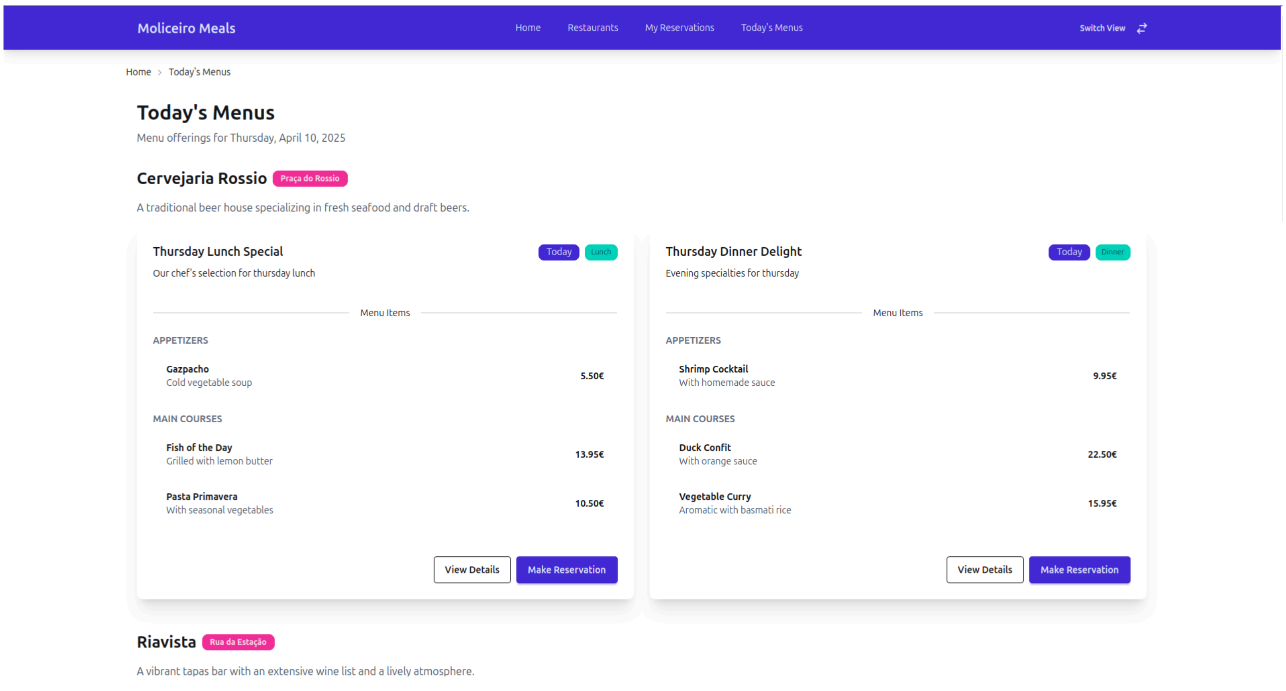
Location: Aveiro
Contact: 234 567 890

A modern Portuguese restaurant with a cozy atmosphere, serving traditional dishes with a contemporary twist.

Thu, Apr 10: Thursday Lunch Special
Thu, Apr 10: Thursday Dinner Delight
+ 12 more menus

Copyright © 2025 - Moliceiro Meals

Once a restaurant is selected, customers can proceed to **view the daily or upcoming menus**. The menu interface displays detailed descriptions of available dishes, including prices and categories, helping users understand their options for the current or future dining days.


Moliceiro Meals | Home | Restaurants | My Reservations | Today's Menus | Switch View

Home > Today's Menus

Today's Menus

Menu offerings for Thursday, April 10, 2025

Cervejaria Rossio — Praça do Rossio

A traditional beer house specializing in fresh seafood and draft beers.

Thursday Lunch Special — Today — Lunch

Our chef's selection for thursday lunch

Menu Items

APPETIZERS

Gazpacho Cold vegetable soup	5.50€

MAIN COURSES

Fish of the Day Grilled with lemon butter	13.95€
Pasta Primavera With seasonal vegetables	10.50€

View Details | Make Reservation

Thursday Dinner Delight — Today — Dinner

Evening specialties for thursday

Menu Items

APPETIZERS

Shrimp Cocktail With homemade sauce	9.95€

MAIN COURSES

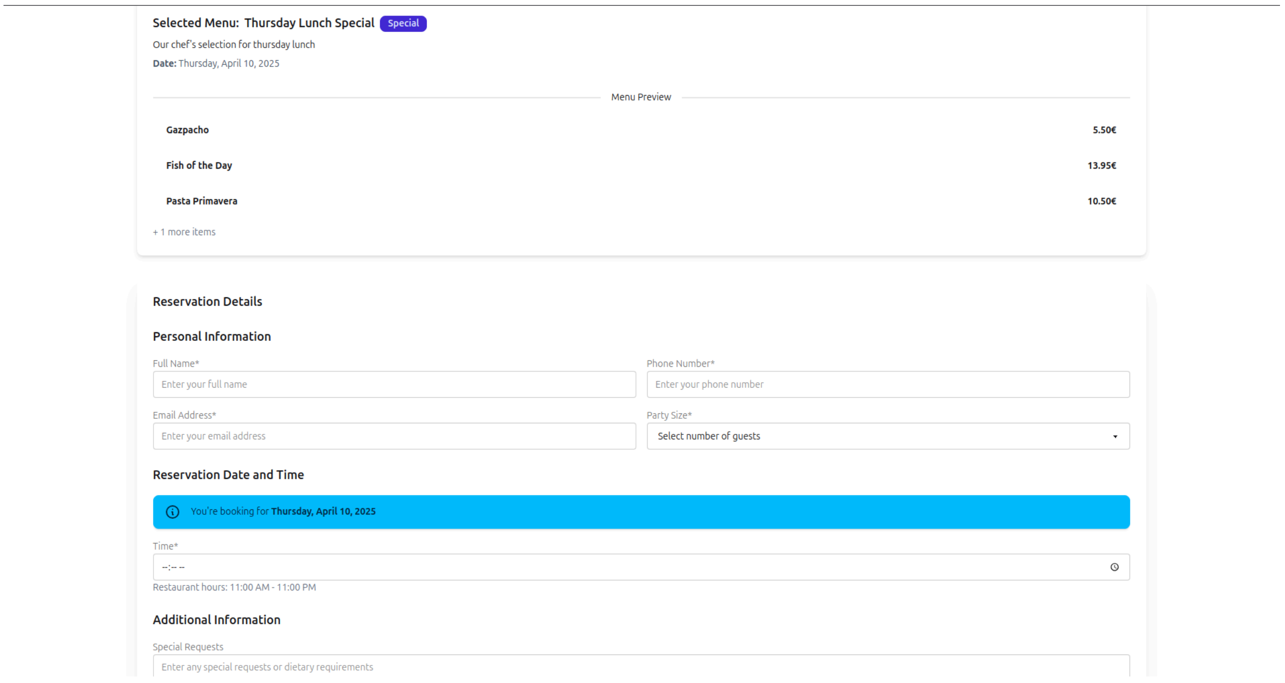
Duck Confit With orange sauce	22.50€
Vegetable Curry Aromatic with basmati rice	15.95€

View Details | Make Reservation

Riavista — Rua da Estação

A vibrant tapas bar with an extensive wine list and a lively atmosphere.

