Dissertation

CO3201 Computer Science Project - Virtual Vending Machine Webapp Project Report

May 2025

Issa Aboobaker (ia252)
SCHOOL OF COMPUTING AND MATHEMATICAL SCIENCES,
UNIVERSITY OF LEICESTER

Declaration

All sentences or passages quoted in this report, or computer code of any form whatsoever used and/or submitted at any stages, which are taken from other people's work have been specifically acknowledged by clear citation of the source, specifying author, work, date and page(s).

Any part of my own written work, or software coding, which is substantially based upon other people's work, is duly accompanied by clear citation of the source, specifying author, work, date and page(s).

I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this module and the degree examination as a whole.

Name: Issa Aboobaker

Signed: 

Abstract

This report documents the design, development, implementation, testing and evaluation of the Virtual Vending Machine Web Application a full-stack web-based simulation of a modern vending machine. The project was created in response to the lack of professional, feature-frequent vending machine simulators currently available, aiming to deliver a more complete and engaging user experience while also demonstrating a strong technical architecture.

The system was built using Java with Spring Boot for the backend and a combination of HTML, CSS, JavaScript and Thymeleaf rendering for the frontend. It supports core vending machine functionality including product browsing, cart-based selection, coin and card payments, receipt generation and transaction management. Additional features include analytics, admin dashboard controls, auto-restocking and a Smart Recommendations algorithm based on a multi-factor scoring model. A clear focus was placed on UI design and interactivity, with all interfaces built to be responsive, visually consistent and user-friendly.

This report outlines the full development lifecycle of the system, starting from background research and design planning, through to feature implementation, system testing, evaluation and final reflection. Key development decisions are supported by literature and existing system analysis. Testing was performed through integration testing, exploratory use, Postman API calls and selected JUnit unit tests.

The project successfully meets its original aims by delivering a complete and fully tested simulation, with clear justification for each technical and design decision. The system's modular architecture, modern payment integration, security handling and dynamic user features demonstrate strong real-world potential and provide opportunities for future expansion or commercial deployment. The report concludes with a critical evaluation of the system's strengths, limitations and overall performance, supported by targeted recommendations for improvement and reflection on the development process.

Table of Contents

1. Introduction

1.1 Overview

The idea behind this project was to design and develop a fully functional Virtual Vending Machine Web Application that replicates the functionality of a real-life vending machine. The goal was to simulate a system where users can view available products, select multiple items, add them to a virtual cart, and complete a purchase using animated coin inputs, all within a visually appealing and interactive webpage user interface.

The core purpose of this project is to bring the convenience and automation of a vending machine into a more usable and accessible virtual space. This concept has multiple real-world applications such as: online product selling platforms, remote kiosks in shops for products and services and digital training platforms for educational purposes.

Finally, it also serves as a functional prototype that could be extended to control actual hardware components like motors and sensors in physical vending machine units.

The project uses a full-stack Java Spring Boot backend and HTML/CSS/JavaScript frontend. The system interacts with a MySQL database to manage product stock, transactions, user accounts and other persistent data. Use of Spring's Model-View-Controller (MVC) pattern ensures a clean separation of concerns, and RESTful APIs allow clear communication between the frontend and backend.

One of the main goals was to create a system that doesn't just look like a vending machine but also behaves like one right down to how products are selected, how coins are inserted, and how change is dispensed if required. The inclusion of extra features such as receipt generation, transaction history viewing, stock level monitoring and others makes this more than just a simple simulation: it is a polished, extensible software system that could easily be connected to other systems to create a complex selling or simulation application. Through researching into other similar systems, I found that while they tend to contain solid functionality, their user interfaces are often lacking - they tend to be outdated, blurry or laggy. I set out to not only add additional features to my own system but to hugely focus on making a significant improvement to the user interface, aiming for smoother performance, a more appealing design and a more engaging, interactive user experience.

The project also aimed to strengthen my own technical skills, including full-stack web development, database design, backend architecture, user interface design, documentation, planning and testing. Over the course of the project development, I faced various design decisions and challenges, from implementing realistic user interactions to managing backend logic for handling transactions and stock updates in real time.

This report explains the complete journey of the project, starting from the initial research and requirements gathering, through system design and implementation, to testing and finally a critical evaluation. It reflects not only the technical work I have done but also how feedback from earlier assessment stages i.e. the Interim Report and the Interview have shaped my final project and this report.

My Aims and Objectives below have been taken from my Interim Report but based on the feedback have been altered and improved to be more specific, become relevant to implementation and show a distinct difference between each other.

1.2 Aims

The aims represent the overall intentions of the project and guide the direction of development. These are high-level goals which reflect overall what the project sets out to achieve:

- To develop a full-featured, web-based vending machine simulation that replicates the user experience of a physical vending machine.
- To create a smooth, interactive, and visually engaging user interface that improves on the flaws found in existing virtual vending systems.
- To strengthen my skills in full-stack web development using Java, Spring Boot, Thymeleaf with HTML, CSS, JavaScript, and MySQL.
- To produce well-documented, maintainable code that follows modern software engineering practices and allows for potential re-use in other future projects.
- To design software architecture that is clean, modular, and scalable, allowing for future feature additions or real-world hardware integration.
- To reflect on feedback from development milestones and apply it throughout the project lifecycle to improve both process and outcome.

1.3 Objectives

The following objectives were set to achieve the aims above. They are specific, actionable, and measurable steps used to guide development and track progress:

1. Analyse existing vending systems (physical and virtual) to identify common features, usability issues, and areas for improvement.
2. Define and document a clear, detailed set of functional and non-functional requirements for the application.

3. Create a structured development plan, including a Gantt chart with clearly defined phases, deadlines, and review checkpoints.
4. Develop the backend using Spring MVC with Java, implementing separated logic for user accounts, product management, transaction handling, payment processing, and stock updates.
5. Design and build the frontend using HTML, Thymeleaf, CSS, and JavaScript — including animations for coin insertion and real-time display updates.
6. Develop an interactive cart system that supports multi-item selection, total calculation, and change dispensing.
7. Integrate REST APIs to connect the frontend and backend, ensuring smooth real-time interaction without requiring full-page reloads (AJAX).
8. Incorporate advanced features such as optional receipt printing, viewing of transaction history, and dynamic UI feedback.
9. Maintain clean code practices throughout the project by using version control, meaningful comments, modular design, and proper file organization.
10. Conduct thorough testing at multiple stages: unit testing of backend logic, UI testing for the frontend, and full system integration testing before finalisation.

2. Background and Literature Review

Overview

This project focuses on developing a full-stack Virtual Vending Machine web application. To help the design and implementation, I examined existing vending systems, both physical and virtual, and relevant user interface (UI) principles. This review presents my findings from academic literature, articles and real systems, directly linking them to the project's development choices.

Traditional and Virtual Vending Machine Concepts

The development of vending machine systems has evolved over time especially in recent periods, driven by both mechanical improvements and the growth of digital integration. Traditional vending machines have always traditionally functioned based on simple input-output principles using electromechanical systems: rotating coils, infrared item detection and cash validation sensors to complete transactions. These systems are reliable but often limited in user interaction and adaptability, especially in such a tech-driven world. As one study explains: “conventional vending machines are typically standalone, offering basic functionality without interactivity or real-time intelligence” **[1]**.

For as long as I can remember vending machines have served as an efficient form of automated self-service, dispensing products with minimal human interaction. As the paper details: “the vending machine is one of these automated machines which supply needed things to the customer” **[1]**. Traditionally, these machines operate in a simple sequence: users insert coins, select an item and simply collect the item - a system heavily praised for its efficiency. It is stated that “any technology requiring more user interactions than these simple three steps will be hardly adopted in vending” **[2]**. From this I understand the importance of simplicity of payment methods for my system.

Beyond their simplicity, vending machines offer numerous benefits. They require no staff, operate consistently 24/7, save customer time and reduce costs. As explained in the source paper: vending machines have “many benefits” such as “man power is not needed”, “ offer “flexibility in time”, “saving time”, “reducing labor cost”, “increasing profitability” and therefore “vending machines are used commonly worldwide” **[1]**. These traits make them attractive in environments that demand scalability with limited resources. Based on this we can further enhance these vast benefits by introducing a new fully virtual system that requires less power to run, no hardware and is more accessible ‘worldwide’.

However, vending is now evolving beyond its initial constraints. In the rise of changing consumer behaviour, especially after COVID-19, demand for contactless, digital retail services has grown. As the literature shows: “Most customers want unmanned retail models and cashless payment methods because customer behaviour has changed” **[1]**. This relates to protocols being introduced during the pandemic such as social distancing and how they have forced people to become accustomed to new ways of making purchases (minimal contact with other customers/staff). This was initially enforced predominantly by online shopping but now as we have adjusted back to a non-restricted lifestyle, there is a rising demand for new and innovative ways to make purchases. These changes have not just influenced how vending machines are used, but also what technologies are expected from them. From this, the ideas demonstrate a need for a modern and technological innovation to enhance vending machines to suit customers’ behavioural needs and expectations and have therefore inspired my project idea.

Recent advancements show vending machines increasingly implementing smart technologies such as machine learning, data analytics and cloud systems. These innovations make it possible to “access real-time data collection, increase sales, make operation more efficient, and supply things to customer desire” **[1]**. Combined with trends on the Internet of Things (IoT) and contactless payments, vending machines are no longer just simple coin triggered mechanical product dispensers; they’re becoming fully connected smart systems. The report supports this shift, stating that “vending machine systems are required to implement using IoT with machine learning, and artificial technologies to satisfy the customer preferences.” **[1]**. Based on this requirement, I am aiming to implement a ML concept-based algorithm dependent feature (Smart Recommendations).

