HW1: Mid-term assignment report

*Marta Dias Rosário Ferreira Cruz [119572]*, v2025-11-03

[1 Introduction 1](#_Toc212045169)

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

[1.2 Current implementation (faults & extras) 2](#_Toc212045171)

[2 Product specification 3](#_Toc212045172)

[2.1 Functional scope and supported interactions 3](#_Toc212045173)

[2.2 System implementation architecture 4](#_Toc212045174)

[2.3 API for developers 4](#_Toc212045175)

[3 Quality assurance 5](#_Toc212045176)

[3.1 Overall strategy for testing 5](#_Toc212045177)

[3.2 Unit and integration testing 6](#_Toc212045178)

[3.3 Acceptance testing 9](#_Toc212045179)

[3.4 Non-functional testing 10](#_Toc212045180)

[3.5 Code quality analysis 11](#_Toc212045181)

[3.6 Continuous integration pipeline [optional] 13](#_Toc212045182)

[4 References & resources 15](#_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 application successfully implements all required functionality for the waste collection booking system, it's important to acknowledge its intentional simplifications:

Authentication and Authorization

The current implementation does not include user authentication or role-based access control (citizen vs. staff). In a production environment, this would be crucial to ensure that only authorized staff members can update booking statuses and that citizens can only manage their own bookings. This was deliberately left out as it wasn't part of the core requirements for this proof of concept.

Simplified Domain Model

The domain model was kept intentionally minimal to focus on the core booking workflow. While sufficient for the requirements, a production system might benefit from additional entities and relationships (e.g., detailed waste categories, equipment tracking, staff assignments) to provide a richer user experience and more detailed operational insights.

**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
  + 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 used to implement all of the UI since this was not the primary focus of this project. Therefore, allowing 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 reviewed, tested, and often refactored to ensure correctness and consistency to the problem domain.

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. **Staff**

Staff members manage the operational side of waste collection. Key interactions:

* + 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 architecture, separating concerns into distinct, loosely coupled components to enhance testability. The system is divided into 2 main components:

1. **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.

1. **Backend**

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

* + Controller layer: Exposes REST endpoints 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 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 -- 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) to canceled.

**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 given 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

The testing strategy adopted for this project follows an approach that combines several complementary methodologies and frameworks.

**Behavior-Driven Development (BDD)**

This was implemented using Cucumber, allowing requirements to be expressed as human-readable Gherkin scenarios.

**End-to-End (E2E) Functional Testing**

Was implemented with Playwright, integrated into Cucumber step definitions. This enabled realistic browser-based simulation of user interactions (e.g., form submission, navigation, status updates) against the live React frontend and Spring Boot backend.

**Unit and Integration Testing**

These tests were written using JUnit 5 and AssertJ, covering core business logic, service layers, and repository behavior. These tests were developed manually (not generated by AI) to reinforce understanding of test design principles and edge-case coverage.

**Performance Testing**

Was introduced via Gatling, with custom Scala-based simulations modeling concurrent citizen bookings, staff updates, and capacity-limit scenarios. While reporting integration in CI remains pending, local execution validates system responsiveness under load.

Note on test Isolation and Shared State: To support complex E2E flows between Citizen and Staff, Cucumber’s dependency injection via PicoContainer was used to share a crucial part of the workflow logic -- booking tokens -- between step definition classes.

Note on TDD: Although it was not strictly followed for all components (primarily due to time constraints), test-first practices were applied selectively specially during the implementation of crucial parts of the logic design.

## Unit and integration testing

Unit and integration tests implemented to validate core business logic and ensure correct service-layer behavior under both valid and invalid conditions. These tests are located at the BookingServiceImpl class, which encapsulates the application’s domain rules—such as booking validation, capacity limits, status transitions, and token generation.

