HW1: Mid-term Assignment Report

Gonçalo Silva, November 7, 2025

DETI – Universidade de Aveiro

Contents

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 full-stack web application is **ZeroMonos**.

It provides garbage collection services for multiple municipalities. It allows citizens to **book the collection of large bulky waste items** (such as mattress, old appliances, etc.).

1.2 Current implementation (faults & extras)

The current implementation includes all the main required features, with a backend exposing a REST API and a web interface demonstrating the core user stories.

Additionally, I implemented some extra features.

Users can now book garbage collection not only at the municipality level but also for specific districts and villages. I also added user authentication using JWT, distinguishing between two roles: **citizen** and **staff**. This role-based access control ensures that certain endpoints ar restricted depending on the user type, preventing unauthorized actions. This feature is particularly important because the system requirements define distinct operations for each user type.

I also added a dashboard where staff members can visually monitor requests and operations for each district/municipality.

1.3 Use of generative AI

AI assistants were used throughout the development process.

For the backend, AI helped clarify authentication configurations and security settings, particularly for JWT implementation, endpoint protection, and CORS configuration. However, tutorial references (listed at the end of the report) served as the primary source, and AI was used only as a supplementary guide. Additionally, AI was used to generate repetitive test cases that followed similar patterns, reducing manual effort.

For the frontend, AI assisted with page layout design and overall UI structure.

Parts I did not delegate to AI:

- Business logic implementation services
- Controller design and implementation
- Backend-to-frontend communications services
- Overall architecture decisions

This approach allowed me to maintain full understanding and control over the core functionality of the system leveraging AI to reduce repetitive work.

2 Product specification

2.1 Functional scope and supported interactions

The ZeroMonos system is designed to streamline bulky waste collection services around multiple municipalities. It provides a digital platform for both citizens and staff members to manage collection requests.

Main actors:

- **Citizen**: A registered user who can request the collection of bulky waste items. A citizen can:
  - Create collection requests
  - View the status of their bookings
  - Cancel bookings
- **Staff member**: An authorized worker who manages incoming collection requests: A staff member can:
  - View and filter all collection requests
  - Update booking state (assigned, in progress, done, canceled)

The system supports user authentication via JWT, ensuring that citizens and staff have access only to features relevant to their roles. Interactions occurr through a web interface connected to a RESTful backend API.

Typical interaction flow:

1. A citizen logs in and submits a bulky waste collection request.
2. The request is stored and becomes visible to staff members.
3. A staff member updates the state of the request accordingly.
4. The citizen can track the status of their request via the web interface.