The industrial context also further emphasises this shift. Globally, vending machine usage has significantly grown, reaching an estimated value of “US$134.4 billion” in 2020, predicted to “reach US$146.6 Billion in 2027” **[1]**. This giant advancement reflects both technological advancement and increased customer reliance on contactless, fast self-service models.

Despite all this there are still challenges within this sector, predominantly cost barriers with credit card integration. As a different academic paper states: “vending machines accepting credit cards are only found in frequented locations like shopping centers and airports, where high sales per machine may compensate the incurred costs” **[2]**. The hardware for contactless card readers adds to running costs, preventing smaller organisations and businesses from adopting smart payment options for vending machines. This further indicates a need for a virtual solution to reduce hardware costs and gives me the motivation to implement an alternative, smarter payment method for my system: Stripe card payments.

Self-service systems also face usability difficulties. As another source shows, these modernised systems offer consistent availability and convenience, but "generally do not allow for customisation of the service nor addressing special cases that may arise" **[3]**. This highlights the importance of careful design planning and user case considerations. Based on this I aim to add some customisation elements (even if simple, still effective) and ensure robust handling of special events that may be occur. This is a key advantage of a virtual system as opposed to a physical one, because a virtual platform allows easier implementation and more control of customisable and personalised elements.

Regardless, the benefit of smart vending is clear. With the ability to remotely monitor stock, adapt to user preferences, and potentially integrate with mobiles, these systems represent the future of automated retail. As vending machine design evolves, it is essential that systems find the perfect balance between technological innovation and the original simplicity that allowed vending machines to become globally recognised.

After deeply analysing how traditional vending machines work and investigating the key concepts of smart vending and virtual systems, I have reinforced the need for my proposed system and along the way identified key features that I need to replicate from existing physical machines and also additional features restricted to virtual systems only that I aim to implement into my new system to be able present it as a usable and effective platform and also justify the desire to replace physical vending machines (or combine them) with a virtual vending machine.

Existing Virtual Vending Machine Systems

I researched, tested and evaluated current virtual vending simulations, analysing pros and cons from each to shape my own development plan. Free to use webapps were difficult to find as there aren't many, but I managed to find a few:

- **"PlayHop's Vending Machine Simulator" [4]**: Offers visual item display and coin insertion with dynamic balance updates. However, while the functionality is complex, it deeply suffers from performance issues and a poor interface. My design aims to retain similar functionalities and build on them further while also ensuring a smoother, more responsive user experience through the frontend.
- **"Dennis Roberts' Virtual Vending Machine" [5]**: Features an interactive design with some useful customisation features (e.g. selecting shape, size and colour) and simulates a proper vending flow, something I aim to implement similarly in my system. However, it lacks the realism of a real vending machine with applications: there is no keypad, payment system and no actual products, just shapes that get dispensed. Also, the UI is quite blurry, and performance is poor, slow and laggy. I aim to improve on all of this for

my system, making it real with actual product selection and design a fully visually appealing and responsive UI.

- **“ArcadeSpot's Surprise Eggs Vending Machine” [6]**: Although it is gamified and designed for children, it still implements some good functionality. There is a keypad to enter a code to select the desired products and a simulated coin payment system (both strong features that I am also implementing). From these I have gained some inspiration in how to create my vending machine main features. However, given that system is just a game, it lacks a realistic UI and professionalism in the layout which is something that I aim to improve on. Additionally, the overall view of the page is overly animated, but I aim to make mine clear and readable. Finally, it only allows vending one item at a time, but I want to introduce a cart system where users can hold multiple items and then purchase them all at once.

From my findings I have concluded that there a few existing virtual vending machine simulations, however most of these are deeply flawed in certain key areas (UI, functionality, aesthetic). Based on this I intend to create a web app that demonstrates advanced functionality through various features, both standard vending machine ones (product view, keypad, cart) and additional ones (transaction management, auto restocks, admin controls). I also am placing a strong focus on the UI design, as none of the existing systems have an interface that is totally responsive, smooth and professional. The existing systems are low-level simulations mostly for experimental or gaming purposes, but I am creating a high-level realistic simulation that could potentially be commercialised for real world applications.

3. Requirements Analysis

After reviewing my improved project objectives and researching similar systems, I identified a clear set of requirements necessary for the successful development of the Virtual Vending Machine Web Application. These requirements have been grouped into three distinct categories: Functional Requirements, Non-Functional Requirements, and Project Constraints and extra sections for assumptions made and the criteria for measuring requirement success have been added below. Each requirement has been directly formulated based on the system's intended goals and user expectations and was also derived from the project's core aims and objectives (see Sections 2.2 and 2.3), ensuring a consistent development direction throughout. In similarity with the aims and objectives, these requirements were taken from my Interim Report and changed/improved based on the relevant assessment feedback.

3.1 Functional Requirements

Functional requirements describe the core features and behaviours that the system must contain to perform its intended operations. These were driven by the aim to simulate the experience of a real vending machine in a virtual environment and ensure an interactive user experience.

1. **Product Display**
    - The system must display all available products in a grid layout, each showing its name, price, and stock availability.
    - Products must update dynamically to reflect stock changes after transactions.
2. **Product Selection & Cart System**
    - Users must be able to click on a product to add it to a virtual cart.
    - Multiple different or same products can be selected in one session.
    - A visual cart area should display selected items with real-time updating.
3. **Cart Management**
    - Users must be able to remove items from the cart and alter quantities before proceeding to payment.
    - The total cost must update in real time when changes are made.
4. **Payment Processing**
    - The application must simulate coin insertion with animations and effects.

- As coins are added, the remaining amount to pay should dynamically update in the display area.
- Overpayment must be handled by triggering the change dispensing logic.

5. **Change Dispensing**
    - When users overpay, the system must automatically calculate and simulate the dispensing of correct change using available coin denominations.
6. **Receipt Generation**
    - After a successful purchase, users must be given the option to generate, view and download a digital receipt.
    - The receipt must include item details, prices, total paid and change returned as well as have the user's username and the date and time written at the top.
7. **Stock Management**
    - Stock quantities must be updated in the database after each transaction.
    - Items that are out of stock should no longer be selectable and must be visually indicated.
8. **Interactivity and Real-Time Feedback**
    - The application should provide immediate UI feedback through animations and dynamic content (e.g. coin insertion animations, confirmation messages).
    - Users must be notified of key actions (e.g. "Item added to cart", "Payment complete", "Insufficient stock").
9. **User Interface Design**
    - The interface must be visually appealing and responsive, with a clear layout and accessible controls.
    - Styling must reflect a modern and slightly gamified vending machine aesthetic using detailed CSS.
10. **Transaction History (Extended Feature)**
    - Logged-in users must be able to view a history of previous transactions through their logged-in home page.
11. **Analytics Page (Extended Feature)**
    - Logged-in users must be able to access a page with visuals that display to them patterns/trends based on their transaction history.
12. **Product Recommendations (Extended Feature)**

- Users should be able to turn on Product Recommendations and see these visually in the Vending Machine UI.

13. Admin Controls (Extended Feature)

- Must Allow admins to log in and access a secured admin dashboard.
- Admins should be able to manually update stock for products and manage auto-stock update features (toggling the setting and setting thresholds).

3.2 Non-Functional Requirements

Non-functional requirements define the quality attributes and technical standards the system must meet to ensure performance, scalability, reliability, and maintainability.

14. Performance

- The web interface must load within 3 seconds under normal conditions.
- Dynamic operations (adding to cart, coin inputs, etc.) must reflect instantly on the UI without requiring full page reloads.

15. Scalability

- The system must be designed in a modular and extensible way, allowing for new features or additional product details to be added without changing the core architecture.

16. Error Handling

- The system must detect and gracefully handle errors such as:
  - Attempting to purchase out-of-stock items
  - Inserting insufficient funds
  - Backend failures (e.g. Database access issues)
- User-friendly error messages should guide the user towards corrective action.

17. Maintainability

- Code should be clean, commented, and structured using good coding practices (e.g. separation of concerns, reusable components).
- Git version control must be used properly and consistently throughout development.

18. **Security & Data Integrity**
    - User actions must not allow them to corrupt data or bypass security measures (e.g. duplicate purchases or skipping payment).
    - Input validation should prevent bad data entry.
    - Passwords should be encrypted/hashed before being stored in the database.

19. **Testing and Validation**
    - All modules should be tested:
        - Unit tests for backend logic.
        - Manual testing for UI and frontend interactivity.
        - Full system testing ensuring everything works across all ends, before finalisation.

3.3 Project Constraints

This project was subject to the following practical constraints which influenced key development decisions:

20. **Technology Stack**
    - Backend: Java with Spring Boot
    - Frontend: HTML with Thymeleaf, CSS, JavaScript
    - Database: MySQL
    - Build Tool: Gradle
    - Architecture: Model-View-Controller (MVC)

    These specific technologies were selected due to their suitability in full-stack web applications and are most consistent with my academic learning experience.

21. **Time Constraints**
    - The full development process had to be completed within the fixed academic term.
    - A detailed project timeline and Gantt chart were created to plan out stages of development, allowing time for testing, improvement and feedback integration.

22. **Development Environment and Tools**
    - Development was carried out using IntelliJ IDEA for code, GitLab for version control and a local MySQL connection.

- There was no requirement for deployment, so a local environment was maintained throughout.

23. Documentation

- A detailed project log was maintained throughout development. This includes progress tracking, issues encountered, design changes and reflections between development stages.

24. User Base Limitation

- The application is currently intended for simulation and demonstration purposes, not for public commercial use. This limited the scope of authentication and wide-scale features in the current setup.

3.4 Assumptions

To keep the project scope realistic, the following assumptions were made prior to development:

- Users would access the system on modern desktop browsers with JavaScript enabled.
- The system would be run locally during testing and demonstrations, without external hosting.
- Only one user would interact with the machine at a time during demonstration, although the system is still built to support multiple users.
- Users are familiar with standard UI layouts.
- Product data is hardcoded into the system and is example data (i.e. things like prices may not be realistic).