A central feature of the application is its **reservation system**, which allows customers to book a table at their chosen restaurant. The process involves selecting a restaurant and preferred time and date, entering basic details like name, email, and party size, and submitting the reservation. Once completed, users receive confirmation and are also able to view or cancel their existing reservations through a simple interface.


Selected Menu: Thursday Lunch Special — Special

Our chef's selection for thursday lunch

Date: Thursday, April 10, 2025

Menu Preview

Gazpacho	5.50€
Fish of the Day	13.95€
Pasta Primavera	10.50€

+ 1 more items

Reservation Details

Personal Information

Full Name*: Enter your full name	Phone Number*: Enter your phone number
Email Address*: Enter your email address	Party Size*: Select number of guests

Reservation Date and Time

You're booking for **Thursday, April 10, 2025**

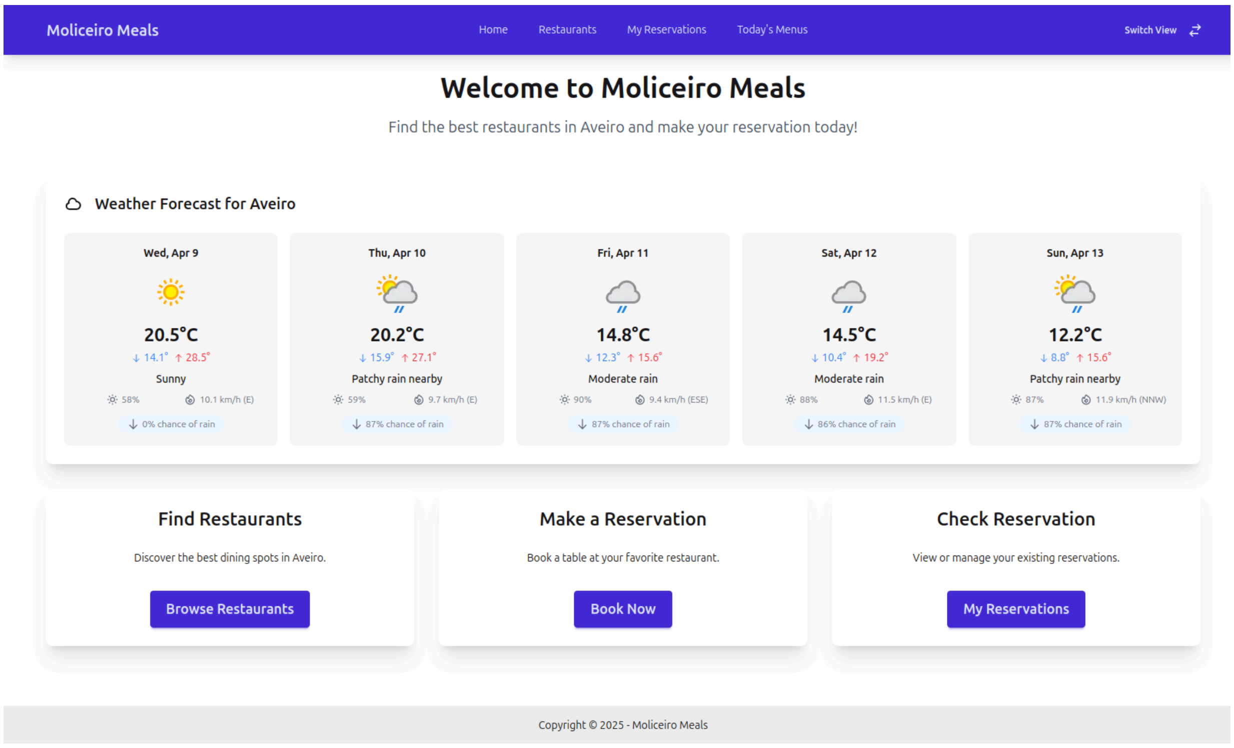
Time*: --:-- --

Restaurant hours: 11:00 AM - 11:00 PM

Additional Information

Special Requests: Enter any special requests or dietary requirements

An additional feature designed to enhance the dining experience is the **weather integration**. Customers are able to see real-time and forecasted weather conditions specific to the restaurant's location. This functionality helps users decide whether to dine indoors or outdoors, depending on the weather suitability for the selected day and time.



Welcome to Moliceiro Meals

Find the best restaurants in Aveiro and make your reservation today!

Weather Forecast for Aveiro

Wed, Apr 9	Thu, Apr 10	Fri, Apr 11	Sat, Apr 12	Sun, Apr 13
20.5°C	20.2°C	14.8°C	14.5°C	12.2°C
↓ 14.1° ↑ 28.5°	↓ 15.9° ↑ 27.1°	↓ 12.3° ↑ 15.6°	↓ 10.4° ↑ 19.2°	↓ 8.8° ↑ 15.6°
Sunny	Patchy rain nearby	Moderate rain	Moderate rain	Patchy rain nearby
58%	59%	90%	88%	87%
10.1 km/h (E)	9.7 km/h (E)	9.4 km/h (ESE)	11.5 km/h (E)	11.9 km/h (NNW)
0% chance of rain	87% chance of rain	87% chance of rain	86% chance of rain	87% chance of rain

Find Restaurants

Discover the best dining spots in Aveiro.

Browse Restaurants

Make a Reservation

Book a table at your favorite restaurant.

Book Now

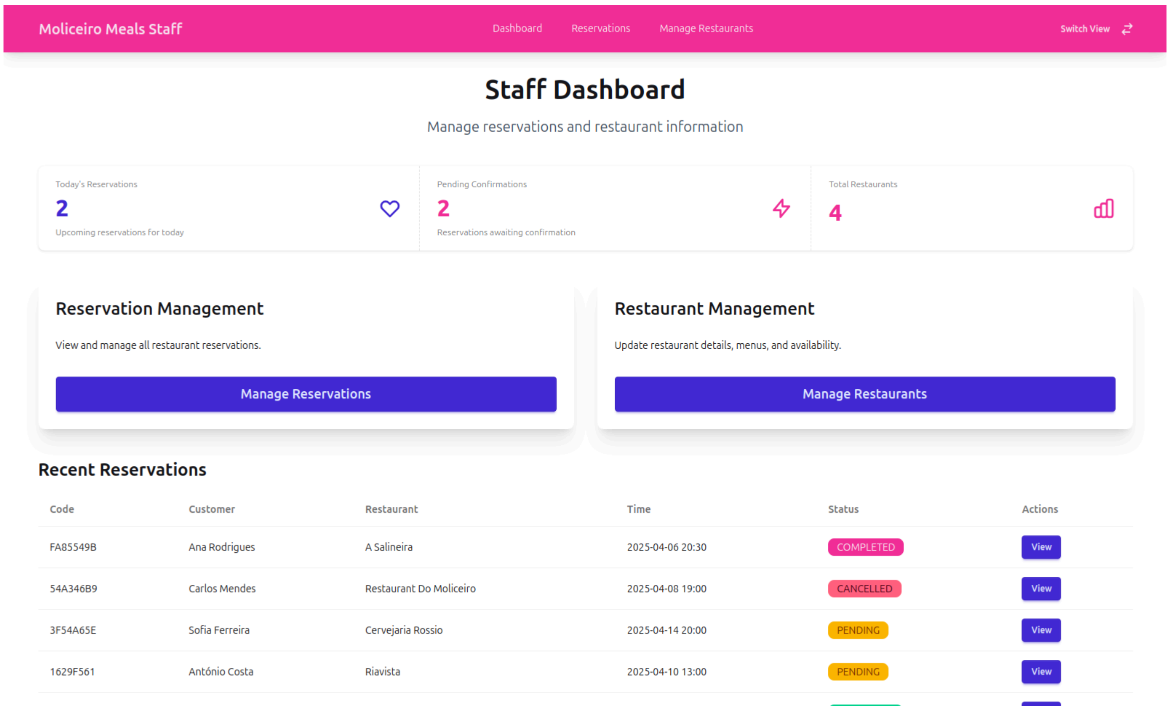
Check Reservation

View or manage your existing reservations.