**Unit testing - Test Strategy**

* + Unit tests check individual methods in an isolated way using mocking –Mockito -- to decouple from external dependencies such as the database (BookingRequestRepo) and external services (MunicipalityService).
  + Integration-like behavior is simulated by controlling mock responses to reflect real-world scenarios -- capacity exhaustion, invalid municipalities, etc.
  + AssertJ was used in the context of assertions.
  + All tests follow a requirement-driven approach: each test case maps directly to a functional or validation rule defined in the problem specification (as seen bellow)

**Unit testing - Requirements to Tests mapping**

Below is a list on how some of the key requirements were mapped to unit tests:

|  |  |
| --- | --- |
| Bookings are only allowed in valid municipalities | Mock municipalityService.isValid() to return false and assert that InvalidBookingException is thrown |
| Maximum 15 bookings per municipality, date, and time slot | Mock repository to return 15 existing bookings; verify rejection |
| Bookings cannot be created in the past or more than 14 days ahead | Call validation method with invalid dates and assert exceptions |
| Staff may update booking status only if booking exists | Mock non-existent token; expect exception on update |
| Cancellation is only allowed for non-final statuses (e.g., not completed) | Mock a completed booking; assert cancellation is blocked |
| Each booking must have a unique, 20 character alphanumeric token | Verify token format and uniqueness across calls |

The results of these tests are depicted bellow:

A screenshot of a computer program

AI-generated content may be incorrect.

**Integration tests**

In addition to unit tests, integration tests were implemented to verify the correct interaction between the web layer and the service layer, ensuring that HTTP endpoints behave as expected under both valid and invalid input conditions. These tests simulate real client requests to the application’s REST API using Spring’s MockMvc, while mocking only the external service dependencies – BookingService -- to isolate the web layer behavior without requiring a database or external systems.

For this purpose I created 2 dedicated test classes:

* + CitizenBookingControllerIntegrationTest: Validates the citizen-facing booking lifecycle (creation, retrieval, cancellation).
  + StaffBookingControllerIntegrationTest: Validates staff-facing operations (listing, filtering, and updating booking status).

**Integration testing - Test Strategy**

* + @SpringBootTest + @AutoConfigureMockMvc: Bootstraps a minimal Spring context with web components, enabling realistic request/response simulation without starting a full server.
  + @MockBean: Replaces the BookingService with a Mockito mock, allowing precise control over service-layer responses (e.g., returning a booking, throwing exceptions) while preserving controller logic validation.
  + JSON serialization/deserialization: Uses Spring’s injected ObjectMapper to ensure payloads match the actual API contract.
  + HTTP status and response body assertions: Validates not only the success or failure of a request but also the structure and content of the returned data (e.g., token, status, municipality).
  + Exception-to-HTTP mapping: Confirms that service-layer exceptions (e.g., InvalidBookingException) are correctly translated into appropriate HTTP responses (e.g., 404 Not Found, 400 Bad Request).

**Integration testing - Requirements to Tests mapping**

Below is a list on how some of the key requirements were mapped to integration tests:

|  |  |
| --- | --- |
| Citizen should be able to create bookings – valid request | Send a valid POST request to /api/bookings with required fields (name, email, municipality, date, time slot). Assert that the response is 201 Created and contains a booking token. |
| Citizen should be able to create bookings – invalid request | Send a POST request to /api/bookings with missing required fields. Assert that the response is 400 Bad Request. |
| Citizen should be able to create bookings – if there’s still capacity available | Send a POST request to /api/bookings for a municipality, date, and time slot that already has 15 bookings. Assert that the response is 400 Bad Request. |
| Citizen should be able to create bookings – if date range is valid | Send a POST request to /api/bookings with a date in the past or more than 14 days ahead. Assert that the response is 400 Bad Request. |
| Citizen should be able to obtain a booking by the token – valid token | Send a GET request to /api/bookings/{token} with a valid token. Assert that the response is 200 OK and contains the correct booking details. |
| Citizen should be able to obtain a booking by the token – invalid token | Send a GET request to /api/bookings/{token} with an invalid or non-existent token. Assert that the response is 404 Not Found. |
| Citizen should be able to cancel a booking by the token – valid token | Send a DELETE request to /api/bookings/{token} with a valid token. Assert that the response is 200 OK and the booking status is updated to CANCELLED. |
| Citizen should be able to cancel a booking by the token – invalid token | Send a DELETE request to /api/bookings/{token} with an invalid or non existent token. Assert that the response is 404 Not Found. |
| Staff should have access to all bookings | Send a GET request to /api/staff/bookings. Assert that the response is 200 OK and contains a list of all bookings. |
| Staff should be able to filter bookings by status | Send a GET request to /api/staff/bookings?status={status} with a valid status. Assert that the response is 200 OK and contains only bookings with that status. |
| Staff should be able to filter bookings by municipality and date | Send a GET request to /api/staff/bookings?municipality={municipality}&date={date} . Assert that the response is 200 OK and contains only bookings for that municipality and date. |
| Staff should be able to update a given booking’s status by it’s token – valid status | Send a PATCH request to /api/staff/bookings/{token}/update?newStatus={status}  with a valid token and a new status. Assert that the response is 200 OK and the booking status is updated. |
| Staff should be able to update a given booking’s status by it’s token – invalid status | Send a PUT request to /api/staff/bookings/{token}/update?newStatus={status} with an invalid status. Assert that the response is 400 Bad Request. |