These assumptions allowed the system to remain manageable within the limitations of a single-developer academic project within a strict timeframe.

3.5 Success Criteria

Project success was evaluated based on both functional performance and the completion of the requirements outlined in the sections above. The key measurement points were:

- **Testing Outcomes**: Each feature was tested manually and through unit tests to ensure correct logic, smooth interaction and UI responsiveness.
- **Requirements**: The final system was assessed against the functional and non-functional requirements defined in Section 4.1 and 4.2. Successful completion of these requirements indicated that the system met its design goals.
- **Usability Demonstrations**: The application was evaluated through walkthroughs of realistic use-case scenarios including coin input, cart management, receipt generation as well as all the extended features.
- **Feedback Integration**: Changes made based on feedback from the Interim Report and Interview were incorporated into the final product and are reflected in the added features and improvements made to both the system and this report.
- **Objective Alignment**: Each original objective from Section 2.3 was addressed through implementation, testing or documentation.

These criteria allowed me to clearly assess whether the system met both technical and user-focused expectations. By comparing testing outcomes with original objectives and responding to feedback, the project demonstrates a full development cycle from concept ideas to evaluation. Full evidence of these success indicators is documented in **Section 7: Testing.**

4. System Design

The design of the Virtual Vending Machine Web Application was planned carefully to follow a clean and modular architecture which separates key system components and ensures scalability, maintainability and implementation of proper coding practices. The overall architecture follows the Model-View-Controller (MVC) design structure and integrates a RESTful approach to allow communication between the frontend and backend. This section outlines the major system components, the interactions between them and the thought process behind key design choices.

4.1 System Architecture Overview

At a high level, the system is split into three main layers:

1. **Frontend**: Built using HTML with Thymeleaf integrated in, CSS for styling and JavaScript for interactivity. This layer handles all user interactions, renders UI components and communicates with the backend via HTTP requests.
2. **Backend**: Created in Java, using Spring Boot as the framework. This layer contains the core business logic and handles incoming requests from the frontend through the appropriate controllers and services.
3. **Database**: A MySQL relational database used for securely storing and querying all project data such as user accounts, product details and transaction information.

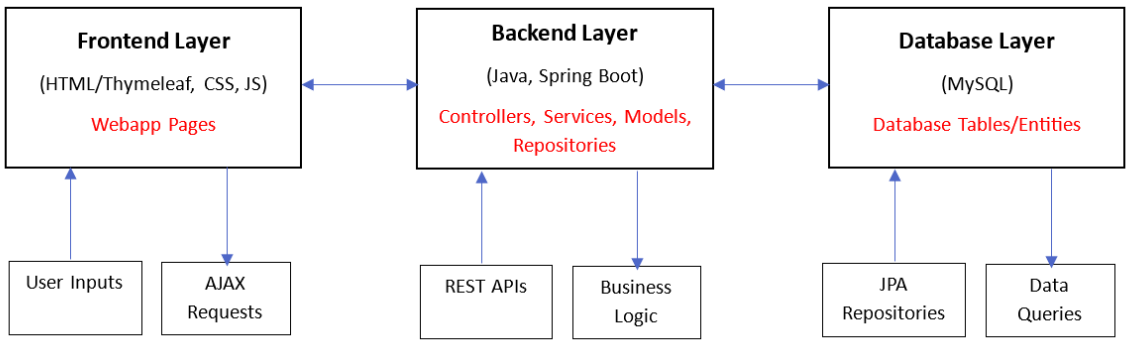
The architecture is specifically designed to facilitate a clear separation of concerns and allow each layer to be expanded independently. REST APIs allow asynchronous data transmission between frontend and backend, allowing real-time updates (e.g. dynamic cart totals, stock changes, and transaction displays).

In a typical flow: the user interacts with the UI -> UI sends/requests data to/from the backend via a POST/GET -> Backend processes logic and interacts with the database through repositories -> Response is sent back to frontend for real-time UI updates. All communication between frontend and backend happen asynchronously via AJAX to allow dynamic updates to the UI which don't require a full page reload.

This overall architecture provides the foundation for a responsive and scalable vending machine simulation.

Figure A1 – System Architecture Diagram

Overview of the system architecture, highlighting communication between frontend, backend, and the database.



4.2 MVC Design Pattern

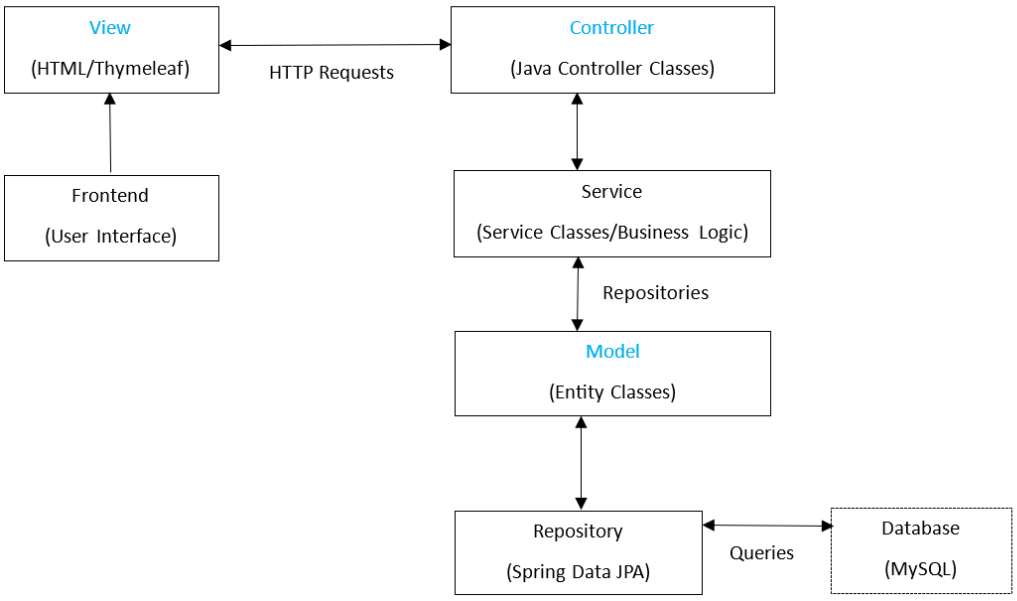
The application is structured around the Model-View-Controller pattern, which is core to Spring Boot. This pattern ensures a logical separation of data management (Model), UI rendering (View) and request handling (Controller).

- **Model**: Represents the data table objects used throughout the system (Product, User, Transaction). These are directly mapped to database tables via JPA annotations and used to move data between layers.
- **View**: Composed of HTML/Thymeleaf templates and responsible for creating the user interface and reflecting the current state of the application.
- **Controller**: Handles incoming HTTP requests from the frontend, processes data via service classes and returns the correct view or JSON response. For example, the 'TransactionController' manages transaction processes including purchase logic and transaction object creation.

This structure helped simplify debugging, testing and overall development, while also keeping the system maintainable and easy to navigate.

Figure A2 – MVC Pattern Diagram

Illustrates the implementation of the Model-View-Controller pattern in the application.



4.3 Component Breakdown

The backend code is divided into several key packages, each with a clear and specific purpose:

Controllers

Located in 'controller/', this package handles routing and frontend HTTP requests:

- 'LoginController', 'RegisterController': Handle user authentication
- 'NavigationController': Allows navigation between pages.
- 'MainController': Processes key features and requests from the main vending machine page.
- 'TransactionController': Manages all transaction related actions (e.g. creating transaction entries, fetching transactions and querying transactions).
- 'AdminController', 'AnalyticsController', 'ReceiptController': Handle specific features for the respective Admin, Analytics and Receipt pages.

Each controller method maps specific URL endpoints (e.g. /user, /filter) to different backend logic and data responses.

Services

Located in 'service/', service classes contain core business logic to act as helpers to controllers and as a bridge between controllers and repositories:

- ‘ProductService’: Creates and retrieves product entries updates and updates product stock.
- ‘TransactionService’: Handles payment logic and validation, saves transactions into the database and fetches transactions.
- ‘AnalyticsService’: Queries the database for transaction history and generates trend data for visualisations on the page.

By keeping additional business logic in service classes rather than controllers, the application follows a clear single-responsibility principle and remains easier to update and maintain.

Models

Located in ‘model/’, models represent the database entities:

- ‘Product’, ‘Transaction’, ‘TransactionProduct’, and ‘User’ represent the core data tables.
- These are mapped using JPA annotations and enable direct storage/retrieval from MySQL.

Models are used across both frontend (Thymeleaf variables, JavaScript) and backend (service classes and DTOs).

Repositories

Located in ‘repository/’, repositories extend Spring Data JPA interfaces:

- ‘ProductRepo’, ‘UserRepo’, ‘TransactionRepo’, ‘TransactionProductRepo’: Provide easy access to CRUD operations without needing to write SQL queries.
- Custom queries (for analytics and filtering) are also added into the repository classes where needed.

DTOs

Located in ‘dto/’, these Data Transfer Objects are used for easy grouping, storing and transferring of analytics data to be used for the visuals in the frontend:

- ‘SpendingTrendDTO’, ‘PurchaseFrequencyDTO’, ‘AnalyticsSummaryDTO’: help transfer processed data from backend to frontend in a cleaner format.
- Each DTO holds the necessary data for a specific visual on the Analytics Page.

Security

Located in 'security/', this layer configures authentication and access through Spring Security:

- 'SecurityConfig': contains the 'securityFilterChain' which defines access rules for users and additionally allows OAuth2 logins and a cookie based "Remember Me" authentication.
- 'UserDetailsServiceImpl': integrates with Spring Security to find the user's details based on the username/email they logged in with.
- 'CustomLoginSuccessHandler': contains logic for handling a successful login - routes users to appropriate dashboards and generates usernames for OAuth2 logins.