My Reservations

Copyright © 2025 - Moliceiro Meals

On the other side, **staff members** are provided with tools to manage operations through a **dedicated dashboard**. From this interface, staff can monitor key statistics such as the day's reservations and current menu configurations. The dashboard serves as the operational hub for daily restaurant management.



Staff Dashboard

Manage reservations and restaurant information

Today's Reservations
2
Upcoming reservations for today

Pending Confirmations
2
Reservations awaiting confirmation

Total Restaurants
4

Reservation Management

View and manage all restaurant reservations.

Manage Reservations

Restaurant Management

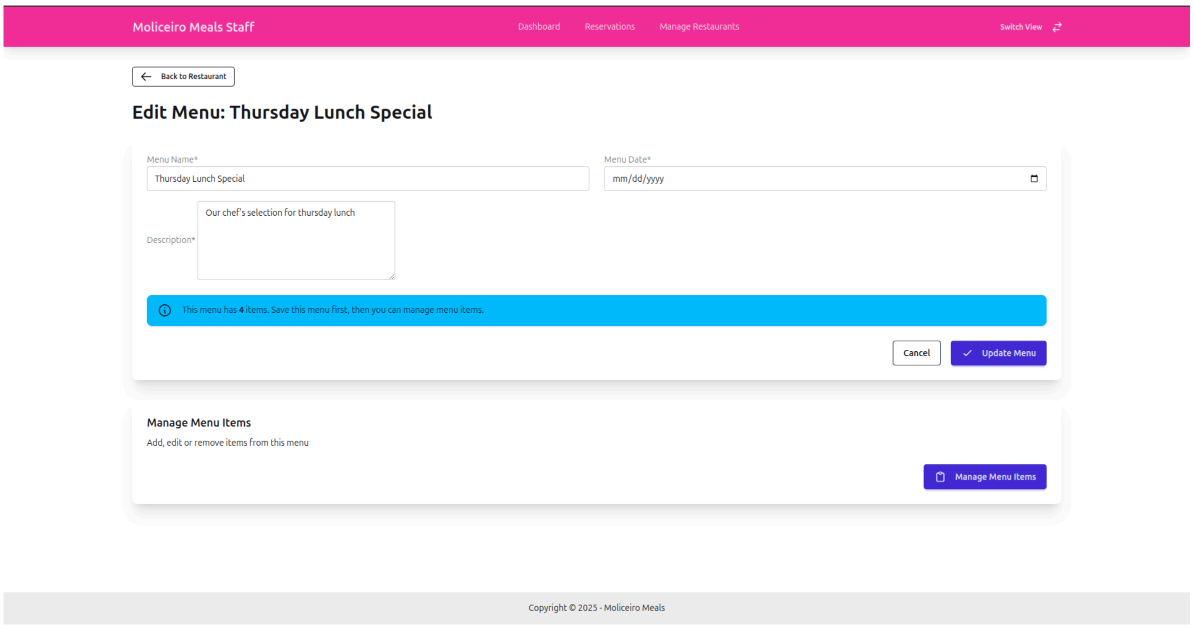
Update restaurant details, menus, and availability.

Manage Restaurants

Recent Reservations

Code	Customer	Restaurant	Time	Status	Actions
FA85549B	Ana Rodrigues	A Salineira	2025-04-06 20:30	COMPLETED	View
54A346B9	Carlos Mendes	Restaurant Do Moliceiro	2025-04-08 19:00	CANCELLED	View
3F54A65E	Sofia Ferreira	Cervejaria Rossio	2025-04-14 20:00	PENDING	View
1629F561	António Costa	Riavista	2025-04-10 13:00	PENDING	View

Staff are also given access to **menu management tools**, allowing them to create new menus, update existing ones, and remove outdated entries. They can add or edit individual menu items—defining names, descriptions, prices, and categories—as well as associate them with specific menus and dates.



Edit Menu: Thursday Lunch Special

Menu Name* Thursday Lunch Special

Menu Date* mm/dd/yyyy

Description* Our chef's selection for thursday lunch

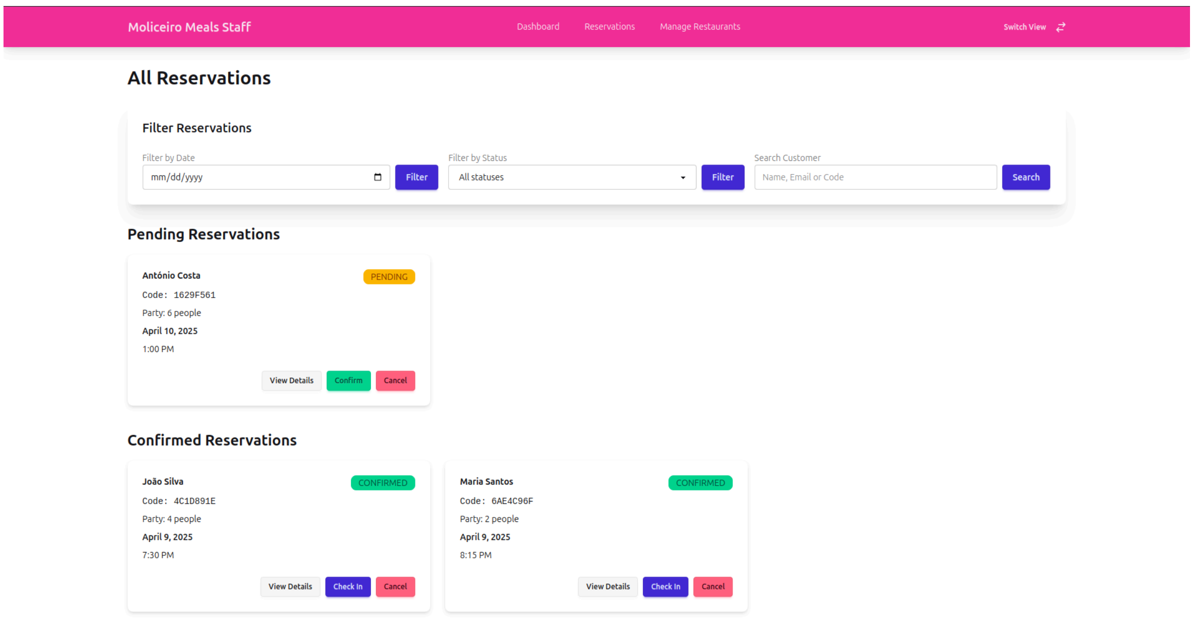
This menu has **4** items. Save this menu first, then you can manage menu items.