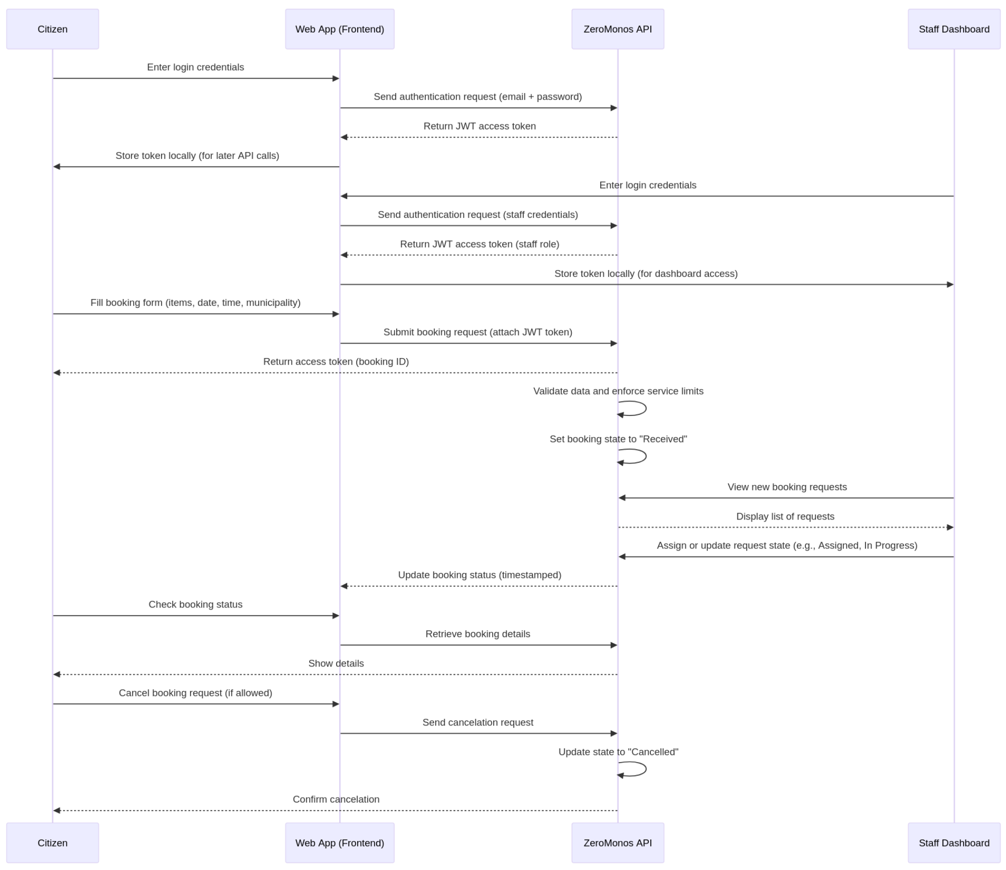

Citizen
Web App (Frontend)
ZeroMonos API
Staff Dashboard
Enter login credentials
Send authentication request (email + password)
Return JWT access token
Store token locally (for later API calls)
Enter login credentials
Send authentication request (staff credentials)
Return JWT access token (staff role)
Store token locally (for dashboard access)
Fill booking form (items, date, time, municipality)
Submit booking request (attach JWT token)
Return access token (booking ID)
Validate data and enforce service limits
Set booking state to "Received"
View new booking requests
Display list of requests
Assign or update request state (e.g., Assigned, In Progress)
Update booking status (timestamped)
Check booking status
Retrieve booking details
Show details
Cancel booking request (if allowed)
Send cancelation request
Update state to "Cancelled"
Confirm cancelation
Citizen
Web App (Frontend)
ZeroMonos API
Staff Dashboard

Figure 1: System interactions

2.2 System implementation architecture

The ZeroMonos system follows a clear separation between frontend and backend, in a way that in the future, it is possible to instead of a web page, develop a mobile app that could perfectly interact with the current backend implementation.

Architecture Overview:

- **Frontend**: Web-based user interface.
- **Backend**: A RESTful API server that implements the business logic of the system.
- **Database**: A relational database that stores all persistent data.

Technologies used:

- **Frontend**: React.js with MaterialUI.
- **Backend**: Spring Boot.
- **Database**: PostgreQSL (H2 for development adn testing).
- **Other tools**: Json Web Tokens (JWT) for authentication.

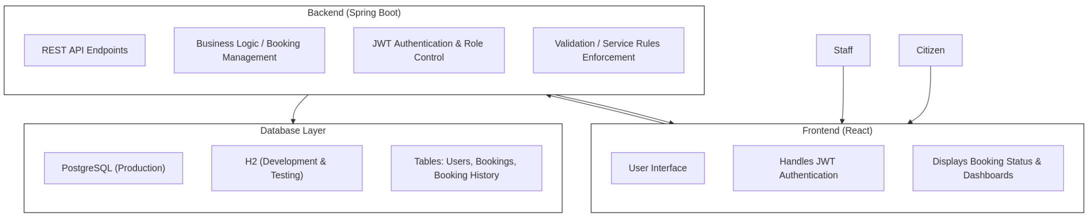

Backend (Spring Boot)
REST API Endpoints
Business Logic / Booking Management
JWT Authentication & Role Control
Validation / Service Rules Enforcement
Staff
Citizen
Database Layer
PostgreSQL (Production)
H2 (Development & Testing)
Tables: Users, Bookings, Booking History
Frontend (React)
User Interface
Handles JWT Authentication
Displays Booking Status & Dashboards