4.4 Database Design

The database was designed using MySQL and follows a relational model that maps clearly to the backend JPA entities. Each model class corresponds to a table in the database and uses annotations to define relationships (e.g. OneToMany, ManyToOne). The system's database schema includes four entities:

1. User: Stores user account data, including username, email and password (encrypted). It also stores the role (user/admin). OAuth2 logins are also stored as entries in this table, using the user's email address from google, a username generated for them by the CustomLoginSuccessHandler and a blank field for the password.
2. Product: Contains all information about the items available in the vending machine, including ID (code), name, category, price, quantity and image file name.
3. Transaction: Stores transaction data such as total cost, total amount paid, change given, date/time and user ID.
4. TransactionProduct: A linking table used to support many-to-many relationships between transactions and products. Each record maps a product to a transaction, including the quantity of the product bought within the transaction.

These entities are used together to ensure that all purchases are properly stored, stock levels are correctly updated, and transaction histories can be accurately retrieved for users or admin purposes.

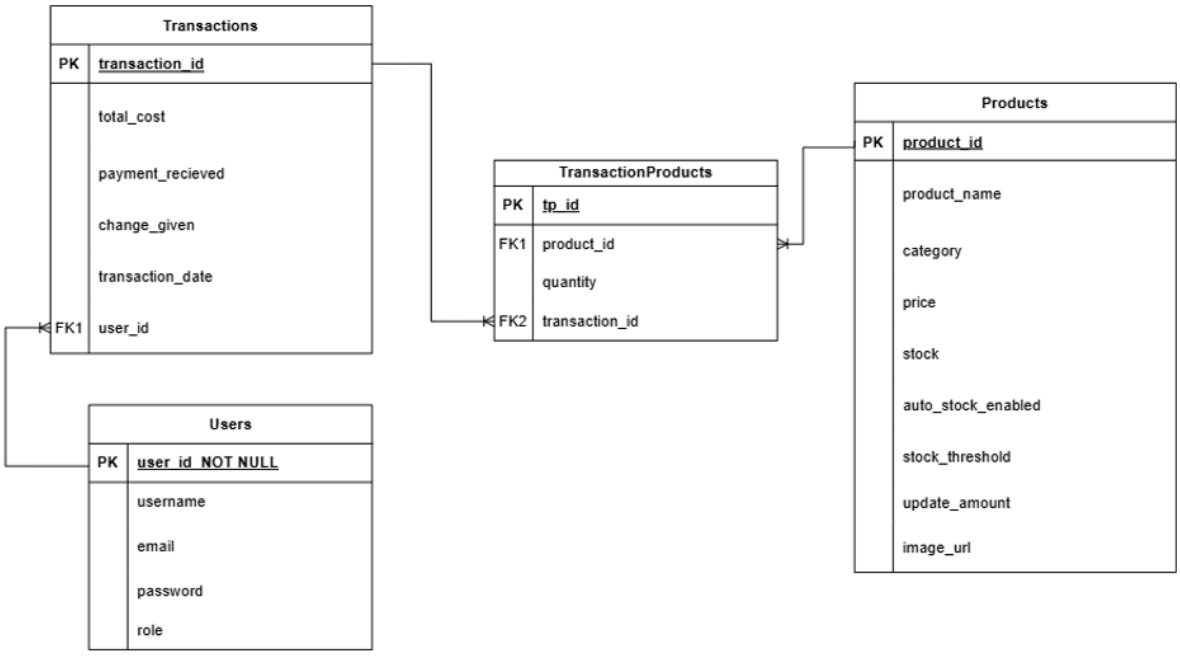
Entity Relationships Overview:

- A User can have many Transaction records.
- A Transaction is linked to multiple TransactionProduct records.
- Each TransactionProduct record connects to a Product.

This structure supports flexible querying and efficient data retrieval, which is especially useful for user transaction history viewing, admin controls and analytics.

Figure A3 – Entity Relationship Diagram (ERD)

Detailed ERD showing entity attributes and relationships, including foreign keys and linking tables. (Created using draw.io).



4.5 User Workflow

The application includes multiple distinct user journeys, depending on the role and purpose of the interaction. Below are the three most important workflows:

1. Standard Purchase Flow (User)

- The user browses available products in the vending machine display of the main page.
- They double click to enhance a product's view and then use a button to add it to their virtual cart.
- Alternatively, they can enter the desired item's code (ID) into the keypad and add it to the cart from there
- As products are added or removed, the total updates dynamically.
- Within the cart, users can modify the quantities of the products, remove the products or clear the entire cart

- When ready, the user proceeds to the payment screen and simulates inserting coins.
- Once full payment is made, the transaction is confirmed, the user can see their items again, a receipt is generated, and stock is updated in the database.
- The user can view the receipt (on a separate page) with an option to download it, or they can go back to the homepage or use the vending machine again.

2. Transaction History & Analytics viewing (User)

- On their home page, logged-in users can navigate to either the transaction history page or the analytics page.
- On the transaction history page, all previous transactions are listed chronologically with details in a table.
- Users can click into each transaction to view receipts.
- Users can double click column headings to sort the table by that column, and subsequently clicking the heading again toggles the sort between ascending and descending.
- Users can also filter/query their transaction data using the filtering menu.
- On the analytics page, data is processed into visuals: summary cards, a bar chart and a line graph showing patterns and insights (e.g. most purchased item, spending trends).

3. Admin Workflow

- Admins log in using dedicated credentials to access a special admin dashboard.
- They are redirected to the secure dashboard after login with product stock tables.
- From the dashboard, they can:
    - Manually update stock values
    - Toggle auto-stock updates for certain products (and set a threshold and update value).

Each workflow was carefully mapped out, implemented and tested (Section 7) to ensure clarity, responsiveness and a maximised user satisfaction. These flows were used as guidance for interface design and logic structure.

Figure A4 – Standard User Workflow

Flowchart of the standard user journey from product selection to transaction confirmation.

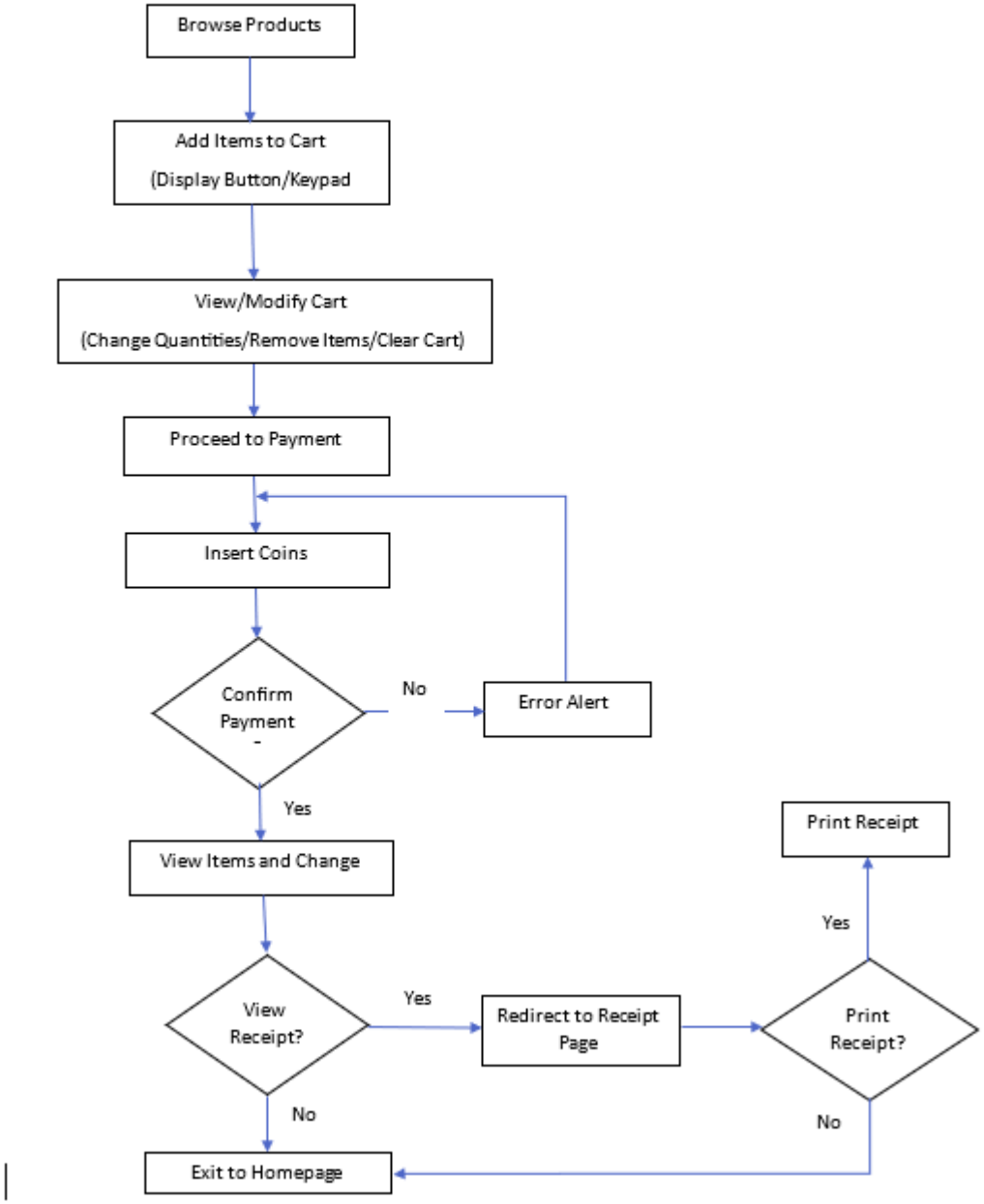
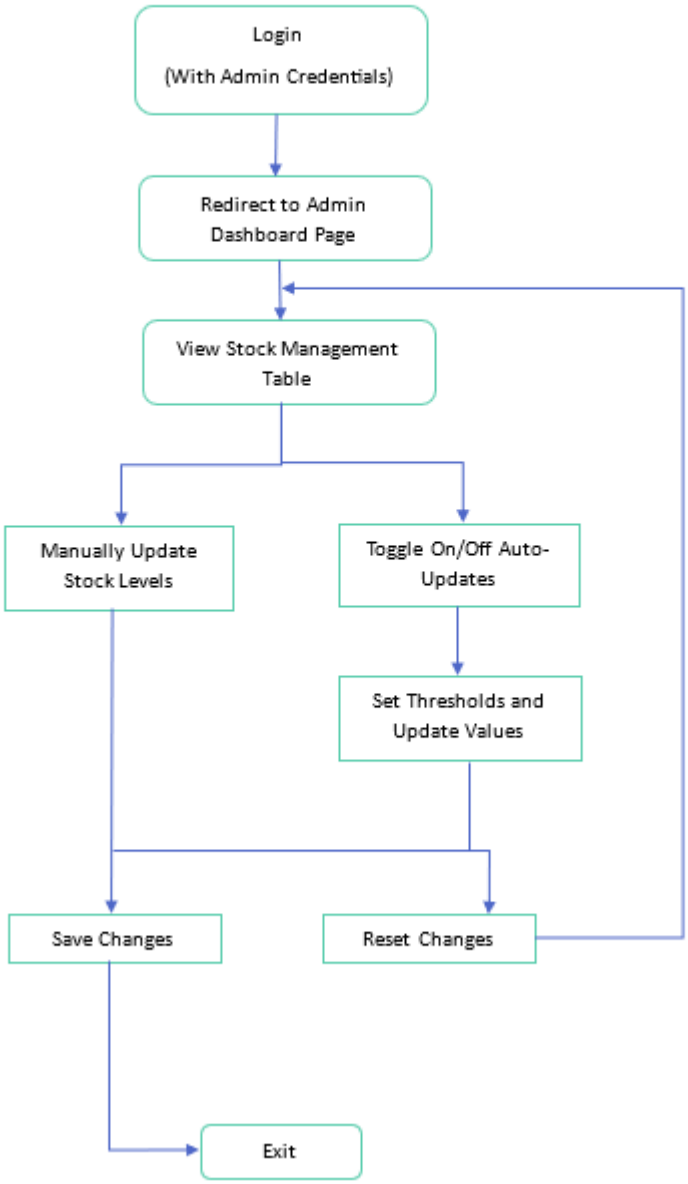


