HW1: Mid-term assignment report

*Marta Dias Rosário Ferreira Cruz [119572]*, v2025-11-02

[1 Introduction 1](#_Toc212045169)

[1.1 Overview of the work 1](#_Toc212045170)

[1.2 Current implementation (faults & extras) 1](#_Toc212045171)

[2 Product specification 2](#_Toc212045172)

[2.1 Functional scope and supported interactions 2](#_Toc212045173)

[2.2 System implementation architecture 2](#_Toc212045174)

[2.3 API for developers 2](#_Toc212045175)

[3 Quality assurance 3](#_Toc212045176)

[3.1 Overall strategy for testing 3](#_Toc212045177)

[3.2 Unit and integration testing 3](#_Toc212045178)

[3.3 Acceptance testing 3](#_Toc212045179)

[3.4 Non-functional testing 3](#_Toc212045180)

[3.5 Code quality analysis 3](#_Toc212045181)

[3.6 Continuous integration pipeline [optional] 4](#_Toc212045182)

[4 References & resources 4](#_Toc212045183)

<All remarks in this color should be removed from the final document!

This a template for the expected **content/structure**. You may use any editing tool to prepare the report (LaTeX included).

Feel free to write in Portuguese or English, but do not mix languages between headings and body…>

# Introduction

## 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 Monos – Municipal Waste Collection Booking System, is designed to streamline the process of scheduling bulky waste collection across multiple municipalities. It provides citizens with an interface to book collection sessions by specifying details such as address, municipality, preferred date, time slot, and item description. Complementing this, a dedicated staff dashboard enables municipal workers to monitor, manage, and update the status of bookings throughout their lifecycle (e.g., from "Received" to "Completed"). The system is built as a full-stack web application, with a React frontend and a Spring Boot backend, ensuring a responsive user experience paired with a robust and maintainable server-side architecture.

## Current implementation (faults & extras)

**Known Limitations**

While the core functionality of the application is fully implemented and tested, two non-functional quality objectives could not be completed within the project timeline:

* SonarCloud Integration in CI/CD Pipeline:

Although SonarCloud analysis was configured locally and the project is registered on SonarCloud, the GitHub Actions workflow failed to successfully publish quality metrics due to environment setup complexities and unresolved dependency conflicts in the CI environment.

* Performance Test Reporting:

Gatling-based performance tests were implemented and execute successfully in local development environments. However, automated report generation and publishing as part of the CI pipeline could not be finalized, limiting visibility into performance trends during automated builds.

These limitations affect reporting and observability but do not impact the functional correctness or usability of the application.

**Additional Features Implemented**

Beyond the basic booking functionality, the following enhancements were delivered:

* Staff Dashboard: A dedicated interface for municipal staff to:
  + View all bookings in a card-based layout
  + Filter bookings by municipality or status
  + Update booking status in real time (e.g., from RECEIVED to ASSIGNED, IN\_PROGRESS, COMPLETED, or CANCELLED)
  + View a brief history of status changes for each booking

This dashboard enables an operational overview and a better platform for workflow management, fulfilling the requirement for staff-side interaction with the system.

## Use of generative AI

AI was used strategically to accelerate development while maintaining all academic integrity and learning objectives for the scope of this course.

For the frontend (React), AI assistance was heavily leveraged to implement all of the UI since this was not the primary focus of this project. Therefore, this usage allowed me to spend more time focusing on the core requirements such as API design, and test coverage.

For the backend (Spring Boot), I first designed the system architecture independently—defining entities, class methods, DTOs, and controller endpoints based on the problem domain. Once all of that was done, I used AI to generate a draft implementation to improve efficiency. All generated code was carefully reviewed, tested, and often refactored to ensure correctness and consistency to the problem domain.

Crucially, I chose not to delegate any test code to AI. Writing unit, integration, and end-to-end tests was intentionally done manually, as testing is central to the TQS course objectives and essential for developing a deep understanding of quality assurance practices.

# Product specification

## Functional scope and supported interactions

The Monos – Municipal Waste Collection Booking System serves two primary user roles, each with distinct responsibilities and interactions:

1. **Citizens**

Citizens are residents who need to schedule the collection of bulky waste. Their main interactions include:

* + Booking a collection:

Providing contact information, address, municipality, preferred date/time slot, and a description of the items.

* + Receiving a booking token:

Upon successful submission, a unique token is issued for future reference.

* + Checking booking status:

Using the token to view the current status (e.g., RECEIVED, ASSIGNED, IN\_PROGRESS, COMPLETED).

* + Canceling a booking:

Upon the booking’s unique token the user can then use it to cancel the booking collection.

1. **Municipal Staff**

Staff members manage the operational side of waste collection. Their key interactions are:

* + Viewing all bookings:

A dashboard displays bookings in a card-based layout, showing token, municipality, date, items, and current status.

* + Filtering bookings:

By municipality or status to prioritize workloads (e.g., all RECEIVED bookings in Lisboa).

* + Updating booking status:

Transitioning bookings through the workflow (e.g., from RECEIVED → ASSIGNED → COMPLETED).