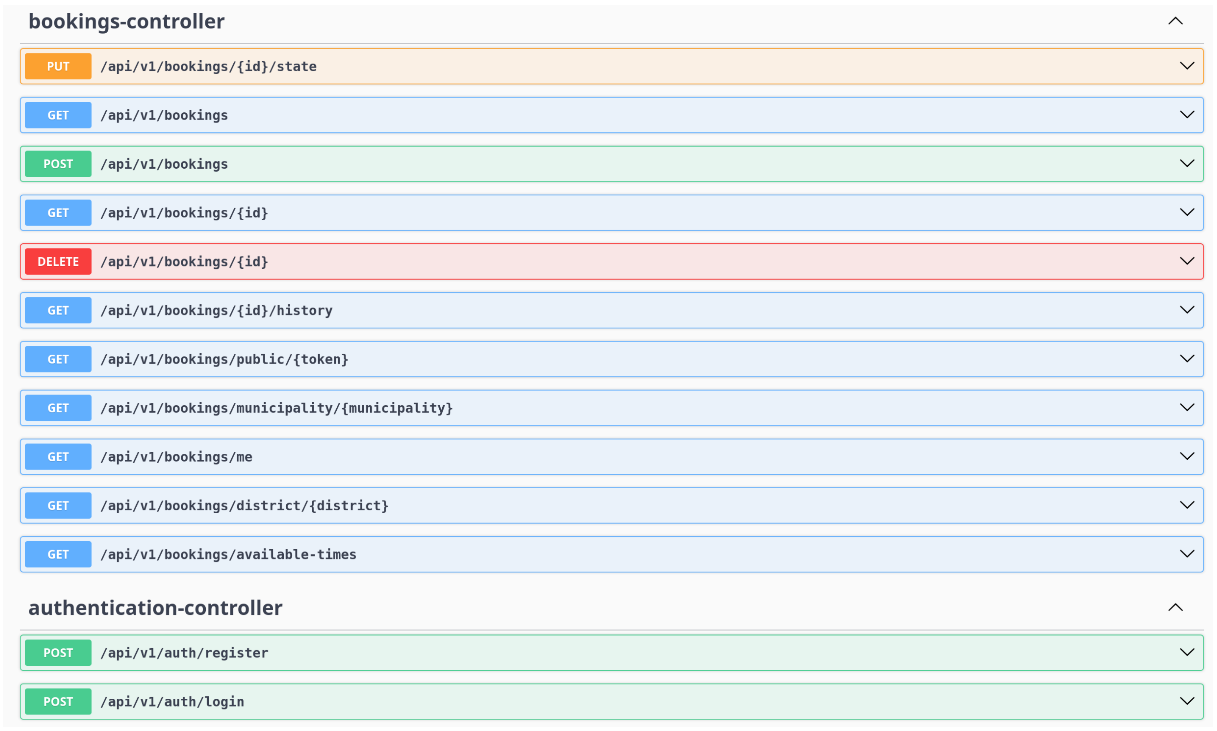
Figure 2: System architecture overview

2.3 API for developers

The ZeroMonos system exposes a set of RESTful API endpoints that allow developers to interact programmatically with the platform. These endpoints support operations for both citizens and staff, including user authentication, booking management, and access to booking history. Role-based access control is enforced via JWT tokens, ensuring that only authorized users can perform specific actions.

Endpoint	**Access Level**
POST /api/v1/auth/register	Public
POST /api/v1/auth/login	Public
GET /api/v1/bookings/public/{token}	Public
POST /api/v1/bookings	Citizen-only
GET /api/v1/bookings/me	Citizen-only
DELETE /api/v1/bookings/{id}	Citizen-only
GET /api/v1/bookings	Staff-only
PUT /api/v1/bookings/{id}/state	Staff-only
GET /api/v1/bookings/municipality/{municipality}	Staff-only
GET /api/v1/bookings/district/{district}	Staff-only
GET /api/v1/bookings/{id}	Citizen or Staff
GET /api/v1/bookings/{id}/history	Citizen or Staff
GET /api/v1/bookings/available-times	Citizen or Staff

Table 1: ZeroMonos API endpoints and their access levels


bookings-controller

PUT /api/v1/bookings/{id}/state
GET /api/v1/bookings
POST /api/v1/bookings
GET /api/v1/bookings/{id}
DELETE /api/v1/bookings/{id}
GET /api/v1/bookings/{id}/history
GET /api/v1/bookings/public/{token}
GET /api/v1/bookings/municipality/{municipality}
GET /api/v1/bookings/me
GET /api/v1/bookings/district/{district}
GET /api/v1/bookings/available-times

authentication-controller

POST /api/v1/auth/register
POST /api/v1/auth/login

Figure 3: ZeroMonos API endpoints

3 Quality assurance

3.1 Overall strategy for testing

The testing strategy was primarily focused on ensuring the correctness of services, controllers, and, in some cases, repository operations. Following a code-first approach, I implemented the application logic first and then developed the corresponding tests to verify its functionality. This allowed for incremental validation of each component as it was developed.