Manage Menu Items

Add, edit or remove items from this menu

Copyright © 2025 - Moliceiro Meals

In terms of reservation handling, staff can **view, update, and manage all customer bookings**. They are able to confirm pending reservations, cancel bookings when necessary, and access an overview of the day's reservation schedule. This enables better planning and communication between the dining team and their guests.



All Reservations

Filter Reservations

Filter by Date mm/dd/yyyy

Filter by Status All statuses

Search Customer Name, Email or Code

Pending Reservations

António Costa — PENDING
Code: 1629F561
Party: 6 people
April 10, 2025
1:00 PM

Confirmed Reservations

João Silva — CONFIRMED
Code: 4C1D891E
Party: 4 people
April 9, 2025
7:30 PM

Maria Santos — CONFIRMED
Code: 6AE4C96F
Party: 2 people
April 9, 2025
8:15 PM

Together, these functionalities ensure that both customer-facing and staff-facing experiences are complete, intuitive, and well-supported. Each interaction is grounded in real user needs, facilitating smooth restaurant operations and a reliable dining experience. Screenshots of relevant pages - such as the reservation form, menu view, and staff dashboard - can be included to visually illustrate the flow of interactions within the application.

2.1 System implementation architecture

The application is built using a layered architecture that separates concerns across different parts of the system. At the top, the presentation layer handles incoming web requests and renders views using Thymeleaf. Business logic is encapsulated within the service layer, which also manages interactions with external APIs. Data persistence is managed through Spring Data JPA, allowing seamless integration with both in-memory and relational databases. Technical concerns such as caching, monitoring, and application configuration are handled within the infrastructure layer.

The core technology stack includes Java 21 as the programming language, with Spring Boot serving as the foundational framework. Spring Data JPA simplifies database interactions, while Thymeleaf is used for server-side rendering of frontend templates. For data storage, the system can be configured to use H2 in-memory databases during development and switch to MySQL in production. Docker is used to containerize the application, enabling consistent deployment across different environments.

Testing is well-covered using JUnit 5 for unit and integration tests, with MockMvc used specifically for testing controllers. Selenium is also configured to support UI testing. To maintain code quality, the project integrates with SonarCloud for static analysis. CI/CD pipelines are implemented using GitHub Actions, automating the build, test, and deployment processes.

The system supports both local development and containerized deployment scenarios. It is designed to be flexible, with the ability to switch databases depending on the environment, which makes setup and transition between development and production smooth.

Several technical features stand out in the implementation. Caching is implemented using Spring's abstraction, with a configurable TTL to control how long data is stored in the cache. An external weather API is integrated into the application and benefits from this caching layer to reduce unnecessary requests. On startup, the application can preload sample data to streamline testing and development. Additionally, monitoring is facilitated through Spring Actuator, which exposes various endpoints for metrics and health checks. The API is documented using OpenAPI/Swagger, making it easy for developers to explore and test endpoints interactively.

2.2 API for developers

The application provides a set of well-defined services through a restful API, enabling developers to interact with the system both in terms of its core business functionality and internal monitoring. These services are organized into two primary categories: meal-related operations and cache usage statistics.

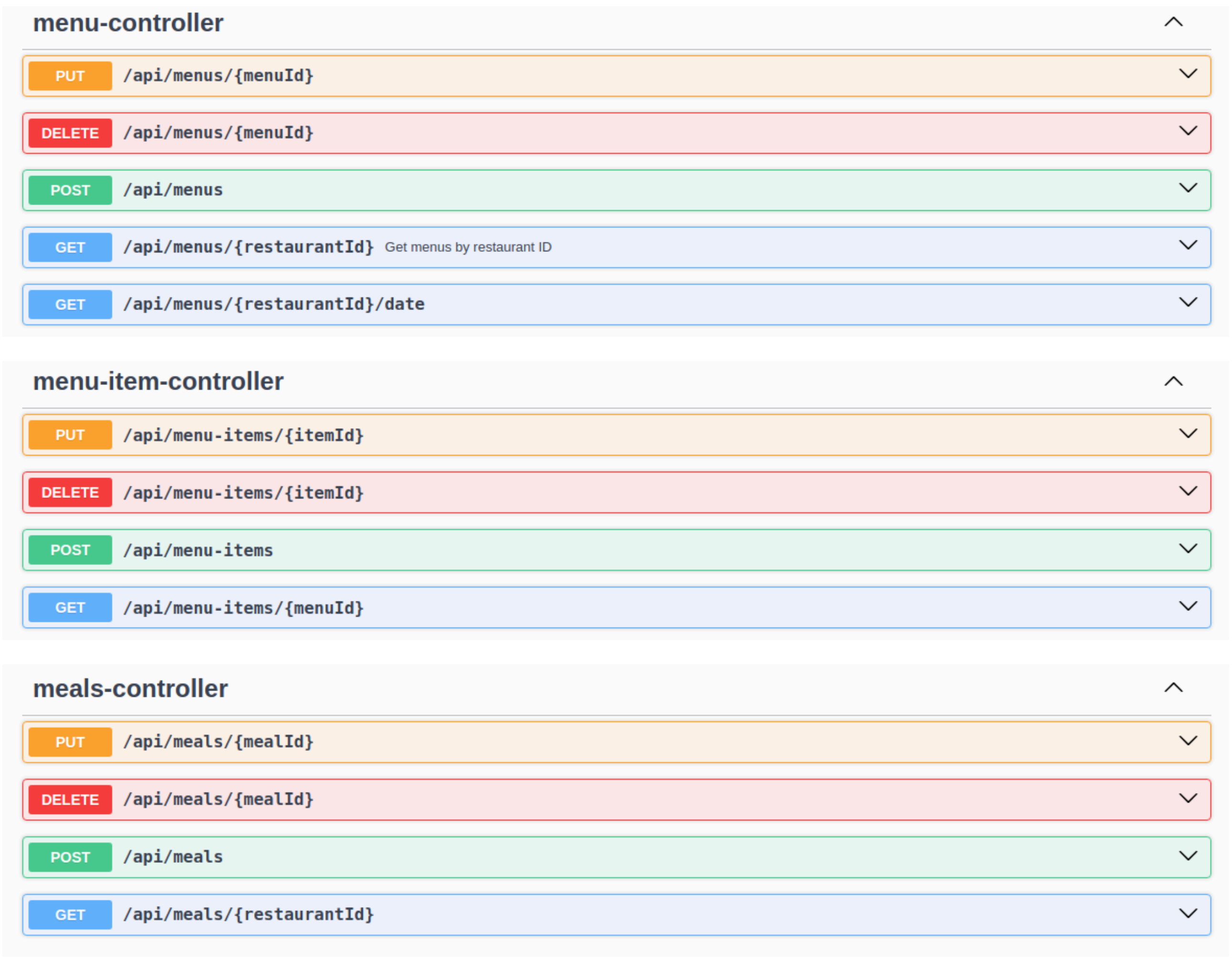
In the problem domain, the system exposes endpoints that allow developers to list available meal options, make meal reservations, and check reservation details. For example, the **`/meals`** endpoint returns a list of all available meals, which can be used to populate interfaces where users choose what they want to book. Once a selection is made, a reservation can be created via the **`/reservations`** endpoint, which handles the booking process. Each reservation is assigned an

ID, allowing the client to retrieve reservation details using **`/reservations/{id}`**. This structure supports a typical flow in which a user views meal options, makes a selection, and then receives confirmation or a “ticket” for their reservation.