Figure A5 – Admin Dashboard Workflow

Illustrates the admin's workflow for logging in, accessing the dashboard and adjusting stock levels and settings.



5. Implementation

5.1 Development Environment & Tools

The development of the Virtual Vending Machine Web Application was carried out using a combination of industry-standard tools and frameworks that allowed for efficient full-stack development. The backend was implemented using Java (17) with Spring Boot, while the frontend was developed using HTML, Thymeleaf, CSS and JavaScript. The project used Gradle build tool to manage dependencies.

All code was written and maintained in IntelliJ IDEA, chosen due to its strong support of Spring Boot and Java development. For version control, Git was used with commits tracked locally and pushed to my project GitLab repository for documentation and evidence.

The database layer was integrated in using MySQL, which was hosted locally and interacted with the backend through Spring Data JPA repositories. All tables were auto generated through the annotated entity classes using Hibernate's schema generation. This allowed easy updates to the model without the need for external SQL schema changes and would allow the project to be easily setup and run by other users.

Project configuration was managed through application.properties, which defined data source settings, Spring Security configuration and OAuth2 client credentials (loaded securely using environment variables during development).

A local environment was maintained for testing and demonstration purposes. The application was run locally via embedded Tomcat using Spring Boot's developer tools, allowing for quick restarts and live reloads during the development process.

*[The backend and frontend project structure, including key directories such as controllers, services, repositories and resources are shown in **Figure B1 – Project Folder Structure** (Appendix B).]*

5.2 Backend Implementation

The backend was developed using Java with Spring Boot and follows a clean MVC architecture setup. It handles product retrieval, transaction processing, receipt generation and persistence of purchase data. Controllers, services and repository classes were used in modular design to enforce scalability and maintainability.

Product Retrieval

Products are loaded via the 'MainController' at the '/main' endpoint. The method 'showMainPage()' fetches all products from the database using 'ProductService.getAllProducts()' and sends them to the frontend using Thymeleaf model attributes. It also extracts unique product categories for use in the filtering UI. Products are displayed in a dynamic grid on the frontend (see **Section 6.3**).

An additional endpoint: '/api/cart/getProduct/{id}' allows the frontend to fetch product data dynamically via AJAX, and '/api/cart/checkStock/{id}' checks stock levels in real time to validate transactions.

*[A detailed visualisation of the AJAX data communication between the frontend and backend during transaction processing is provided in **Figure B9 – AJAX Data Communication Flow Diagram** (Appendix B).]*

Transaction Processing

The core transaction workflow begins when the frontend sends a POST request to '/api/transactions/create'. This is handled by the 'TransactionController.createTransaction()' method, which receives:

- A map of product IDs and quantities
- The total payment received
- The current user's username (or "Guest")

This data is passed to 'TransactionService.createTransaction()', where:

1. Validation ensures products exist and are in stock.
2. Total cost is calculated using 'calculateTotalCost()', based on product prices and quantities.
3. Payment is validated to ensure it meets or exceeds the total cost.

If validation is successful, a new 'Transaction' object is created and saved using 'TransactionRepo' (repository class). Each purchased product is also saved as a 'TransactionProduct' entry, mapping product IDs to the transaction and storing the quantity. These are saved via 'TransactionProductRepo'.

After the transaction is saved, the controller calls 'TransactionService.deductStock()', which:

- Deducts the quantities from current product stock.
- Triggers auto-restock logic (if enabled and stock is below threshold).
- Returns a list of any auto-restocked items for frontend display.

The controller returns a JSON response including the transaction ID, formatted change value, and restock alert message (if applicable).

Receipt Generation

Users can view receipts for completed transactions via a button navigating to '/receipts/{transactionId}'. This is handled by the 'ReceiptController', which:

- Fetches the transaction using 'TransactionService.getTransactionById()'.
- Maps product IDs to names and prices using 'ProductRepo'.
- Prepares the transaction details for display in a Thymeleaf-rendered 'receiptpage.html' view.

The receipt includes all key purchase details: product names, quantities, individual prices, subtotals, grand total, payment received, change and date/time.

For PDF downloads, the frontend captures the rendered receipt and processes it via html2canvas and jsPDF (see **Section 6.3**).

Error Handling and Security

All backend logic is protected by:

- Server-side validation to prevent underpayment or over-ordering
- Input checking (e.g. null/empty username values)
- Spring Security access control (see **Section 6.4**)

Guest transactions are accepted, but their records are stored with the username set as "Guest". Logged-in users have their usernames saved with each transaction, enabling access to analytics and receipts.

Repository Integration

All data persistence is handled via Spring Data JPA repositories:

- ProductRepo handles product details.
- TransactionRepo stores transaction records.
- TransactionProductRepo for recording individual product and quantity entries per transaction.
- UserRepo manages user accounts.

These repositories support CRUD operations and custom query methods, ensuring a strong backend integration with the MySQL database.

5.3 Frontend Implementation

The frontend of the Virtual Vending Machine Web Application was implemented using a combination of HTML (with Thymeleaf integration), CSS and JavaScript. Its purpose is to provide a responsive and highly interactive user interface that closely mimics a physical vending machine. Strong focus was placed on user experience through animated interactions, real-time updates and easy navigation.

Main Vending Page ('mainpage.html')

The main vending interface is served at the '/main' endpoint, rendered through 'mainpage.html'. On page load, the 'MainController' populates a model with all available products and passes it to Thymeleaf, which dynamically renders a vending machine type grid layout of products.

Each product box shows:

- The product name, price, and image.
- Availability indication (out of stock products are blurred out and have a giant red 'X' over them).
- An enhanced view modal triggered on a double-click.
- An item code (same as the Product ID) for selecting items using the keypad.

*[A visual of the main vending machine product grid, cart sidebar and relevant instructions panel is provided in **Figure B4 – Main Vending Machine Page**(Appendix B).]*

This dynamic structure is supported by JavaScript functions within 'mainpage_script.js' including:

- 'addToCart(id)' - adds an item to the cart.
- 'updateCartTotal()' - recalculates total cost of the whole cart.
- 'removeItem(id)' - removes an item from the cart.
- UI rendering logic for product animations, stock indicators and alerts.

The cart is shown in a sidebar-style container, dynamically updating with item details and subtotal values. Within the cart users can control individual items: increasing/decreasing the quantities or removing the item from cart. Once the user is satisfied, they can click "Checkout", which replaces the cart with a custom "payment modal" containing an interface for the payment panel, using JavaScript-controlled visibility toggles.

*[The JavaScript logic responsible for adding items to the cart and dynamically updating the cart interface is detailed in **Figure B2 – JavaScript Cart Functions** (Appendix B).]*

*[The complete user journey through the UI, from product selection to payment completion is visualised in **Figure B8 – Frontend User Interface Flow Diagram** (Appendix B).]*

Payment Logic ('payment_script.js')

Inside the payment interface, users can insert coins virtually by clicking buttons for coin values (e.g. 10p, 50p, £1). These buttons trigger animated coin elements that move into a visual coin slot area of the machine, using CSS transitions and class changes.

JavaScript functions manage:

- Coin animation ('insertCoin()'),
- Payment total calculation ('updateInsertedAmount()')
- Validation to enable the "Confirm Payment" button only when enough payment is made.