3.2 Unit and integration testing

Unit tests were primarily used to verify the correctness of backend components such as services, controllers, and repositories. Each test focused on a small, isolated part of the system to ensure that the core business logic behaved as expected.

Integration testing focused on verifying interactions between components, particularly the frontend and backend. For this purpose, **Cucumber** with **Selenium** was used to define BDD-style scenarios covering key user stories, such as booking creation, status updates, and role-based access.

Due to technical issues integrating Cucumber with Spring Boot, the frontend integration tests were implemented but could not be executed, as the system could not locate the Cucumber tests.

Example unit test snippet (service layer):

Listing 1: Booking service unit test

```
@Test
@DisplayName("Create␣booking")
void testCreateBooking() {
    BookingRequest request = BookingRequest.builder()
        .district("District")
        .municipality("Lisbon")
        .village("Sintra")
        .postalCode("0000-000")
        .date(testDate)
        .time(LocalTime.of(10, 0))
        .description("item␣1")
        .build();

    Booking result = bookingsService.createBooking(request, "
        bob@email.com");

    assertThat(result).isNotNull();
    assertThat(result.getState()).isEqualTo(State.RECEIVED);
}
```

Example integration test snippet (controller layer):

Listing 2: BookingsController integration test

```
@Test
@WithMockUser(username = "bob@email.com", roles = {"CITIZEN"})
@DisplayName("POST␣/api/v1/bookings␣creates␣a␣booking␣
    successfully")
void testCreateBooking() throws Exception {
    BookingRequest request = BookingRequest.builder()
        .district("District")
        .municipality("Lisbon")
        .village("Sintra")
        .postalCode("0000-000")
        .date(futureDate)
        .time(LocalTime.of(10, 0))
        .description("item␣1")
        .build();

    mockMvc.perform(post("/api/v1/bookings")
        .contentType(MediaType.APPLICATION_JSON)
        .content(objectMapper.writeValueAsString(request)))
        .andExpect(status().isCreated())
        .andExpect(jsonPath("$.municipality").value("Lisbon"))
        .andExpect(jsonPath("$.state").value("RECEIVED"));
}
```

3.3 Acceptance testing

Acceptance testing focused on user-facing workflows defined in the main user stories:

- Citizens creating, viewing, and canceling bookings
- Staff updating booking states and monitoring requests

Example acceptance scenario:

Listing 3: Booking tracking acceptance test

```
Scenario: Citizen creates a new booking successfully
    Given I am logged in as a citizen
    When I create a booking with the following details:
        | district        | Lisbon              |
        | municipality    | Lisbon              |
        | village         | Chiado              |
        | postalCode      | 1000-100            |
        | date            | 2025-11-10          |
        | time            | 10:00               |
        | description     | Collect electronics |
    Then the booking should be created successfully
    And I should receive a booking token
```

3.4 Non-functional testing

For non-functional testing, I used **Google Lighthouse** to evaluate the performance and quality of the web application. Lighthouse provides automated audits for performance, accessibility, best practices, and SEO. These audits were run on the frontend (development mode).

The results highlighted performance as the weakest area, mainly due to the way data is fetched and processed from external APIs. In particular, when loading the application, four consecutive API calls are made to the *E-Redes* service to retrieve all districts and municipalities. These are then preprocessed on the client side and cached in `localStorage` for later reuse. This of course could be improved in the future. While this approach reduces subsequent load times, the initial startup is noticeably slow, as the API responses must all be processed before the user can interact with the interface. Furthermore, the villages API is accessed each time the municipality selection changes, which adds further latency during navigation.