Complementing these core services is a set of endpoints focused on system introspection, specifically cache performance. Since the application uses Spring’s caching abstraction - particularly for integrating with external services like a weather API - it is important for developers and system operators to monitor cache efficiency. This is achieved through Spring Boot Actuator, which provides metrics endpoints such as **`/actuator/caches`** to inspect overall cache behavior. More detailed statistics, including cache hits and misses, are available through metric-specific endpoints like **`/cache/stats`**, and **`/cache/contents`**. These endpoints provide real-time insights into the effectiveness of the caching strategy, helping to guide configuration decisions.

To facilitate interaction with these APIs, the application includes integrated OpenAPI/Swagger documentation. This enables developers to explore, test, and understand each endpoint’s input and output formats through a user-friendly web interface, typically accessible via **`/swagger-ui.html`** when the application is running.

Together, these features make the project both functionally complete for its meal reservation purpose and operationally transparent through its built-in monitoring capabilities.



menu-controller

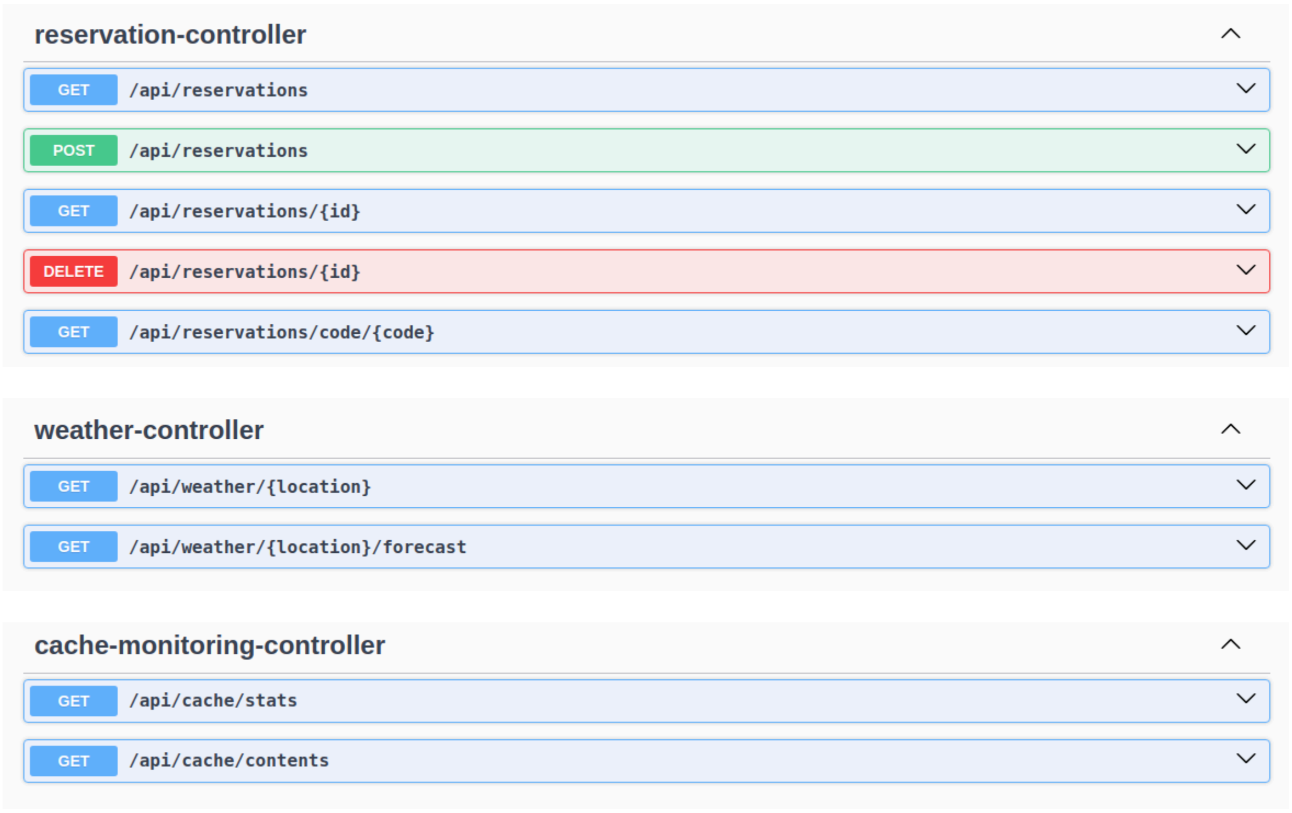
- PUT `/api/menus/{menuId}`
- DELETE `/api/menus/{menuId}`
- POST `/api/menus`
- GET `/api/menus/{restaurantId}` Get menus by restaurant ID
- GET `/api/menus/{restaurantId}/date`

menu-item-controller

- PUT `/api/menu-items/{itemId}`
- DELETE `/api/menu-items/{itemId}`
- POST `/api/menu-items`
- GET `/api/menu-items/{menuId}`

meals-controller

- PUT `/api/meals/{mealId}`
- DELETE `/api/meals/{mealId}`
- POST `/api/meals`
- GET `/api/meals/{restaurantId}`


reservation-controller

GET /api/reservations

POST /api/reservations

GET /api/reservations/{id}

DELETE /api/reservations/{id}

GET /api/reservations/code/{code}

weather-controller

GET /api/weather/{location}

GET /api/weather/{location}/forecast

cache-monitoring-controller

GET /api/cache/stats

GET /api/cache/contents

3 Quality assurance

3.1 Overall strategy for testing

The application follows a thoughtful and layered testing strategy that reflects best practices in modern software development. Rather than relying on a single methodology, the testing approach combines several complementary techniques and tools to validate the system's behavior across different levels of abstraction.

A multi-layered testing architecture is at the core of this strategy. Tests are organized into clearly defined directories - such as unit, integration, end-to-end, web, and controller - highlighting the project's commitment to the separation of concerns. Each test layer targets a specific scope: unit tests validate isolated logic, integration tests ensure cooperation between components, and end-to-end tests cover real-world application scenarios from the user's perspective.

The project prioritizes test isolation to ensure clarity and reliability. For example, controller tests use **`@WebMvcTest`** annotations to load only the web layer, while other components are mocked to avoid unintended dependencies. Mockito is used extensively across the project to mock services and repositories where appropriate, allowing developers to write targeted, behavior-driven tests that remain fast and predictable.