* + Viewing status history:

Each booking card shows the last three status changes with timestamps for auditability.

**Visual Summary**

The application supports a linear yet bidirectional service flow, illustrated below:

A diagram of a check-in status

AI-generated content may be incorrect.

## System implementation architecture

The Monos application follows a layered (n-tier) architecture, separating concerns into distinct, loosely coupled components to (mainly) enhance testability. The system is divided into 2 main components:

**Frontend**

Which is served by a React web application. Implements a component-based structure, with dedicated pages for citizen booking, status verification, and the staff dashboard.

Uses Vite as the development and build tool for fast bundling and serving of static assets.

Communicates with the backend through fetch-based HTTP calls to REST endpoints.

Data display is dynamic and responsive, leveraging React state and hooks for real-time interactions.

**Backend**

Served by a Spring Boot application that adheres to a classical layered architecture:

* + Controller layer: Exposes REST endpoints (@RestController) for citizens and staff.
  + Service layer: Contains business logic (e.g., booking creation, status updates, validation).
  + Data layer: Uses Spring Data JPA with Hibernate for ORM and PostgreSQL as the primary database (with H2 in-memory database used for testing and CI environments).
  + DTOs (Data Transfer Objects) are used to decouple internal domain models from API contracts.
  + Logging and error handling are centralized using Spring’s exception handling mechanisms.

**Integration and Deployment**

The application is designed to run as a monolithic service but supports containerization via Docker (with a docker-compose.yml for local development including PostgreSQL).

During development and testing, the frontend runs independently (via Vite) while the backend runs as a standalone Spring Boot process or container.

## API for developers

The Monos backend exposes a RESTful API to support both citizen-facing interactions and internal staff operations. All endpoints return JSON and follow consistent HTTP status conventions. The base URL for all endpoints is /api.

**Citizen Endpoints**

These endpoints enable citizens to create and verify bookings.

* + POST /api/bookings
    - Description: Create a new bulky waste collection booking.
    - Request Body:

{

"contactInfo": "user@example.com",

"address": "Main Street 123",

"municipality": "Lisboa",

"collectionDate": "2025-11-10",

"timeSlot": "MORNING",

"description": "Old sofa and mattress"

}

* + GET /api/bookings/{token}
    - Description: Retrieve booking details and current status using the booking token.
  + DELETE /api/bookings/{token}
    - Description: Changes a given booking’s status (by it’s token)

**Staff Endpoints**

These endpoints support staff dashboard functionality.

* + GET /api/staff/bookings
    - Description: Retrieve all bookings within a date window (default: -7–14 days from today).
  + GET /api/staff/bookings?municipality={name}&date={YYYY-MM-DD}
    - Description: Filter bookings by municipality and collection date.
  + GET /api/staff/bookings?status={status}
    - Description: Filter bookings by current status (e.g., RECEIVED, ASSIGNED).
  + PATCH /api/staff/bookings/{token}/update?newStatus={status}
    - Description: Update the status of a specific booking.

**Municipality Endpoint**

* + GET /api/bookings/municipalities
    - Description: Return a list of supported municipalities (e.g., ["Lisboa", "Porto", "Braga"]) fetched from an external api.

# Quality assurance

## Overall strategy for testing

[what was the overall test development strategy? E.g.: did you do TDD? Did you choose to use Cucumber and BDD? Did you mix different testing tools, like REST-Assured and Cucumber?...]

We did it all baby

## Unit and integration testing

[where did you use unit and integration test? for what? which was the implementation strategy?]

[may add some screenshots/code snippets for clarification, but do not dump all tests here….]

Unit testing was used for business logic – together with service testing because we needed mocking

Also specify here the requirements and how we passed that to the code

## Acceptance testing

[which user-facing test cases did you considered? How were they implemented and automatized?]

[may add some screenshots/code snippets for clarification]

## Non-functional testing

[which non-functional test cases did you consider? Expected: performance study with, at least, load tests]

## Code quality analysis

[which tools/workflow did you use to for [static code analysis](https://www.sonarsource.com/learn/static-code-analysis-using-sonarqube/)?

Show and interpret the results.]

[you may add some interesting lessons learned, e.g., some code smell reported by the tool that was difficult to spot and otherwise you wouldn’t address it]

A screenshot of a computer

AI-generated content may be incorrect.

## Continuous integration pipeline [optional]

[did you implement a CI pipeline? What was the setup? Illustrate with screenshots, if applicable]

# References & resources

Project resources

|  |  |
| --- | --- |
| **Resource:** | **URL/location:** |
| Video demo | <short video-demonstration of your solution; copy into the Git folder, under /docs> |
| QA dashboard (online) | [**optional**; if you have a quality dashboard available online (e.g.: sonarcloud), place the URL here] |
| CI/CD pipeline | [**optional**; if you have th CI pipeline definition in a server, place the URL here] |

Reference materials

<If applicable: document the key components (e.g.: libraries, API) or key references (e.g.: blog post) that were helpful and certainly **would help other students pursuing a similar work**>