The remaining balance is dynamically updated after each coin insertion. Once the total has been met or exceeded, the "Confirm Payment" button sends the purchase data (via AJAX POST to '/api/transactions/create') and receives confirmation details, including the change value and new transaction ID.

Upon success, a modal ('successModal') appears with a summary of the purchase:

- Total paid, change given and a display of all items purchased (images, IDs, names, quantities)
- Action buttons for receipt viewing, returning to the home page or closing the modal to immediately start a new purchase.

*[The transaction success modal, summarising purchase details and providing receipt options, is illustrated in **Figure B6 – Successful Transaction Modal** (Appendix B).]*

The modal and cart areas are styled and structured to reflect a real-world vending experience using styles from ‘mainpage_styles.css’.

*[The JavaScript functions handling coin animation, payment tracking, and dynamic updates during checkout are shown in **Figure B3 – JavaScript Payment Functions** (Appendix B).]*

*[The payment panel interface, including coin insertion animations and real-time balance tracking, is displayed in **Figure B5 – Payment Interface Screenshot** (Appendix B)]*

Receipt Page (‘receiptpage.html’)

When a user chooses to view their receipt (‘/receipts/{transactionId}’), the backend fetches transaction data and product names/prices, which are rendered into a detailed receipt layout in ‘receiptpage.html’.

The page displays:

- User info (or “Guest” if not logged in)
- Date and time of transaction
- Items breakdown (product name, quantity, unit price, subtotal)
- Total cost, amount paid, and change given.

The design uses fixed positioning and a monospace style (Courier New) to emulate a printed real life receipt appearance. This is further enhanced with a glowing header and structured box using ‘receiptpage_styles.css’.

Interactive features include:

- “Download Receipt” button - triggers ‘downloadReceipt()’ function in ‘payment_script.js’, which uses “html2canvas” and “jsPDF” libraries to effectively capture a box image of the whole receipt and attach it onto a downloadable PDF file.
- “Return to Home” - redirects users to the home page (‘/home’).

Error handling is built in - if a receipt ID is invalid or unauthorised, an error message is shown instead of the receipt box, to prevent the page from crashing.

*[A completed receipt page showing full purchase details, styling, and user options is shown in **Figure B7 – Receipt Page After Transaction** (Appendix B).]*

Homepage and Navigation (‘homepage.html’)

The homepage (‘/home’) acts as the user dashboard. It is dynamically updated based on login state:

- If logged in: shows options to view analytics and transaction history.
- If guest: analytics and transaction history access is blocked and the buttons to access these are disabled.

The layout and theme are consistent with the vending machine aesthetic and are styled via ‘homepage_styles.css’. Backgrounds, button animations and gradient colours are used to keep the interface engaging by reflecting a futuristic aspect.

Styling and Theme

All pages use a shared base style in ‘general_styles.css’ for basic common elements such as buttons, headings, labels and hover effects. Each core page also includes a page-specific stylesheet (e.g. ‘mainpage_styles.css’, ‘receiptpage_styles.css’) to customise layout and visuals.

Throughout the frontend, responsiveness and accessibility were considered. Button hover effects, modal transitions, and colour contrasts contribute to an experience that is both functional and appealing. Overall, the aesthetic of the UI was an important factor for me, so I spent lots of time and effort trying to style it as detailed as possible and use a unique and consistent theme throughout of neon futuristic, to try to maximise user engagement.

5.4 User Authentication and Security

The authentication and security of the system were implemented using Spring Security, which handles user registration, login, role-based access, session management and OAuth2 login via Google. The system supports three user types: guests, standard users, and admins.

Registration and Login

The registration process is managed by 'RegisterController', which handles GET and POST requests to an endpoint: '/register'. The registration form collects a username, email, password, confirm password and a required consent checkbox. Submitted data is validated to ensure passwords match and to check that the username and email are not already in use. On success:

- The password is hashed using BCrypt encoder.
- A new 'User' entity is created with the role attribute set to "USER" and stored in the database.

Users are then redirected to the login page ('/login') with a confirmation message.

The login process is handled by Spring Security, which uses a custom login page ('login.html'). Users can log in using either their username or email, both of which are supported in 'UserDetailsServiceImpl'. Upon a successful login, the custom handler class: 'CustomLoginSuccessHandler' routes users based on their role:

- Admins are redirected to '/admin/dashboard'.
- Standard users are sent to the homepage.

OAuth2 Login (Google)

An additional login option using Google OAuth2 was integrated to improve accessibility. Google credentials are configured in 'application.properties', and the login flow is managed by Spring Security's '.oauth2Login()' configuration. The login page includes a Google login button which redirects users to Google's consent screen.

On a user's first OAuth2 login:

- The app retrieves the user's email and name.
- It checks if a user with that email already exists.
    - If it does, it logs them in.
    - If not, it creates a new user in the database with a generated username based on the user's name from their Google account, and a blank password. If the username already exists it adds a number suffix on to the end of the name, e.g. "Username1".

Subsequent logins reuse the same account.

Security Configuration

All relevant files are found in the 'security' directory.

Access control is managed in the 'SecurityConfig' class. Key settings include:

- Open access to '/login', '/register', and static resources.
- Restricted access to '/admin/**' endpoints, only for users with role set to "ADMIN".
- Protected user-only pages like '/transactions' and '/analytics' for role = "USER".

There is also a "Remember Me" login feature, which uses a cookie with a 1-day validity.

The 'UserDetailsServiceImpl' and 'CustomLoginSuccessHandler' classes work together to support both traditional logins and OAuth2 tokens, ensuring consistent user session handling.

Session and Guest Handling

Guests can access the main vending machine page and complete purchases as a guest, without having to log in. In these cases, the transaction is recorded without an associated user. Logged-in users have their user ID stored in the Transaction record, allowing access to receipts, transaction history and analytics later on.

Role-based access prevents users from accessing pages outside of their scope. Any unauthorised attempts to access admin routes are blocked by Spring Security, which redirects to the login or homepage.

Security Practices

Key security measures implemented:

- All passwords are stored hashed using BCrypt.
- Input data validation during registration and login.
- Role-based access control on all routes.
- Secure handling of Google Oauth2 tokens and prevention of duplicate accounts.
- Server-side validation for all transactions to prevent bypassing payment logic.

This setup ensures a robust security implementation, while still offering flexibility and accessibility to users.

5.5 Admin Features and Stock Management

Administrative functionality was implemented to provide system maintainers (admins) with full control over the vending machine's stock and auto-restock configuration. These controls were designed to be simple, responsive and secure, and are only accessible to users with the "ADMIN" role.

Admin Dashboard

Upon logging in as an admin, using the admin credentials, users are redirected to the admin dashboard ('/admin/dashboard'), handled by the 'AdminController' class. This controller retrieves all available products via a method in 'ProductService' and sends them to the admin dashboard view ('admin.html') using AJAX, which displays an up-to-date structured table of product information.

The table includes:

- Product names, categories and IDs.
- Product stock (pre-filled with current stock).
- Buttons to increment, decrement, or empty the stock field.
- An input field to manually set the stock value.
- Auto-restock toggle (checkbox).
- Threshold and update amount fields (shown when auto-restock is enabled).
- Undo buttons per row and global Save and Reset buttons.

These interactive elements are implemented using JavaScript to enable real-time feedback. When a stock value is changed, the new provisional stock value is displayed next to the current stock (visually highlighted). The undo button resets that row's inputs to their original values. A "Save All" button on the side sends changes in bulk to the backend to be via an AJAX POST request, and the changes are then permanently saved by the backend and reflected in the table in real time.

Backend Update Handling

The 'AdminController.updateStock()' method receives a list of modified products from the frontend. For each product:

- It fetches the existing entity from the database.
- Applies stock changes (ensuring stock is not negative).
- Updates auto-restock settings if toggled on or off.
- Saves the updated product using ProductRepo.

After a successful update, the frontend displays a toast message confirming the save and removes the modified highlights and the change are now applied and displayed appropriately (e.g. provisional stock becomes the current stock after being saved).

Auto-Restock Behaviour

Auto-restock allows products to be replenished automatically after a transaction, ensuring key products never run out unexpectedly. Each product entity includes:

- A Boolean flag ‘autoStockEnabled’.
- An integer ‘stockThreshold’ to know at which point to restock.
- An integer ‘updateAmount’ to define how much to restock by.

After a purchase, the ‘ProductService’ checks if a purchased product has auto-restock enabled and whether its current stock is below the defined threshold. If so, it increases the stock automatically by the specified update amount.

This logic is embedded directly into the transaction flow, meaning auto-restock is evaluated immediately after a product is purchased. This removes the need for any scheduled background tasks and ensures product availability is preserved in real time.

Security and Access Control

Only users with the “ADMIN” role can access the admin dashboard or perform stock updates. This is enforced by both:

- Route-level restrictions in SecurityConfig (e.g. ‘.requestMatchers("/admin/**").hasRole("ADMIN")’)’
- Annotation for the controller class: ‘@PreAuthorize("hasRole('ADMIN')")’

Attempts to access these endpoints as a standard user or guest are denied by Spring Security and will appropriately re-route.

Additional Notes

Although the admin system currently focuses on product and stock management, the design is extensible. With minimal adjustments, future functionality could include:

- Viewing user accounts or analytics across all users.
- Creating new products with images and descriptions.
- Managing transaction logs and export tools.

This is further discussed in **Section 8: Evaluation.**

*Full testing of additional features such as transaction history, analytics, card payments and the smart recommendation algorithm are included in **Appendix C – Additional Features Implementation**.*

6. Testing

6.1 Testing Strategy

The Virtual Vending Machine Web Application was tested using a combination of manual black-box testing, integration testing and exploratory testing methods to validate functionality, usability, performance and security. Both backend and frontend were tested thoroughly in a full-stack environment, with emphasis placed on real user flows from product selection through to payment completion and receipt generation.