The results of these tests are depicted bellow:

A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

## Acceptance testing

Acceptance testing was implemented using a Behavior-Driven Development (BDD) approach to validate end-to-end user workflows from both citizen and municipal staff perspectives. The goal was to ensure the system behaves as expected under realistic usage conditions.

**Test Scenarios and Coverage**

A single but comprehensive end-to-end (E2E) scenario --“Successful booking creation and status check” -- was defined in Gherkin syntax to cover the full lifecycle of a booking:

* + Citizen workflow:
    - Navigates to the booking form
    - Fills in contact info, address, municipality, date, time slot, and item description
    - Submits the form and receives a unique booking token
    - Navigates to the status-check page and verifies the initial status is RECEIVED
  + Staff workflow:
    - Accesses the staff dashboard
    - Locates the booking through the shared token
    - Updates its status to ASSIGNED
    - Confirms the new status is reflected in the UI

**Automation Framework and Implementation**

The acceptance tests were fully automated using the following stack:

* + Cucumber JVM for BDD and scenario definition
  + Playwright for browser automation (chromium in headless mode)
  + JUnit 5 as the test runner
  + PicoContainer for dependency injection through SharedContext so that we’re able to pass the booking token between step definitions

**Step definitions were organized into two classes:**

* + CitizenBookingSteps.java: Handles all citizen-facing interactions
  + StaffBookingSteps.java: Manages staff dashboard actions

**Key technical practices include:**

* + Explicit waiting by using page.waitForSelector() to avoid race conditions – tests need elements that have not been loaded into the page yet
  + Shared state management – booking token -- through a SharedContext object, enabling data exchange between citizen and staff steps within the same scenario
  + Visual debugging support: Automatic screenshot capture on test failure for easier troubleshooting – this was mainly used when setting up GitHub Actions.

The results of these tests are depicted bellow:

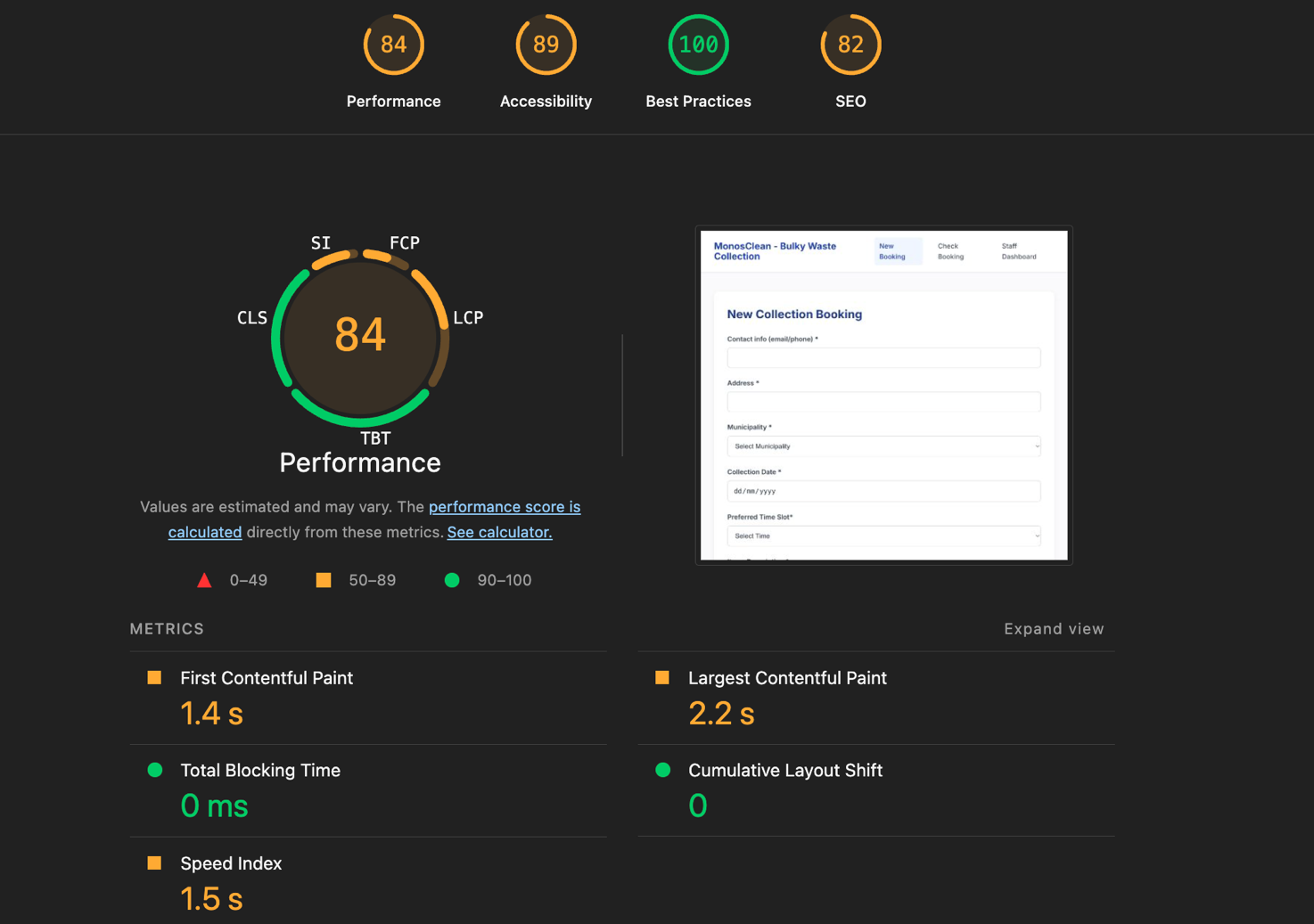
A screenshot of a computer

AI-generated content may be incorrect.

## Non-functional testing

To evaluate the system’s frontend performance and user experience, non-functional testing was conducted using Lighthouse.A Lighthouse performance audit was executed on the live frontend (<http://localhost:5173/> -- vite distribution server) under simulated desktop conditions. The results are as follows:

* + Performance: 84/100
    - First Contentful Paint: 1.4 s
    - Largest Contentful Paint: 2.2 s
    - Total Blocking Time: 0 ms
    - Cumulative Layout Shift: 0
    - Speed Index: 1.5 s
  + Accessibility: 89/100
    - Minor issues were identified, including form elements lacking associated labels.
  + Best Practices: 100/100
    - The application adheres to modern web standards, including secure headers and proper use of web APIs.
  + SEO: 82/100
    - The main issue was a missing <meta name="description"> tag and an invalid robots.txt file, both of which can impact search engine indexing.



## Code quality analysis

For static code analysis, I made use of SonarQube -- SonarCloud service -- as the main tool for QA. The integration was set up to automatically make an analysis of the codebase on every commit or pull request, providing immediate feedback on code quality, security, and maintainability.

The workflow involved:

* + Integration: Configuring the project's CI/CD pipeline (GitHub Actions) to trigger a SonarQube analysis.
  + Analysis: SonarQube scanned the entire codebase, evaluating it against predefined rules for bugs, vulnerabilities, code smells, test coverage, and duplications.
  + Reporting: The results were aggregated into a dashboard, accessible via the SonarCloud web interface, which provided a clear summary of the code's health.

**Initial Analysis Results**

The initial analysis revealed several areas requiring improvement:

* + Security: 2 security issues were identified
  + Reliability: 1 open issue was detected
  + Maintainability: 14 code smells were found
  + Coverage: Test coverage was at 56.3%
  + Duplications: 8.5% of the codebase contained duplicated lines

**Improved Analysis Results After Refactoring**

After addressing the issues highlighted by SonarQube, significant improvements were achieved:

* + Security: All security issues were resolved, bringing the count down to 0
  + Reliability: All reliability issues were fixed, now standing at 0
  + Maintainability: While 14 code smells remain, they are minor and don't impact the quality gate
  + Coverage: Test coverage improved substantially to 87.3%, surpassing the industry standard of 80-90%
  + Duplications: Slight improvement in code duplication, now at 8.4%

Note on Maintainability: Due to time constraints I had no time to fix those, giving higher priority to the main security and reliability issues pointed.