Container-based testing is another strong aspect of the strategy. Integration tests involving the database use **Testcontainers**, as defined in **`TestcontainersConfiguration.java`**, to spin up temporary MySQL instances during test runs. This ensures that tests closely mirror the production environment without the need for manual database setup, making integration tests more robust and environment-agnostic.

The project also makes good use of Spring Boot's testing framework, leveraging annotations and utilities that allow tests to run within the Spring context when needed. This includes automatic configuration loading, dependency injection, and simplified bootstrapping of application components for testing.

Rather than strictly following a Test-Driven Development (TDD) or Behavior-Driven Development (BDD) paradigm, the testing strategy appears pragmatic and balanced. While tools like Cucumber were not adopted, the mix of unit tests, rest assured integration tests for external services, and the use of Testcontainers demonstrates a strong commitment to reliability, maintainability, and real-world validation.

Together, these testing practices form a comprehensive strategy that ensures each part of the system - from a simple utility function to a full database-backed workflow - is validated under realistic and reproducible conditions.

3.2 Unit and integration testing

The application adopts a structured testing approach that incorporates both unit and integration testing to ensure robustness and maintainability. This strategy is designed to validate individual components in isolation, as well as their interactions across different layers of the system.

Unit testing primarily targets the service layer, where the core business logic resides. These tests use the Mockito framework to isolate the service classes from their dependencies, such as repositories or external APIs. This isolation ensures that tests remain focused on verifying the behavior and correctness of the logic without interference from other components. These tests confirm the correctness of data queries and mappings. Utility and helper classes are also covered through standard unit tests, ensuring that small, reusable methods function reliably across different inputs.

Integration testing complements the unit tests by verifying how different layers of the application work together. These tests assess the interactions between controllers and services to confirm that incoming requests are handled correctly and that responses are generated as expected. Similarly, service-to-repository integration is tested to ensure that the application's business logic is effectively backed by reliable data access operations.

A significant part of the integration testing effort focuses on external API communication. The application integrates with a weather service, and this functionality is tested using the rest assured library. These tests validate the correct handling of API responses and ensure that fallback mechanisms behave as intended in case of failure or unexpected data formats.

By combining focused unit tests with comprehensive integration testing, the project ensures a high level of code quality and confidence in system behavior, both in isolation and in collaboration.

```java
class MealTest {

   @Test

   void testConstructorAndGetters() {

       Restaurant restaurant = new Restaurant();

              Meal meal = new Meal("Steak", "Delicious grilled steak", new
BigDecimal("15.99"), restaurant);

       assertEquals("Steak", meal.getName());

       assertEquals("Delicious grilled steak", meal.getDescription());

       assertEquals(new BigDecimal("15.99"), meal.getPrice());

       assertEquals(restaurant, meal.getRestaurant());

   }
```

3.3 Functional testing

The functional testing strategy of the application is centered on validating user-facing features by simulating realistic interactions with the system's rest controllers. These tests focus on ensuring that key business workflows behave correctly under a variety of input scenarios, mirroring the kinds of operations a user would perform through the interface.

Testing is organized around core features, with dedicated test classes for each major domain. For example, the **`MenuControllerTest`** and **`MenuItemControllerTest`** verify the functionality of menu and item management. These include full CRUD operations - creating, updating, retrieving, and deleting menus and individual items. Tests cover common user actions, such as adding a new dish to a restaurant's menu or updating existing descriptions and pricing.

Reservation functionality is covered in depth through the **`ReservationControllerTest`**, which simulates the process of creating a reservation, viewing reservation details, updating reservation times or guest counts, and deleting reservations. These tests also include validation logic, such as checking that a valid restaurant ID is supplied when making a booking.

The **`RestaurantControllerTest`** ensures that users can browse available restaurants, retrieving relevant information for each location. These tests confirm the retrieval logic and ensure that restaurant data is returned in a consistent, usable format.

Another notable area of functional testing is the weather service integration, which is validated through the **`WeatherControllerTest`**. This set of tests ensures that the system can retrieve current or forecasted weather data, which is intended to support features like recommending outdoor dining when weather conditions are favorable. The tests check for correct handling of API responses and fallback behavior in the event of API errors or timeouts.

Together, these tests cover critical user flows, including creating reservations and validating associated restaurant data, updating and deleting existing reservations, managing restaurant menus and their individual items, retrieving restaurant-specific menus for display and accessing weather forecasts for user-facing features such as dining recommendations.

This functional testing approach ensures that key application flows are not only implemented correctly but also remain stable and user-focused throughout ongoing development.

```java
@Test
void testDeleteMenu() throws Exception {
    Long menuId = 1L;

    doNothing().when(menuService).deleteMenu(menuId);

    mockMvc.perform(delete("/api/menus/{menuId}", menuId))
            .andExpect(status().isOk());

    verify(menuService, times(1)).deleteMenu(menuId);
}
```

3.4 Non functional testing

In addition to validating core functionality, the project includes a robust set of non-functional tests aimed at ensuring performance, reliability, and correctness under various operational conditions. These tests cover aspects such as caching behavior, data validation, error handling, and configuration flexibility.

One key area of focus is performance optimization through caching. The weather service integration includes targeted tests in the **`WeatherServiceTest`** class to verify caching effectiveness. These tests compare the behavior of cached responses against fresh API calls, confirming that repeated requests for the same weather data return quickly without unnecessarily invoking the external API. The caching strategy is further validated by testing expiration policies, ensuring that stale data is correctly invalidated and refreshed when needed.

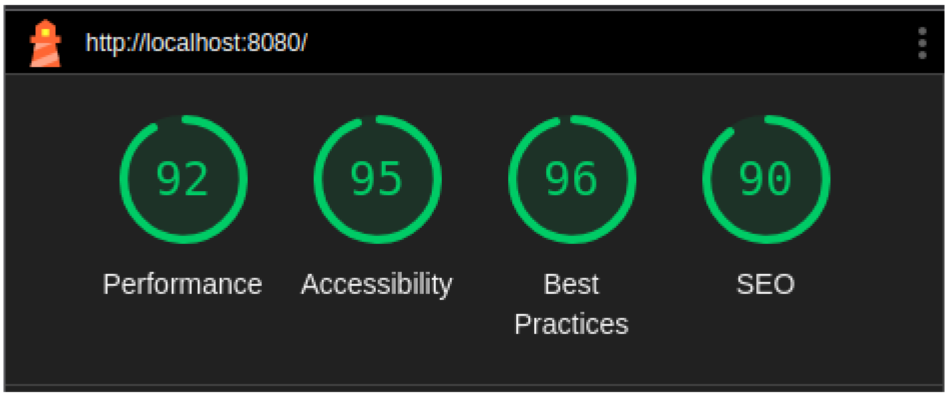
Data validation is also thoroughly tested to enforce business rules and prevent invalid data from entering the system. For example, **`MenuItemTest`** includes explicit tests to ensure that prices cannot be negative, while **`MenuServiceTest`** verifies that all required fields - such as menu names and associated dates - are properly enforced. These validations help maintain data integrity across the application.

Error handling has been rigorously tested as well, ensuring that the application responds appropriately to a variety of failure scenarios. This includes tests for invalid inputs, such as attempts to create menu items with negative pricing, and tests for resource-not-found situations, like querying a menu by date when no matching record exists. Additionally, the weather service tests include cases where the external API is unavailable or returns an error, validating the system's ability to respond with clear and meaningful error messages.