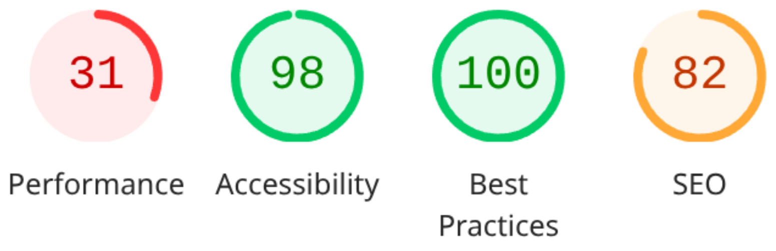
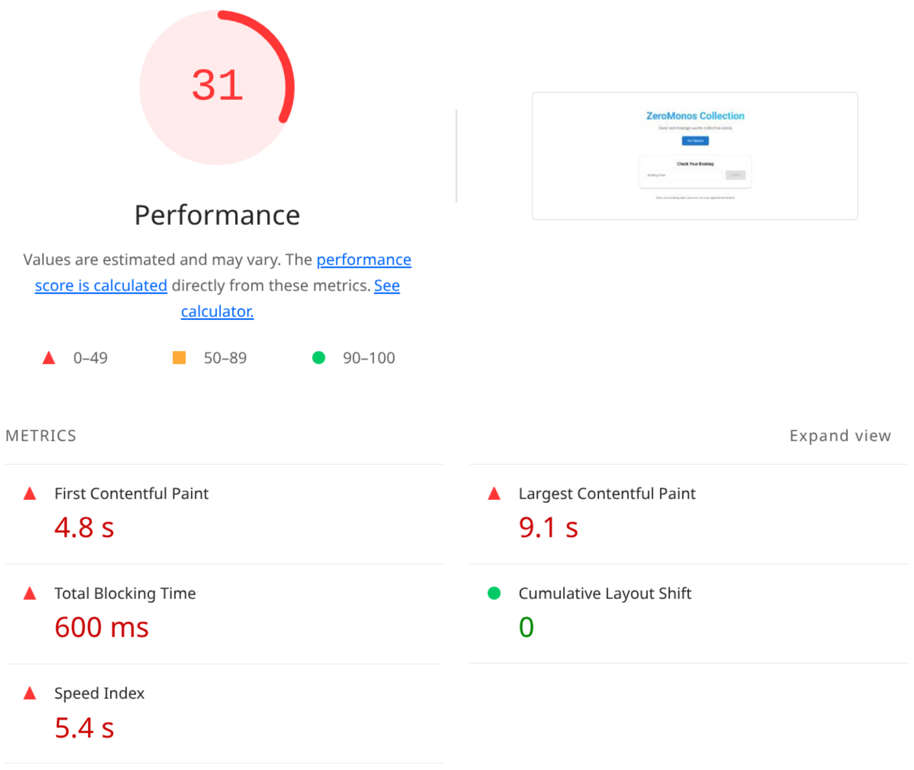


Figure 4: Lighthouse general audit



31

Performance

Values are estimated and may vary. The performance score is calculated directly from these metrics. See calculator.

▲ 0–49 ■ 50–89 ● 90–100

METRICS Expand view

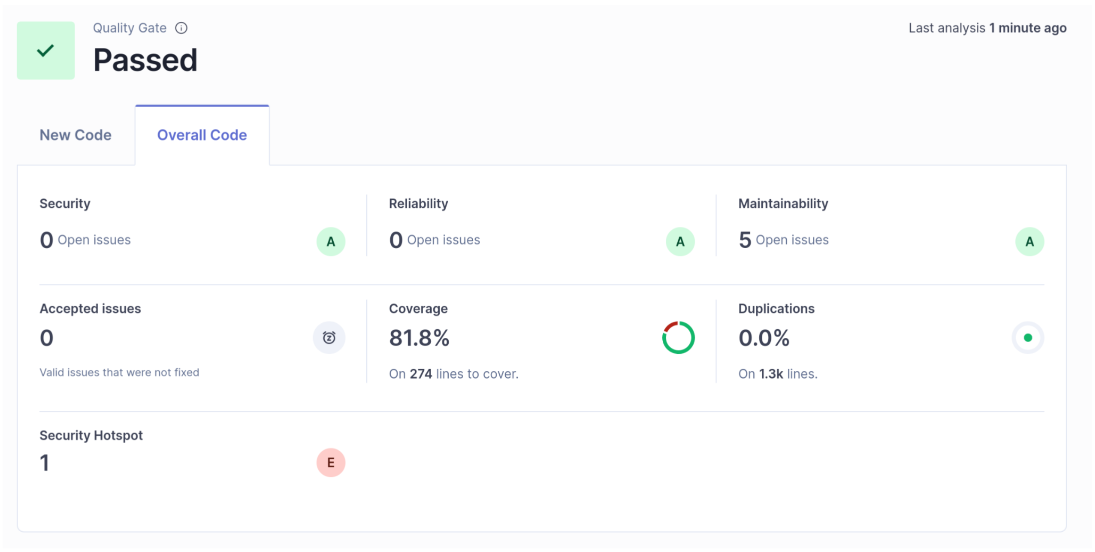
▲ First Contentful Paint 4.8 s	▲ Largest Contentful Paint 9.1 s
▲ Total Blocking Time 600 ms	● Cumulative Layout Shift 0
▲ Speed Index 5.4 s	

Figure 5: Lighthouse performace audit

3.5 Code quality analysis

For static code analysis, I used **SonarQube**, which was deployed locally through Docker. The tool was integrated into the project to assess various quality dimensions, such as security, reliability, maintainability, and test coverage.