**Lessons Learned**

One of the most valuable lessons from using SonarQube was its ability to uncover subtle "code smells" that are easily missed during manual review. The tool was particularly helpful in identifying thread-safety issues, string literal duplications, and potential security vulnerabilities in logging practices. While some code smells remain, the critical issues affecting security and reliability have been completely resolved.

Important Note: Due to the tight project timeline, I did not have enough time to address all the issues reported by SonarQube. The current state reflects a functional codebase that has passed quality checks, but a thorough refactoring pass is necessary to resolve the identified maintainability issues to ensure the long-term health and stability of the project.

Below you can find the summary report given by Sonar which outlines all the issues mentioned above:

A screenshot of a computer

AI-generated content may be incorrect.

## Continuous integration pipeline

The Continuous Integration pipeline for this project is fully implemented and consists of two distinct, parallel jobs that are triggered on every push or pull request. This setup ensures both code quality and functional correctness are validated automatically.

1. **End-to-End Tests Job**

This job simulates a real user experience by testing the entire application stack, from the frontend to the backend API.

* Purpose: Validate the integrated system's functionality through automated browser-based tests.
* Steps:
  + Environment Setup: Checks out the code and sets up the necessary environments, including JDK 17 for the backend and Node.js for the frontend.
  + Application Build & Start:
    - The backend Spring Boot application is built and started on its default port.
    - The frontend application is built and served on a separate port.
* Test Execution:
  + The Playwright testing framework is installed.
  + A suite of end-to-end (E2E) tests is executed. These tests use an automated browser to interact with the running frontend, which in turn makes requests to the running backend, verifying the complete flow.
  + Screenshots are captured during test execution for debugging purposes if any tests fail.
  + Validation: The job passes only if all E2E tests succeed, ensuring that new changes do not break the core user workflows.

A screenshot of a computer

AI-generated content may be incorrect.

1. **SonarQube Analysis Job**

This job is dedicated to static code analysis and quality assurance.

* + Purpose: To analyze the backend Java codebase for potential bugs, vulnerabilities, code smells, and to measure test coverage.
  + Steps
    - Environment Setup: The job initializes by checking out the source code and setting up JDK 17, which is required to compile the Spring Boot application.
    - Dependency Management: It caches Maven packages to speed up subsequent builds and then executes the `mvn clean verify` command. This step compiles the code, runs all unit tests, runs all integration tests, and triggers the SonarQube analysis using the configured plugin.
    - Analysis & Reporting: The results of the static analysis are sent to the SonarQube server for detailed reporting and tracking of code quality metrics over time.

A screenshot of a computer

AI-generated content may be incorrect.

By implementing these two parallel jobs, the CI pipeline provides comprehensive feedback: the SonarQube job ensures code health and maintainability, while the End-to-End Tests job guarantees that the application functions as expected from a user's perspective.

# References & resources

Project resources

|  |  |
| --- | --- |
| **Resource:** | **URL/location:** |
| Video demo | <https://github.com/martacruzz/tqs-hw1/blob/main/doc/monos-vid.mov> |
| QA dashboard (online) | <https://sonarcloud.io/project/overview?id=martacruzz_tqs-hw1> |
| CI/CD pipeline | <https://github.com/martacruzz/tqs-hw1/actions> |

Reference materials

1. **Framework & Library Documentation**
   * [Spring Boot official documentation](https://docs.spring.io/spring-boot/index.html)
   * [React official Documentation](https://react.dev/)
   * [Playwright](https://playwright.dev/)
   * [Cucumber](https://cucumber.io/docs)
   * [Gatling](https://gatling.io/docs/)
   * [H2 Database](https://www.h2database.com/)
2. **Testing & Quality Assurance**
   * [Mockito](https://site.mockito.org/)
   * [AssertJ](https://assertj.github.io/doc/)
   * [SonarCloud](https://docs.sonarcloud.io)
3. **CI/CD & DevOps**
   * [GitHub Actions](https://docs.github.com/en/actions)
   * [Docker](https://docs.docker.com/)
4. **Project-Specific Resources**
   * [Monos Source Code Repository](https://github.com/martacruzz/tqs-hw1)