The application also includes configuration testing to validate its behavior in different runtime environments. By using Testcontainers, the project is tested against multiple database configurations, ensuring compatibility and reliability whether using H2 in development or MySQL in production. Cache configurations are also tested using a custom **`TestCacheConfig.java`** setup to confirm that the cache is initialized and behaves as expected under different scenarios.

Complementing these backend tests is a performance analysis of the web interface using **Lighthouse**, Google's automated tool for auditing web applications. The Lighthouse report indicates strong scores across critical metrics such as **Performance**, **Accessibility**, and **Best Practices**. This suggests that the application delivers a fast and accessible user experience, adheres to frontend development standards, and avoids common pitfalls that could hinder usability or SEO. These results validate frontend responsiveness and confirm that user-facing components load efficiently and operate smoothly, even under varying network or device conditions.

Although a formal load or performance test suite is not detailed here, the cache behavior tests act as a foundational step toward performance optimization. These non-functional tests collectively ensure that the system is not only functionally complete but also reliable, performant, and resilient under real-world conditions.



3.5 Code quality analysis

A key part of maintaining the long-term health and scalability of the application was the adoption of static code analysis through **SonarQube**. This tool was integrated into the development workflow to continuously assess code quality, flag potential issues early, and support informed refactoring decisions. The analysis revealed both strengths and areas for improvement, particularly around maintainability and test coverage.

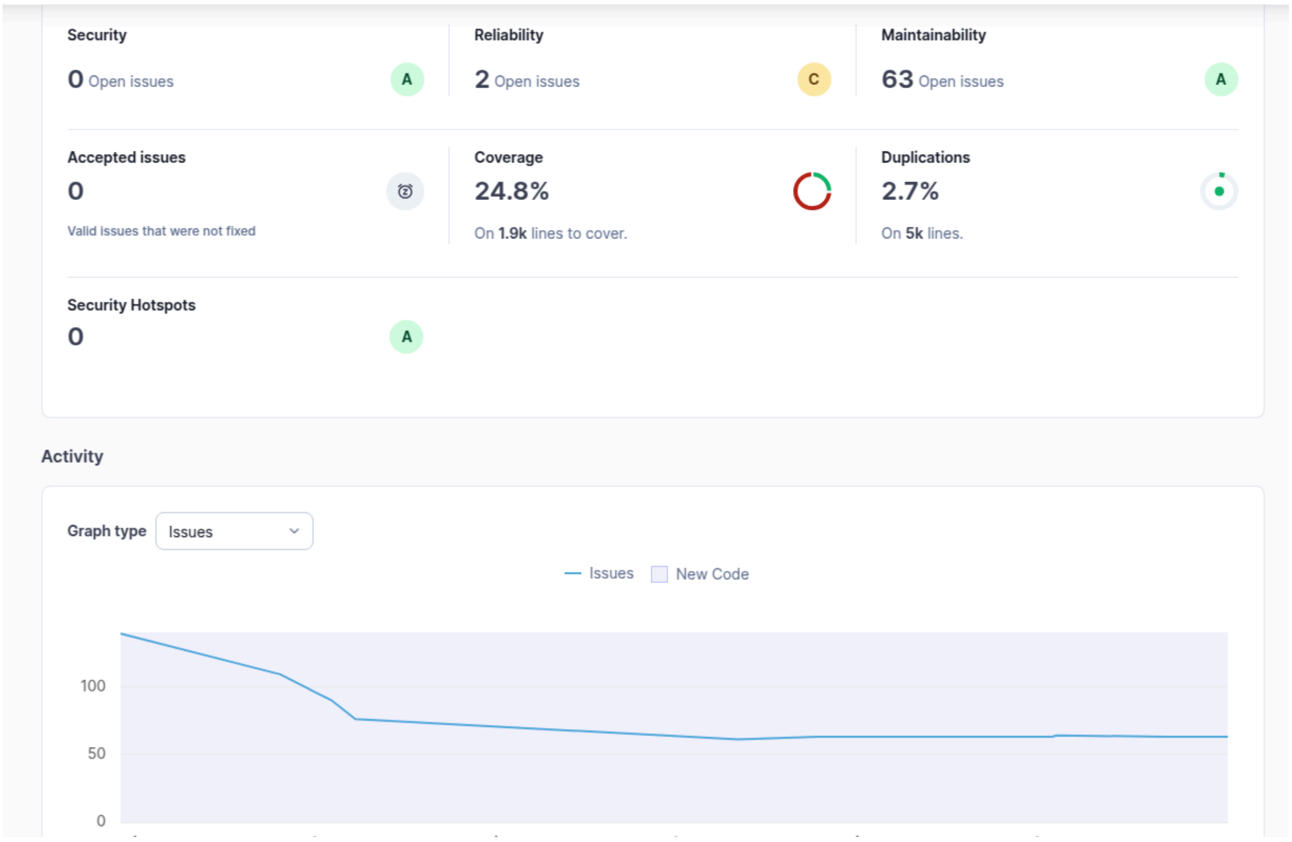
The SonarQube dashboard provided several critical quality metrics. **Security** scored an *A*, with no issues or hotspots detected, indicating strong handling of sensitive operations and external data. **Reliability** received a *C*, due to two minor issues that, while not critical, warranted attention. The **Maintainability** score was an *A*, despite identifying 63 code smells - none severe, but enough to justify focused cleanup. **Coverage** stood at 24.8%, a known limitation resulting from a decision to prioritize business logic testing over frontend controllers. **Duplications** were minimal at 2.7%, demonstrating consistency in code reuse.

Several valuable insights emerged from this analysis. One key issue was the identification of **potential null handling problems** within the **`WeatherService`** class. In some instances, API responses were being accessed without sufficient validation, exposing the system to possible **`NullPointerExceptions`**. Another finding was the presence of **methods with high cyclomatic complexity**, particularly within the **`ReservationService`**, where the reservation validation logic had grown too dense and difficult to maintain. Additionally, SonarQube detected **redundant validation logic** in both **`MenuService`** and **`RestaurantService`**, which had gone unnoticed during development. This redundancy was subsequently resolved by refactoring the logic into a shared utility class, improving maintainability and reducing duplication.

In response to these findings, several improvements were implemented. Exception handling was enhanced throughout the application, particularly in areas involving external API communication. Complex methods were simplified and broken down where possible to improve readability and testability. A shared **`ValidationUtils`** class was created to standardize and consolidate validation logic across services. Comprehensive null checks were also introduced to prevent runtime issues and improve robustness.

Despite the benefits, some challenges and trade-offs were encountered. **False positives** occasionally appeared in the analysis, particularly where deliberate design decisions diverged from SonarQube's general recommendations. These cases were addressed by suppressing specific warnings with justification. The **low test coverage** metric was a conscious trade-off: test development was focused on critical business logic and integrations rather than UI endpoints, which naturally lowered the overall percentage. Additionally, some **modernization suggestions**, such as converting loops to Java Streams, were deferred to preserve consistency with the existing codebase and avoid introducing style fragmentation.