The latest analysis successfully passed the *Quality Gate*, with the following key metrics:



Quality Gate

Passed

Last analysis 1 minute ago

New Code | Overall Code

Security 0 Open issues A	Reliability 0 Open issues A	Maintainability 5 Open issues A
Accepted issues 0 Valid issues that were not fixed	Coverage 81.8% On 274 lines to cover.	Duplications 0.0% On 1.3k lines.
Security Hotspot 1 E		

Figure 6: SonarQube results

Overall, SonarQube rated the project with high marks in all major categories. The **Quality Gate was passed**, confirming that the code meets the defined thresholds for maintainability and test coverage. One lesson learned was the importance of maintaining test coverage and code readability, as SonarQube helped identify small areas of improvement, such as redundant code blocks and some fields that could be static.

The single **security hotspot** detected by SonarQube was in the `SecurityConfiguration` class, where the following code disables Spring Security's CSRF protection:

Listing 4: CSRF configuration flagged as a security hotspot

```
http
    .csrf(csrf -> csrf.disable())
    .cors(cors -> {})
    .authorizeHttpRequests(auth -> auth
        .requestMatchers(
            "/api/v1/auth/**",
            "/api/v1/bookings/public/**",
            "/swagger-ui/**",
            "/v3/api-docs/**",
            "/swagger-ui.html"
        ).permitAll()
        .anyRequest().authenticated());
```

SonarQube correctly highlighted this as a potential risk because disabling CSRF protection can expose an application to cross-site request forgery attacks. However, in this case, the API is **stateless and uses JWT-based authentication**, meaning that each request carries its own authentication token and does not rely on server-side sessions or cookies. Therefore, disabling CSRF protection is **safe and appropriate** in this context.

3.6 Continuous integration pipeline

A simple **Continuous Integration (CI)** pipeline was implemented using **GitHub Actions** to ensure that code committed to the main branch passes all tests.

The workflow is triggered automatically on every push to the `master` branch. It performs the following steps:

1. Checks out the project repository.
2. Sets up the Java Development Kit (JDK 21) using the Temurin distribution.
3. Caches Maven dependencies to improve build performance.
4. Runs all unit and integration tests defined in the backend (`HW1/zeromonos-collection`) using Maven.

Workflow configuration file:

Listing 5: GitHub Actions CI pipeline for backend

```
name: Backend CI

on:
  push:
    branches: ["master"]

jobs:
  test:
    name: Run HW1 Tests
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Setup JDK 21
        uses: actions/setup-java@v4
        with:
          distribution: temurin
          java-version: 21
          cache: maven

      - name: Run tests in HW1 folder
        working-directory: HW1/zeromonos-collection
        run: mvn clean verify -B
        env:
          JWT_SECRET: ${{ secrets.JWT_SECRET }}
```

4 References & resources

4.1 Reference materials

The following online resources and tutorials were used during development:

- **E-Redes Open Data – Districts of Portugal:** https://e-redes.opendatasoft.com/explore/dataset/districts-portugal/api/?disjunctive.dis_code&disjunctive.dis_name&sort=year
- **E-Redes Open Data – Municipalities of Portugal:** https://e-redes.opendatasoft.com/explore/dataset/municipalities-portugal/api/?disjunctive.dis_code&disjunctive.dis_name&disjunctive.con_code&disjunctive.con_name&sort=year
- **GeoAPI Portugal:** service used for location and mapping references. https://geoapi.pt/?mapa=3
- **JWT Authentication in Spring Boot (Medium):** tutorial explaining how to implement JWT-based authentication. https://medium.com/@pendemmukulsai/implementing-jwt-authentication-in-spring-boot-2025-03a565333814

- **Baeldung – Spring REST API Documentation with OpenAPI:** `https://www.baeldung.com/spring-rest-openapi-documentation`
- **Amigoscode – Spring Boot 3 + Security + JWT (YouTube):** `https://www.youtube.com/watch?v=KxqlJblhzfI`