HW1: Mid-term assignment report

Ana Loureiro [104063], v2025-11-05

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 software developed consists of a web-based application with the name ZeroMonos, with the objective of helping municipalities manage the collection of large bulky waste items, provided by their citizens

1.2 Current implementation (faults & extras)

The current implementation of ZeroMonos successfully delivers the core functionalities required for the assignment, like the creation and submission of bookings by citizens, a staff page where the state of the request can be updated, as well as filter them by municipalities or states, and a search feature that filters bookings by token or name.
However, since the list of municipalities is retrieved from an external API, if it is down, citizens cannot select a municipality when creating a booking, and the staff members cannot filter the requests by municipality. Additionally, due to the API requests limit, filtering bookings by district is also not implemented, even though the supporting logic is already coded.

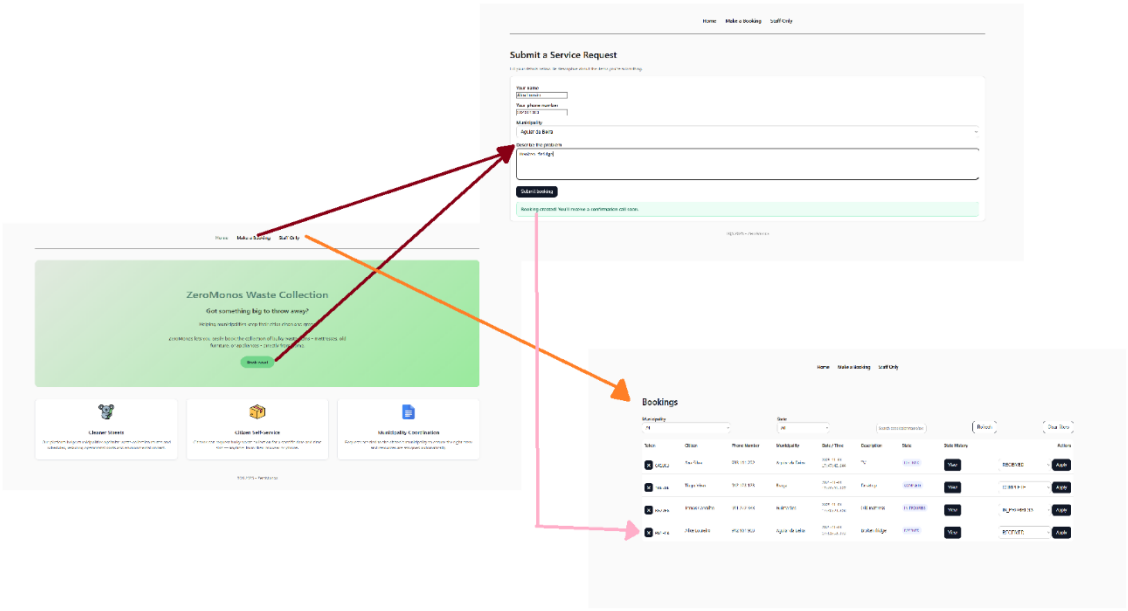
1.3 Use of generative AI

ChatGPT (GPT-5) was used as a coding assistant throughout the development of this project. It was mainly used for the coding of the frontend (specifically the structure, and some implementation of the scripts), as well as debugging errors throughout the project.
GPT-5 was also used as a writing assistant for this report, specifically to help with the tone and structure of phrases.

2 Product specification

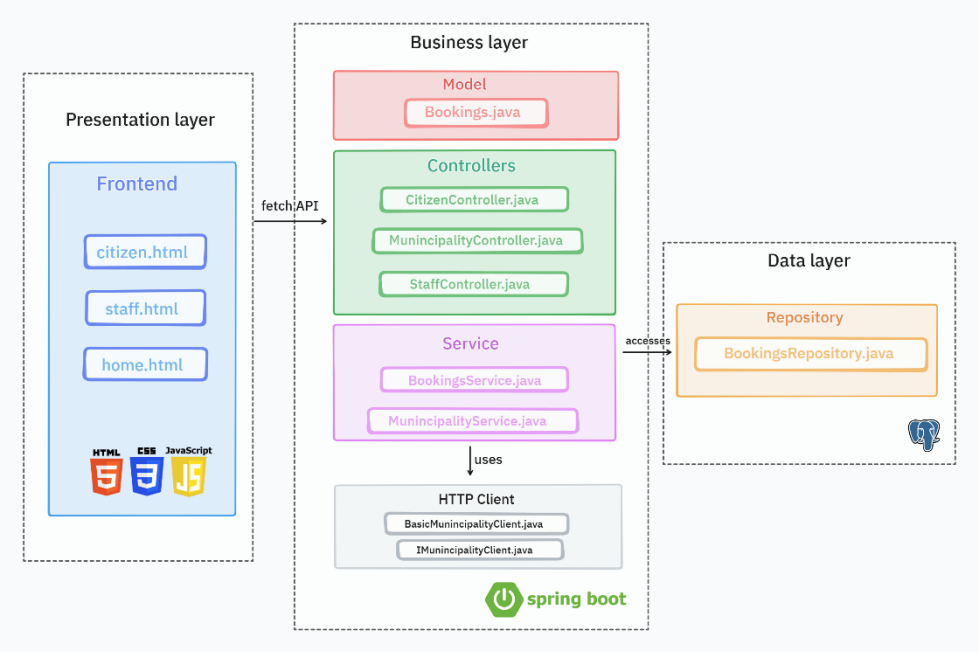
2.1 Functional scope and supported interactions

