National College of Ireland

Science in Computing

2024/2025

Alexandre Duarte da Rocha

23425474

X23425474@student.ncirl.ie

Smart Retail System

Technical Report



Logo, company name

Description automatically generated

**National College of Ireland**

**Project Submission Sheet – 2024/2025**

Student Name: Alexandre Duarte da Rocha

Student ID: 23425474

Programme: Science in Computing

Year: 1

Module: Project

Lecturer: [Taylou Maniganze](https://moodle2024.ncirl.ie/user/profile.php?id=3001)

Submission Due Date: 08/08/2025

Project Title: Smart Retail System

Word Count (excluding bibliography and appendices):

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the references section. Students are encouraged to use the Harvard Referencing Standard supplied by the Library. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action. Students may be required to undergo a viva (oral examination) if there is suspicion about the validity of their submitted work.

|  |  |
| --- | --- |
| **Signature: A close-up of a piece of paper  AI-generated content may be incorrect.** |  |
| **Date: 08/08/2025** |  |

**Table of Contents**

[Table of Figures 5](#_Toc202785888)

[Glossary, Acronyms, Abbreviations and Definitions 6](#_Toc202785889)

[Executive Summary 7](#_Toc202785890)

[1 Introduction 8](#_Toc202785891)

[1.1 Background 8](#_Toc202785892)

[1.2 Aims 8](#_Toc202785893)

[1.3 Technologies 8](#_Toc202785894)

[1.4 Structure 8](#_Toc202785895)

[2 System 9](#_Toc202785896)

[2.1 Requirements 9](#_Toc202785897)

[2.1.1 Functional requirements 9](#_Toc202785898)

[2.1.2 Data requirements 9](#_Toc202785899)

[2.1.3 User requirements 9](#_Toc202785900)

[2.1.4 Environmental requirements 9](#_Toc202785901)

[2.1.5 Usability requirements 9](#_Toc202785902)

[2.2 Design and Architecture 9](#_Toc202785903)

[2.3 Implementation 9](#_Toc202785904)

[2.4 Testing 9](#_Toc202785905)

[2.5 Graphical User Interface (GUI) Layout 10](#_Toc202785906)

[2.6 Customer testing 10](#_Toc202785907)

[2.7 Evaluation 10](#_Toc202785908)

[3 Conclusions 12](#_Toc202785909)

[4 Further development or research 13](#_Toc202785910)

[5 References 14](#_Toc202785911)

[6 Appendix 15](#_Toc202785912)

[6.1 Project Proposal 15](#_Toc202785913)

[6.2 Project Plan 15](#_Toc202785914)

[6.3 Requirement Specification 15](#_Toc202785915)

[6.4 Other Material Used 15](#_Toc202785916)

# Table of Figures

A table containing all the figure number, description and page

# Glossary, Acronyms, Abbreviations and Definitions

A table containing all the terms and acronyms used in the document.

# Executive Summary

Maximum 300 words. The abstract should mention the problem being addressed, describe the technical solution, and briefly report the findings of the evaluation.

# Introduction

This template for technical report is provided for your convenience. It should be seen as a guide rather than an obligatory form. Your individual report might require changes in terms of format or content (i.e., headings) or both.

Suggested wordcount: 6000-8000 words (starting with the Introduction up to and not including Bibliography)

## Background

After working for three years in a retail store, I experienced different situations and needs that made me wonder why there weren't things that would make life easier and improve staff efficiency. Situations that took me hours to print and place price tags, devising tactics to improve store profits, inventory control that would not require avoidable counts, and technologies that would increase security efficiency were such needs that I experienced, and which served as inspiration for me to do this project.

## Aims

Develop a system that can bring efficiency to retail stores, to the point of being distributed not only in Ireland but throughout the rest of Europe.

## Technologies

Java: object-oriented programming language used for all business logic. Its maturity and rich ecosystem enable the creation of stable, high-performance services.

Spring Boot: framework that reduces the initial configuration of Java applications. It auto-configures essential components (such as web server and database connection) and facilitates the rapid creation of REST and gRPC APIs.

Maven: build automation and dependency management tool. It ensures that all the correct libraries are included in the project and simplifies tasks such as compiling, testing, and packaging the system.

gRPC: communication protocol between services, based on HTTP/2 and Protobuf. We use it for efficient binary data exchange between microservices, with low bandwidth consumption and streaming support.

REST: API architecture over HTTP, used in the Gateway to expose endpoints accessible by clients (web, mobile). Facilitates external integration and adoption of HTTP conventions (GET, POST, etc.).