In conclusion, SonarQube proved invaluable in identifying subtle and potentially critical issues that might have been overlooked in manual reviews. The tool not only supported better engineering practices but also played a key role in reducing technical debt. Looking ahead, earlier integration of SonarQube quality gates and improved test coverage for frontend components would further strengthen code reliability. However, even at this stage, the project's core business logic demonstrates a high standard of quality with minimal smells and solid coverage.



Security
0 Open issues — A

Reliability
2 Open issues — C

Maintainability
63 Open issues — A

Accepted issues
0
Valid issues that were not fixed

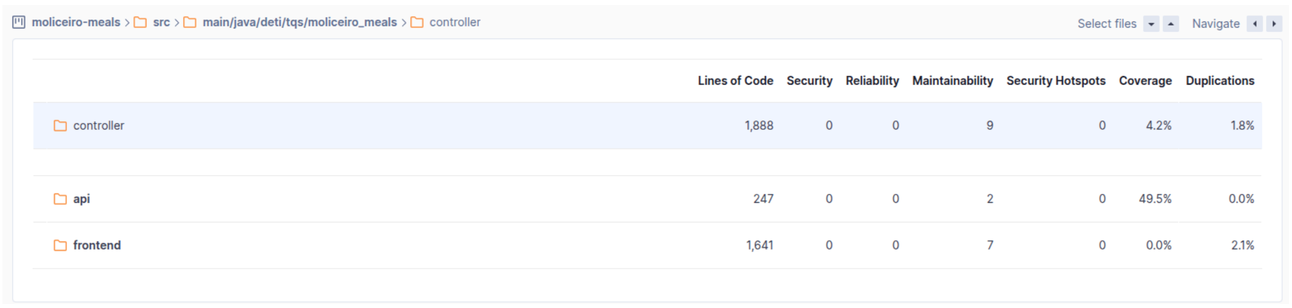
Coverage
24.8%
On 1.9k lines to cover.

Duplications
2.7%
On 5k lines.

Security Hotspots
0 — A

Activity

Graph type: Issues

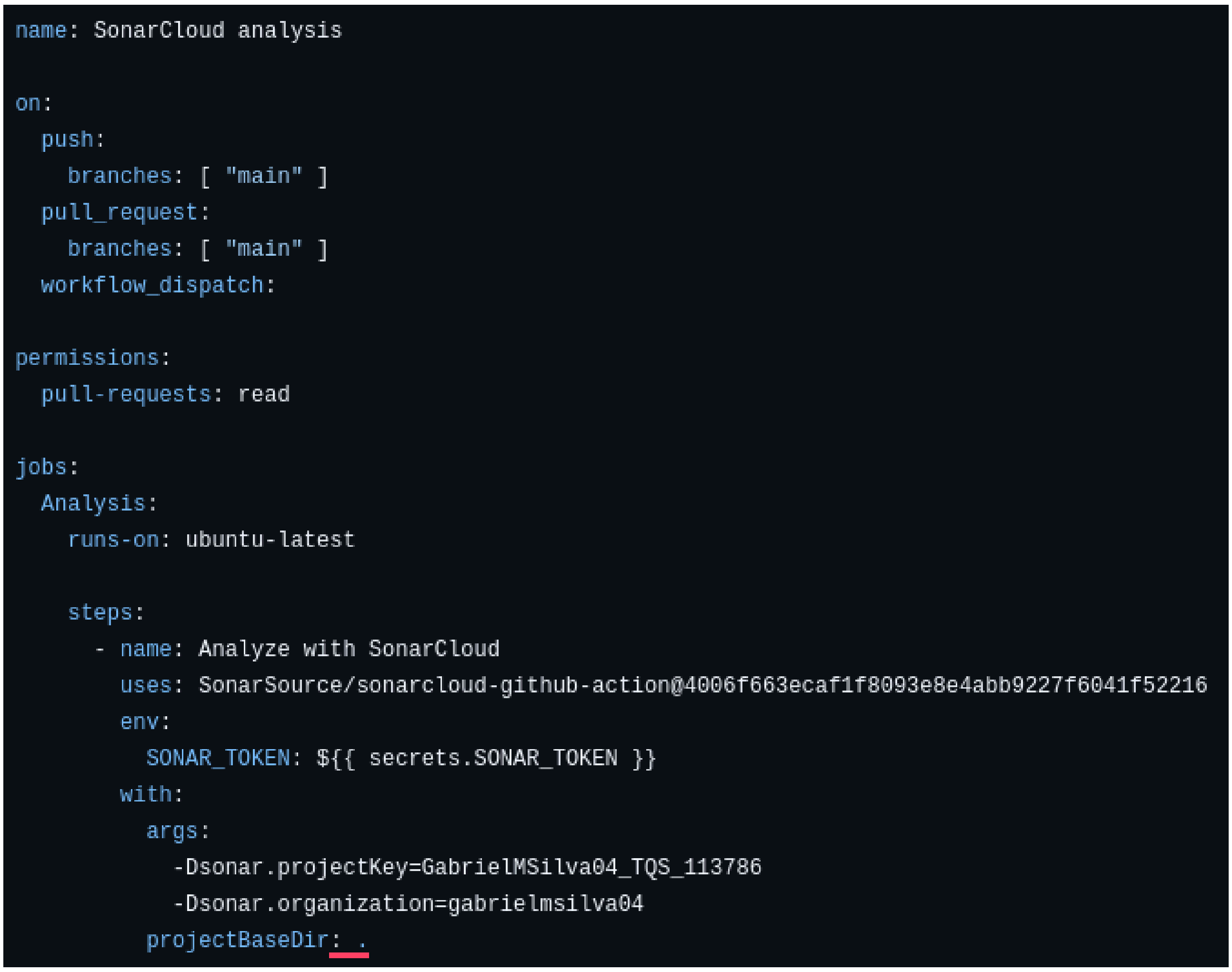


moliceiro-meals > src > main/java/deti/tqs/moliceiro_meals > controller

	Lines of Code	Security	Reliability	Maintainability	Security Hotspots	Coverage	Duplications
controller	1,888	0	0	9	0	4.2%	1.8%
api	247	0	0	2	0	49.5%	0.0%
frontend	1,641	0	0	7	0	0.0%	2.1%

3.6 Continuous integration pipeline

SonarCloud Github Action



```yaml
name: SonarCloud analysis

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
  workflow_dispatch:

permissions:
  pull-requests: read

jobs:
  Analysis:
    runs-on: ubuntu-latest

    steps:
      - name: Analyze with SonarCloud
        uses: SonarSource/sonarcloud-github-action@4006f663ecaf1f8093e8e4abb9227f6041f52216
        env:
          SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
        with:
          args:
            -Dsonar.projectKey=GabrielMSilva04_TQS_113786
            -Dsonar.organization=gabrielmsilva04
          projectBaseDir: .
```

4 References & resources

Project resources

Resource:	**URL/location:**
Git repository	https://github.com/GabrielMSilva04/TQS_113786
Video demo	https://github.com/GabrielMSilva04/TQS_113786/tree/main/HW1/docs/videos

Reference materials

Weather API:
https://www.weatherapi.com/