The target audience of this web application are municipalities' staff and its citizens.
With it, citizens can request the collection of large waste items, by accessing the booking page and submitting a form that includes their name, phone number, municipality and a short description of the items to be collected.
Municipal staff are then responsible for managing and updating these requests. On their specific page, they can view all the bookings submitted by the citizens (as well as filter them by municipality, state or search by name, token or description), update their request's progress state (from RECEIVED to ASSIGNED, IN_PROGRESS, COMPLETE or CANCELED), view the booking's state history in order to track its progress over time, and optionally delete bookings that aren't valid.



2.2 System implementation architecture

The ZeroMonos project is divided into a three-tier architecture, composed of a presentation layer, a business layer and a data layer.
The **presentation layer** (frontend) consists of **HTML**, **CSS** and **JavaScript** webpages that communicate directly with the backend through RESTful endpoints.
The **business layer** (backend) was implemented using Spring Boot, which handles request routing, business logic and integration with an external API (**GeoAPI**)
The **data layer** uses **Spring Data JPA** with a **PostgreSQL** database to record the booking information.



Technologies and Frameworks used:

- **Backend:** Java 17, Spring Boot, Spring Web, Spring Data JPA
- **Testing:** JUnit 5, Mockito, MockMvc, Selenium WebDriver, REST-Assured
- **Frontend:** HTML5, CSS, JavaScript
- **Build Tool:** Maven
- **Code Quality**: SonarQube, JaCoCo

2.3 API for developers

Endpoint	HTTP Method	Description	Request Body (if applicable)
`/citizen`	POST	Creates a new booking request submitted by a citizen.	`{ "citizenName": "Ana", "citizenPhone": "933111222", "municipality": "Porto", "description": "TV" }`
`/staff/bookings`	GET	Retrieves a list of all bookings.	–
`/staff/bookings/{token}`	GET	Returns details of a specific booking.	–
`/staff/bookings/{token}/state`	PUT	Updates the booking's state	`"ASSIGNED"`
`/staff/bookings/{token}/history`	GET	Fetches the full state change history for a booking.	–
`/staff/bookings/{token}`	DELETE	Deletes a booking.	–
`/api/munincipalities`	GET	Retrieves all Portuguese municipalities from the GeoAPI.	–

3 Quality assurance

3.1 Overall strategy for testing

The test development strategy for ZeroMonos was based on a layered approach, combining unit, service, integration and functional testing to ensure reliability and usability.
The development followed a **Test-Driven Development (TDD)** for the service layer, and **Behavior-Driven Development (BDD)** principles for the web interface testing.
Different tools were used to implement these tests, such as:

- **JUnit 5** and **Mockito** for unit testing and mocking dependencies.
- **Spring Boot Test** and **MockMvc** for integration testing of REST endpoints.
- **Selenium WebDriver** for functional and acceptance testing on the web interface.
- **SonarQube** for static code analysis and quality metrics

3.2 Unit and integration testing

Unit tests were used for the service layer, to verify business logic and interaction with dependencies. Using Mockito to mock the needed dependencies (such as repositories and HTTP client), the verifying logic didn't hit the database or call the external API.
The files created were:

- **BookingsServiceTest.java**, that tested the generation of booking tokens, timestamps and state transitions as well as handling the errors when testing functions with an invalid token.
- **MunincipalityServiceTest.java**, that validated parsing of the municipality data from the external API and handled API failures with mock responses

Integration tests were made to verify that the controller and service layer worked correctly together. Using Spring Boot's **WebMvcTest** and **MockMvc**, these tests simulated HTTP requests without requiring the frontend or a real database connection.
The files created were:

- **MunincipalityControllerTest.java**, that verified the **/api/munincipalities** endpoint responses.
- **BookingsControllerTest.java**, which ensured the staff and citizen endpoints behaved as expected:
    - **POST /citizen** – confirms that submitting a valid booking returns a generated token and sets the initial state to *RECEIVED*.
    - **GET /staff/bookings** – retrieves all existing bookings and checks that the response is correctly formatted as a list.
    - **GET /staff/bookings/{token}** – fetches the details of a specific booking using its token and validates all returned fields.
    - **PUT /staff/bookings/{token}/state** – verifies that updating the booking state correctly changes the record and logs the change in the booking's history.
    - **GET /staff/bookings/{token}/history** – ensures that the booking's state history is accurately returned.