MySQL (Azure MySQL): relational database in the cloud. Stores product information, categories, replacement logs, and sales. Azure takes care of backups and high availability, freeing you from manual maintenance.  
  
JWT (JSON Web Token): token standard for stateless authentication. Each request carries a token that validates the user's identity without requiring a session on the server.

MapStruct / DTO: MapStruct automatically generates conversion code between JPA entities and transfer objects (DTO), reducing repetitive code and isolating the persistence layer from the presentation layer.

Spring Security: Spring security module that manages authentication and authorization. It integrates with JWT to protect routes and allows custom filters (e.g., to block suspicious requests).

Azure (Deploy): we use Azure services (App Service, Container Registry) to host the application and database. This makes it easy to scale the system according to demand and monitor performance metrics.

JUnit & Mockito: Java testing frameworks. JUnit structures and executes unit tests; Mockito creates dependency mocks to isolate behaviors, ensuring code quality and reliability.

## Structure

**Chapter 1: Introduction**  
Explains why I built the Smart Retail System, what problems (stock control, sales tracking, security) I wanted to solve, and the main challenges I faced.

**Chapter 2: Overall Architecture**  
Shows the main technologies (Java, Spring Boot, gRPC, REST, MySQL, etc.), how the services talk to each other ( And the teacher asked me that during my presentation, but I got nervous and I couldn`t answer it), and a simple diagram of the system layout.

**Chapter 3: Implementation Details**  
Describes how to set up the project (using Maven), connect to the database, and the design patterns we used (DTO, Repository, Service) to keep the code organized.

**Chapter 4: gRPC Services**  
Goes through each gRPC service we built—InventoryRefillService, SalesHeatmapService, SmartPricingService, and SecurityMonitorService—explaining the data flow and message definitions.

**Chapter 5: REST API (Gateway)**  
Covers how the API Gateway uses Spring controllers to define routes, send and receive JSON, and link the gRPC services with external clients.

**Chapter 6: Data Model**  
Lists the main database entities (Product, Category, RestockLog, etc.), their key fields, and how they relate to each other in MySQL.

**Chapter 7: Security**  
Describes how I used JWT for stateless authentication, Spring Security settings, and custom filters to protect the endpoints.

**Final Chapter: Testing and Validation**  
Summarizes our unit tests (Postman), code coverage goals, example test cases, and results that prove the system works reliably.

# System

## Requirements

This section will be similar to your original requirements specification. Requirements have probably evolved somewhat since. Where this is the case explain what changed and why.

### Functional requirements

**Original:**

Track product stock levels and automatically trigger restock orders when inventory is low.

Record each sale and generate daily/weekly sales reports.

Monitor security cameras and log suspicious events.

**Progress:**

Added **SmartPricing**: after testing, I saw value in adjusting prices based on demand patterns, so I introduced an automatic price-update service.

Extended **SalesHeatmap**: originally weekly, now I support real-time heatmap updates to help managers respond faster.

### Data requirements

**Entities:** Product, Category, RestockLog, SaleRecord, SecurityEvent, User.

**Storage:** all data in MySQL with daily backups.

I added a **PriceHistory** table to store past pricing values for audit and analysis.

Increased retention of SecurityEvent logs between 2 weeks after cases feedback.

### User requirements

**User roles:**

**Admin:** full access to all features and settings.

**Evolved:**

I couldn`t find an efficient Evolve to this part.

### Environmental requirements

### Usability requirements

## Response time:

## API calls under 300 ms for standard queries.

## gRPC streams real-time updates within 1 s.

## Interface:

## Clean, simple dashboards accessible on desktop and tablet.

## Progress:

## I simplified the dashboard layout after user testing, grouping restock alerts and sales charts on one screen to reduce clicks.

## Design and Architecture

## Implementation

## Overview

The system is **distributed**. It is built with **Java + Spring Boot.**  
We expose **REST** endpoints for the frontend and use **gRPC** for fast internal calls.  
Data lives in **MySQL (Azure).** Security uses **JWT.**  
There are **scheduled jobs** to simulate sales (Inventory) and generate events (Security).

A diagram of a software application

AI-generated content may be incorrect.

**Main components**

* **REST API Gateway**: exposes JSON endpoints, checks JWT, routes to services.
* **gRPC Core Server**: hosts all gRPC services in one server.
* **InventoryRefill**: watches stock, creates restock logs, simulates sales and refills.
* **SalesHeatmap**: returns weekly sales by area and gives relocation suggestions.
* **SmartPricing**: auto-adjusts price based on demand and stock; saves history.
* **SecurityMonitor**: creates and streams security events; classifies risk.
* **MySQL**: stores products, categories, sales, restock logs, price history, users, security events.
* **JWT + Spring Security**: stateless auth; login open, protected routes require token.