Manual testing was primarily performed during development iterations using live application testing in the browser, API Testing (Postman) and direct database inspection through MySQL Workbench. Frontend components were validated through interactive testing using Chrome DevTools.

The primary focus was integration testing: ensuring that data flows correctly between the frontend, backend and database layers. Although formal unit testing was not the initial priority, unit tests for critical backend service classes have been added in using JUnit to further validate business logic (see **Section 7.4 Unit Testing**).

This overall combined approach ensured that the final system met the functional, security and usability standards necessary for a responsive, secure and user-friendly web application.

6.2 Backend Testing

The backend of the Virtual Vending Machine Web Application was extensively tested to ensure that all server-side logic, API endpoints and security mechanisms functioned correctly and reliably. Testing primarily focused on the RESTful API endpoints and database interactions, using manual black-box testing with Postman, direct database inspection through MySQL Workbench and Spring Security access control validation.

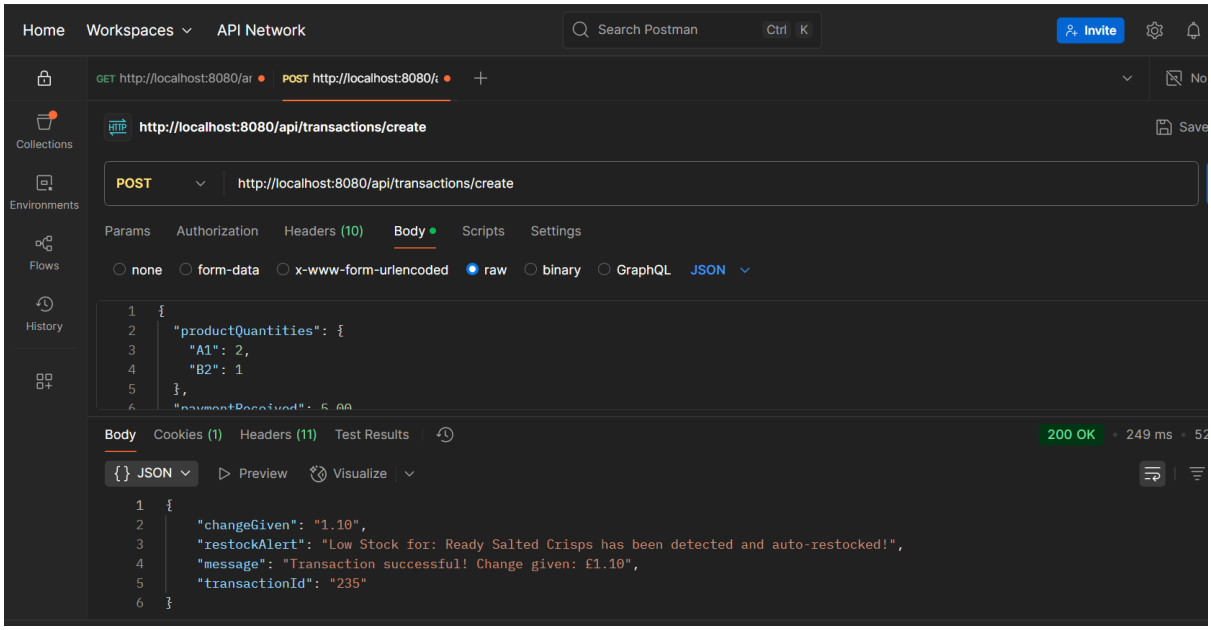
API Endpoint Testing

All public and private API endpoints were manually tested using Postman. Each test verified:

- Correct response structure (JSON format with expected keys).
- Appropriate status codes (200 OK, 400 Bad Request, 403 Forbidden).
- Accurate data returned based on input parameters.
- Server-side validation preventing bad data from being processed.

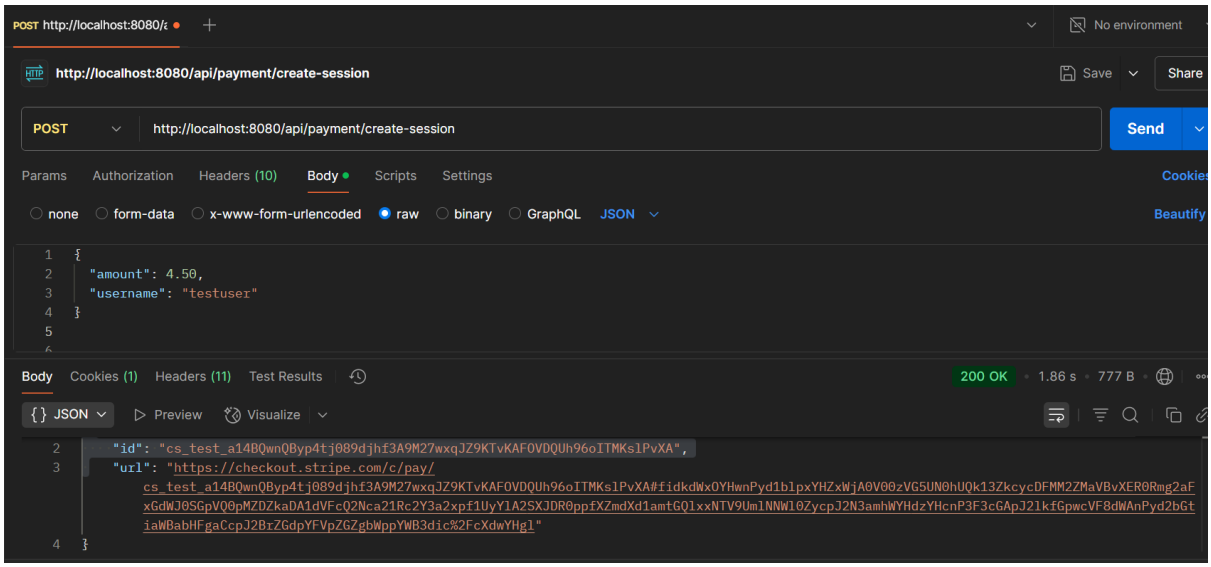
Key examples tested include:

- **Transaction Creation**:
  Sending a valid POST request to '/api/transactions/create' and verifying a successful JSON response with transaction ID and change.

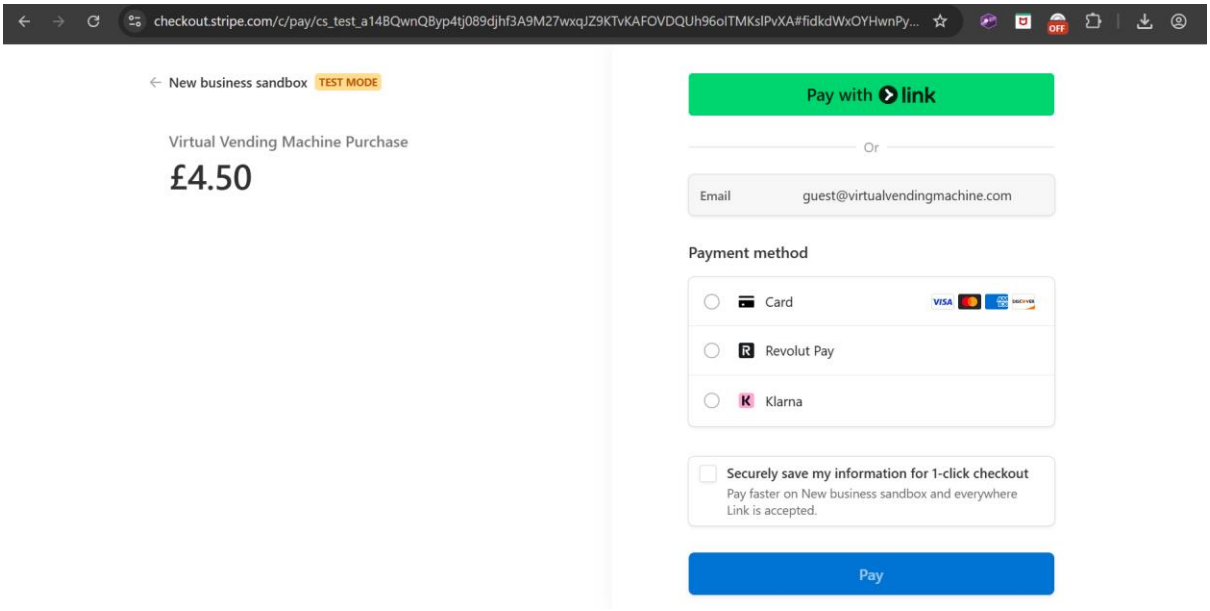


Test Result: Pass. Transaction successfully created with an ID, change value, message and (additional) restock alert.