3.3 Acceptance testing

Acceptance tests were made to verify that the web-application behaves correctly with the user's inputs. These were implemented using **Selenium Webdriver**, in a BDD approach, and tested the following scenarios:

- Submitting a new booking:

  Given the citizen is on the *"Make a Booking"* page
  When they fill in their personal details, select a municipality, and submit the form
  Then a confirmation message is displayed showing a unique booking token
  And the booking appears in the staff dashboard with the state *RECEIVED*.

- Viewing and filtering bookings on the staff page

  Given the staff member is on the *"Staff only"* page
  When they apply filters by municipality, state, or keyword
  Then only the bookings that match the selected criteria are displayed in the table.

- Updating a booking's state

  Given the staff member is viewing the list of bookings (*"Staff only"* page)
  When they select a new state from the dropdown and click *Apply*
  Then the booking's state updates successfully
  And a confirmation message "State updated" appears.

- Deleting an existing booking

  Given the staff member is on the bookings list page (*"Staff only"* page)
  When they click the X button next to a booking
  Then a confirmation prompt appears
  And once confirmed, the booking is permanently removed from the list
  And a message "Booking deleted successfully" is displayed

Each scenario opened the web interface (http://localhost:8081) and interacted with UI elements (buttons, dropdowns, messages). The tests validated the presence of expected visual feedback ("State updated successfully") and correct dynamic behavior of filters and modals.

3.4 Non-functional testing

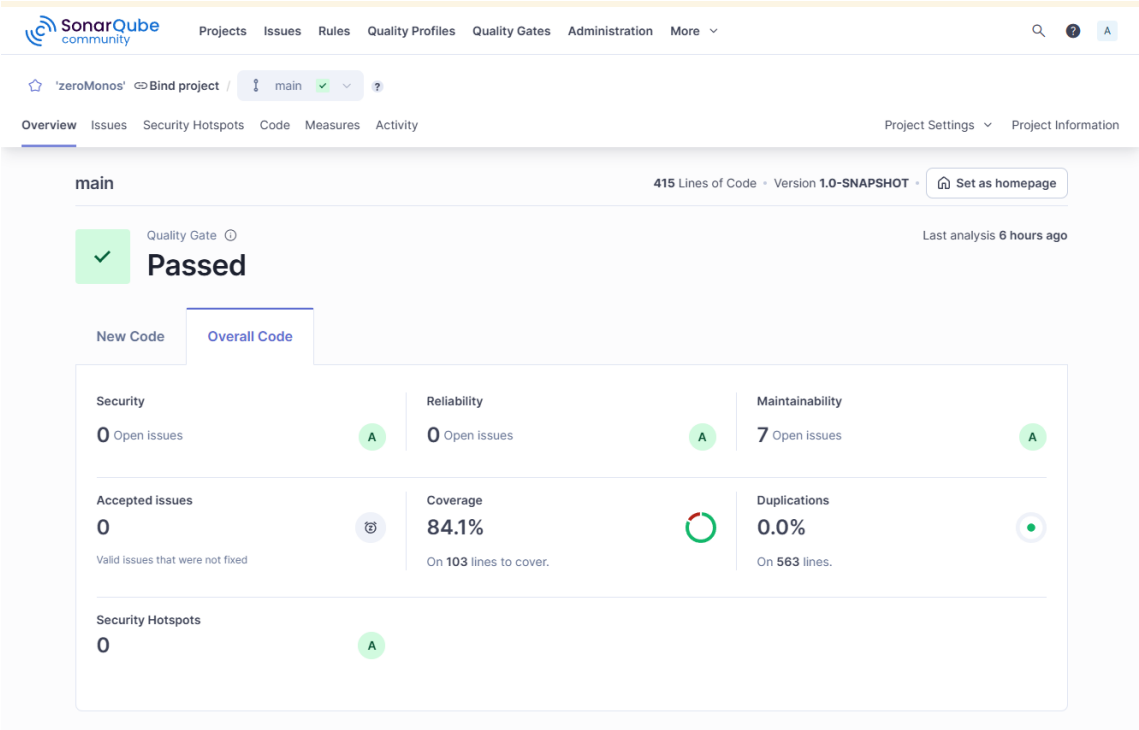
No non-functional testing was performed.

3.5 Code quality analysis

Code quality analysis was performed using **SonarQube**, deployed locally via Docker. Each commit was analyzed using Maven's Sonar plugin, reporting code smells, test coverage, complexity, and maintainability metrics.

Key insights from the SonarQube report:

- **Code coverage:** ~80% for the service and controller layers.
- **No major code smells** or duplicated code detected after refactoring.
- The overall maintainability and reliability ratings were both **A**, with minimal technical debt.



4 References & resources

Project resources

Resource:	URL/location:
Video demo	https://youtu.be/r5LclDDO31c

Reference materials

Most of the work was developed based on the experience acquired during the TQS practical classes, with significant guidance taken from the documentation and README files done throughout the lab sessions.