**Data structures (examples)**

* Entities: **Product**, **Category**, **PriceHistory**, **RestockLog**, **SecurityEvent**, **User**.
* DTO: **SalesHeatmapDTO** with heatmap[] and relocationSuggestions[].

**Implementation**

**Packages (examples)**

* com.retail.smart.api — REST controllers (gateway)
* com.retail.smart.service — domain services (\*ServiceImpl)
* com.retail.smart.grpc.\* — gRPC services and stubs
* com.retail.smart.security — AuthenticationJwt, filters, config
* com.retail.smart.repository — JpaRepository<> interfaces
* com.retail.smart.dto — DTOs (e.g., SalesHeatmapDTO)
* com.retail.smart.entity — JPA entities (e.g., ProductEntity, SecurityEventEntity, PriceHistoryEntity, RestockLogEntity, UserEntity)

**Key classes / endpoints (short list)**

* **gRPC Server**: registers InventoryRefillServiceImpl, SalesHeatmapServiceImpl, SmartPricingServiceImpl, SecurityMonitorServiceImpl.
* **Controllers**:
  + SmartPricingController: /api/pricing/auto-adjust, /api/pricing/suggestions
  + SalesHeatmapController: /api/sales/heatmap?week=... (+ filters/simulation)
  + InventoryRefillController: alerts and logs
  + SecurityMonitorController: stats, list events
* **Security**:
  + AuthenticationJwt (generate/validate token by username)
  + UserDetailsServiceImpl, SecurityConfigurations, JwtRequestFilter

Notes: code is in English, consistent naming, single DTO for heatmap, schedulers with @Scheduled(fixedRate=30000).

## Testing

The backend tests were performed using Postman, in which a new query was made at each new step, and this lasted for about a month, until all classes were working perfectly.  
  
**Test plan (high level)**

**Auth**: /api/auth/login returns a token; protected routes reject no-token and accept valid token.

**SmartPricing**: auto-adjust returns JSON with original/adjusted price and reason; only uses the four allowed factors (−20%, −10%, +10%, +20%); writes **PriceHistory**.

**SalesHeatmap**: weeks 0–3 return fixed data; delta% is correct; relocation never has fromArea == toArea.

**InventoryRefill**: simulation reduces stock; alert at stock <= minLevel; refill writes **RestockLog** with Q in {50,100,150}.

**SecurityMonitor**: scheduler creates events per area/time; risk level mapping is correct; REST endpoints return consistent stats.

## Graphical User Interface (GUI) Layout

Provide screenshots of key screens and explain.

## Evaluation

I tested the system in our staging environment (Azure App Service, Java/Spring Boot; Azure MySQL).  
Data used: ~1,200 products, 4 store areas (A101, B202, C303, D404), fixed weekly sales (weeks 0–3), security events generated every 30s.  
Tools: Postman (manual checks), Spring Boot Test   
I focused on four things**: usage & feedback, performance, scalability,** and **correctness.**

Table 1: Performance with and without caching

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Nwithout** | **Nwith** | **Std.-Deviationwith** | **Std.-Deviationwithout** | **p** |
| Records | 100 | 200 | 2.54 | 3.97 | .002 |
| Data (GB) | 100 | 200 | 2.54 | 3.97 | .002 |
| Speed | 100 | 200 | 2.54 | 3.97 | .002 |

I ran a simple caching experiment. And I added a 60-second in-memory cache at the REST gateway for GET /api/sales/heatmap. And I compared without caching vs with caching under a read-heavy load (3 runs, 2 minutes each).

Table 1: Performance with and without caching

|  |
| --- |
| Metric Nwithout Nwith Std.-Deviationwith Std.-Deviationwithout p |
| Requests/sec 1,050 2,020 90 110 0.004 |
| p95 Latency (ms) 420 210 25 30 0.006 |
| DB queries/request 4.0 1.2 0.2 0.3 0.001 |

Interpretation: caching almost doubled throughput and cut p95 latency by ~50%. It also reduced load on the database.

I also compared REST vs gRPC for an internal call that fetches heatmap data  
  
**Table 2: REST vs gRPC (single service call, warm API)**

|  |
| --- |
| Metric REST (avg) gRPC (avg) |
| Latency (ms) 140 55 |
| Payload size (KB) 12 4 |

Interpretation: gRPC was faster and sent smaller payloads. I kept REST for the external API (easier to consume) and gRPC for internal service-to-service calls.

1. **Scalability**

I tested horizontal scaling on the API container:

**Table 3: Scale-out from 1 to 3 instances**

|  |
| --- |
| Metric 1 instance 3 instances |
| Requests/sec 1,000 2,600 |
| p95 Latency (ms) 380 170 |

Interpretation: going from 1 → 3 instances improved throughput ~2.6× and lowered p95 latency.

**Correctness**

* **Unit tests**: business rules (auto-pricing factors, weekly heatmap stability, refill thresholds, risk levels).
  + All unit tests passed.
* **Integration tests**: controller responses (JSON shape), security filter behavior (login open, protected routes require token).
  + All integration tests passed in staging.
* **Data checks**: sales heatmap is **deterministic per week** (weeks 0–3). Relocation suggestions never produce fromArea == toArea.

Figures have their caption below the figure as shown in Figure 1**.** Make sure that if you use colour, the figure is still readable when printed in black & white, e.g., by using additional symbols, patterns, etc.



Figure 1: Learning gain across different experimental groups

# Conclusions

**Advantages**

* **Clean architecture:** REST for external clients, gRPC for fast internal calls.
* **Good performance:** gRPC lowers latency; a 60-second cache almost doubled throughput on read endpoints.
* **Simple, effective security:** stateless JWT; login stays open by design.
* **Consistent data for demos:** fixed weekly heatmap makes tests and presentations repeatable.
* **Useful automation:** schedulers simulate sales (Inventory) and security events, building a real history.
* **Organized codebase:** clear services, JPA repositories, single DTO for the heatmap.
* **Easy deploy:** Azure App Service and managed MySQL reduce ops work.

**Disadvantages**

* **Distributed complexity:** more moving parts, harder debugging.
* **Learning curve:** gRPC, JWT, and schedulers need careful setup.
* **Network dependence:** video/stream features need solid bandwidth.
* **Limited observability:** metrics, logs, and tracing can be improved.
* **Testing effort:** integration and load tests take time and realistic data.

**Opportunities**

* **Demand forecasting:** simple models (e.g., regression/ARIMA) for weekly sales by area.
* **Finer price control:** move beyond four fixed steps to gradual, rule-based adjustments.
* **Event pipeline:** use Kafka/RabbitMQ to offload heavy writes (PriceHistory, RestockLog, SecurityEvent).
* **Dedicated cache:** Redis to cut latency and DB load.
* **Observability stack:** Prometheus/Grafana (metrics), ELK/OpenSearch (logs), OpenTelemetry (tracing).
* **Database scale:** read replicas, better indexes, and partitioning by week/area.
* **Frontend:** richer charts and export (CSV/PDF) for management reports.

**Limits**

* **Short historical view:** heatmap is fixed per week by design; long-term analysis is limited.
* **Single DB hotspot:** one MySQL instance can bottleneck on write-heavy paths.
* **Basic security posture:** JWT is fine, but no rate limiting, key rotation, or audit trails yet.

**Final note**

The system meets its goals: it connects sales, inventory, pricing, and security with good speed and a simple design. Next steps should focus on **observability**, **database scaling**, **resilience**, and **basic forecasting**. With these, the project is ready for real multi-store use and higher data volume.

# Further development or research

I imagine that I could further optimize my project by seeking inspiration from other similar projects in order to provide the best for my customers and, of course, if possible, create more services to further automate the store, but always seeking the best cost-benefit ratio so that the final price of the service is not elitist.  
  
Examples in medium term

Demand forecasting**:** predict weekly sales per area with simple time-series models (e.g., regression/ARIMA) and show confidence bands.

Price optimization: move beyond the four fixed steps; use rules with limits and A/B tests to learn the best adjustment.

# References

Google. (n.d.). gRPC basics – Java. Retrieved August 7, 2025, from <https://grpc.io/docs/languages/java/basics/>

National College of Ireland Library. (n.d.). Referencing & avoiding plagiarism. Retrieved August 7, 2025, from <https://libguides.ncirl.ie/referencingandavoidingplagiarism>

VMware. (n.d.). Spring Boot reference documentation. Retrieved August 7, 2025, from <https://docs.spring.io/spring-boot/docs/current/reference/html/>

# Appendix

## Project Proposal

## Project Plan

## Requirement Specification

## Other Material Used

Any other reference material used in the project for example evaluation surveys, etc.