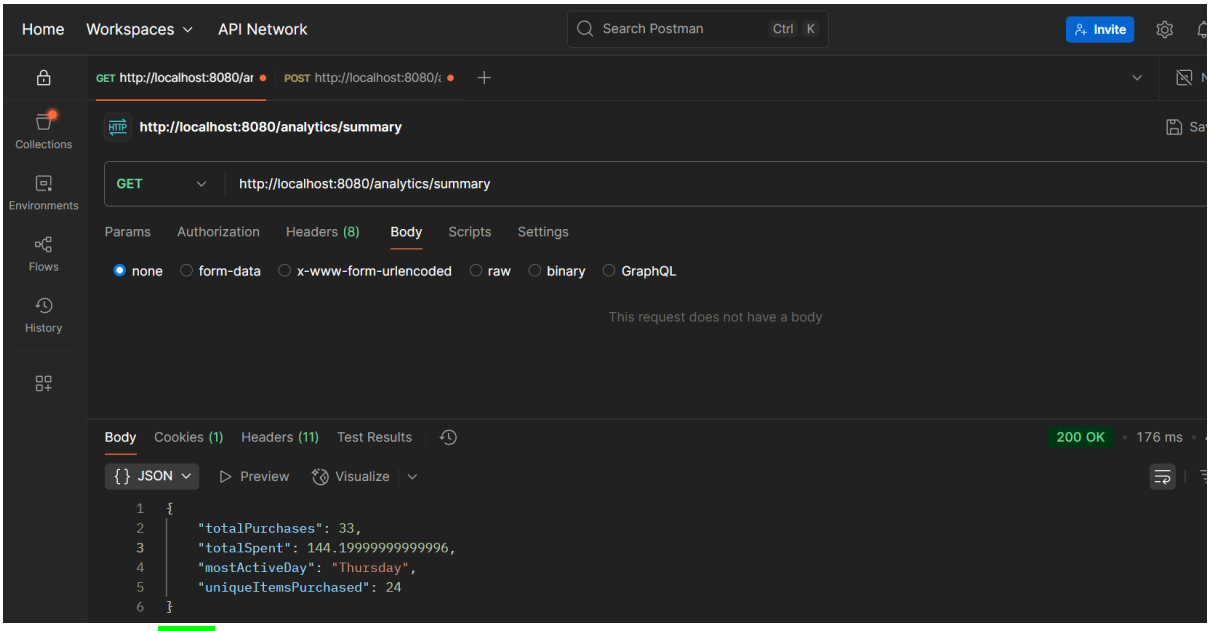
- **Payment Session Creation (Stripe)**:
  Testing a POST request to '/api/payment/create-session' and checking redirect URL generation for Stripe checkout.



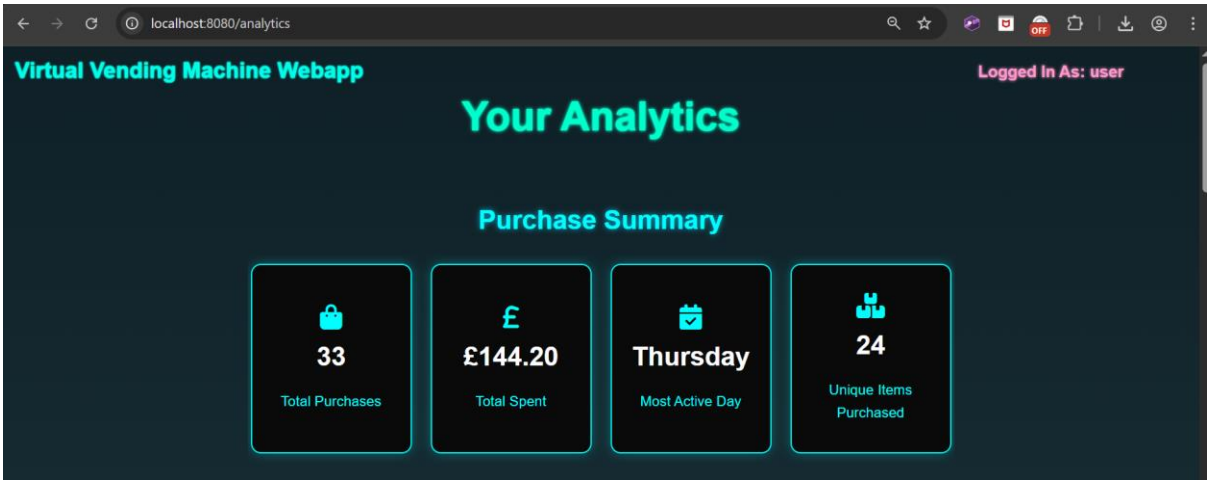
Test Result: Pass. Correct Status Code (200 OK) and correct ID and URL returned as JSON. URL successfully redirects to the Stripe checkout page:



- **Analytics Summary Fetching**:
  Verifying GET request to '/api/analytics/summary' returns correct user analytics.



Test Result: Pass. Correct Status Code: "200 OK". Successfully returns analytics summary data as JSON. Minor issue: 'totalSpent' is sent unrounded, but this is rounded before being displayed in the frontend.

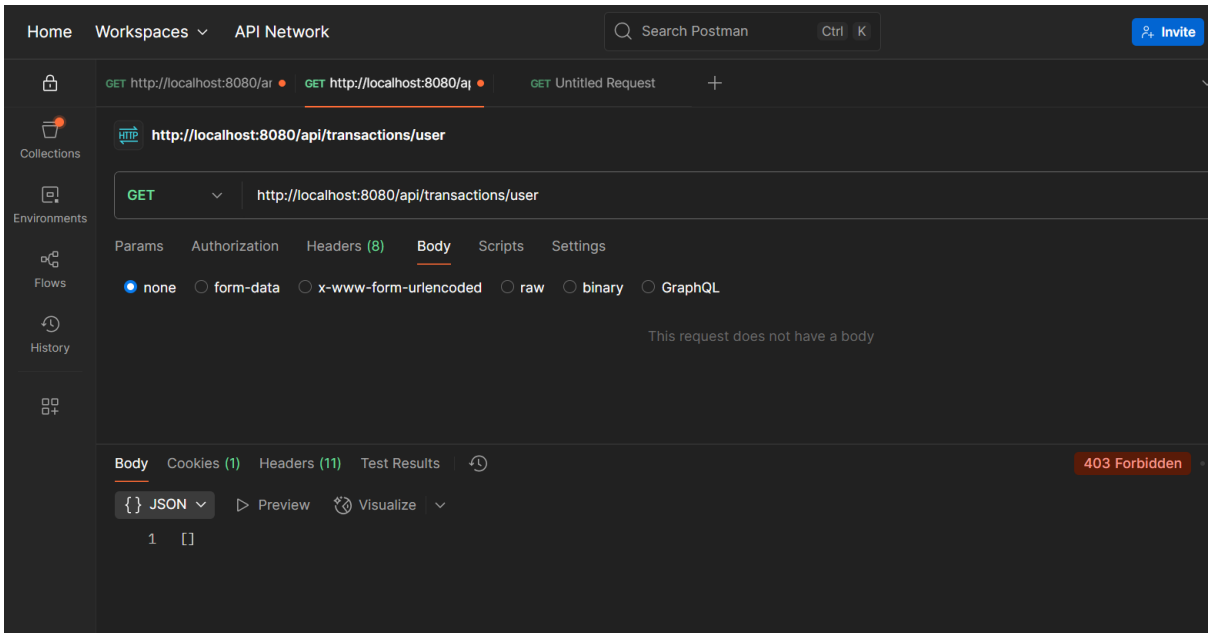


The sent JSON data matches the data found on the analytics page and 'totalSpent' is rounded here.

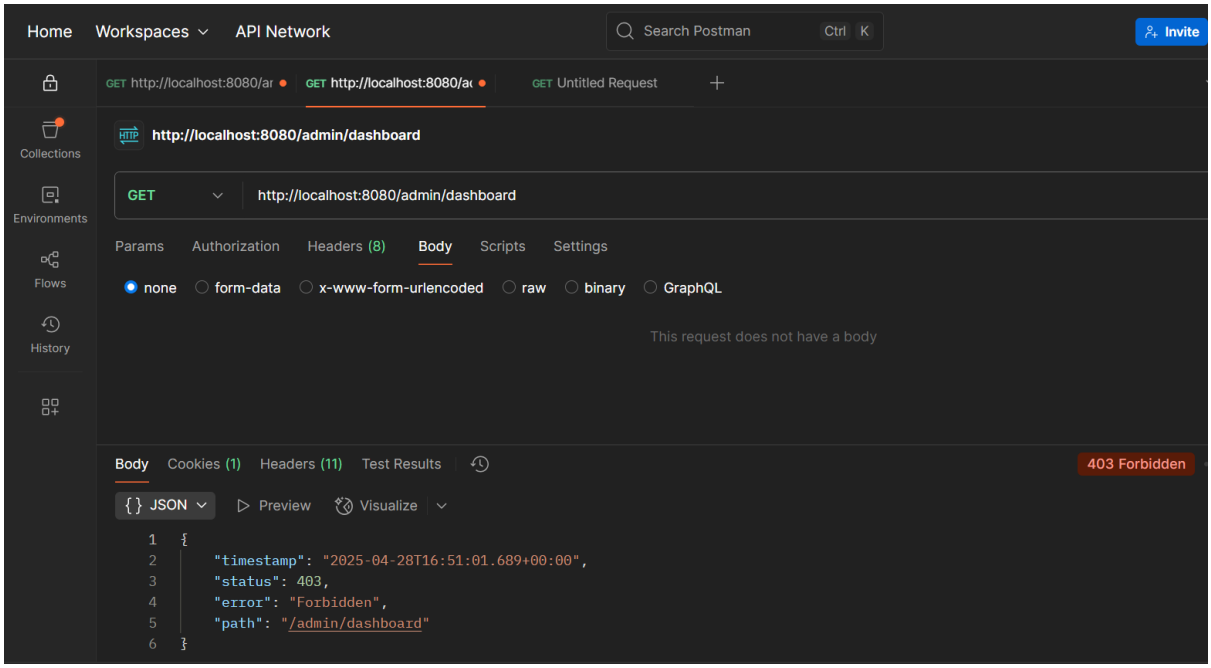
Security Testing

Backend route protection was validated to ensure unauthorised access was correctly blocked:

- Guest access restrictions were tested by manually trying to call user-only endpoints (e.g., '/api/transactions/user') without being logged in.
- Role-based access control was tested by attempting to access admin routes (e.g., '/admin/dashboard') as a standard user.



Test Result: Pass. Transactions history page is forbidden for unauthenticated (Guest) users.



Test Result: Pass. Admin dashboard is forbidden for standard users ('ROLE = "USER"').

Spring Security was successfully confirmed to:

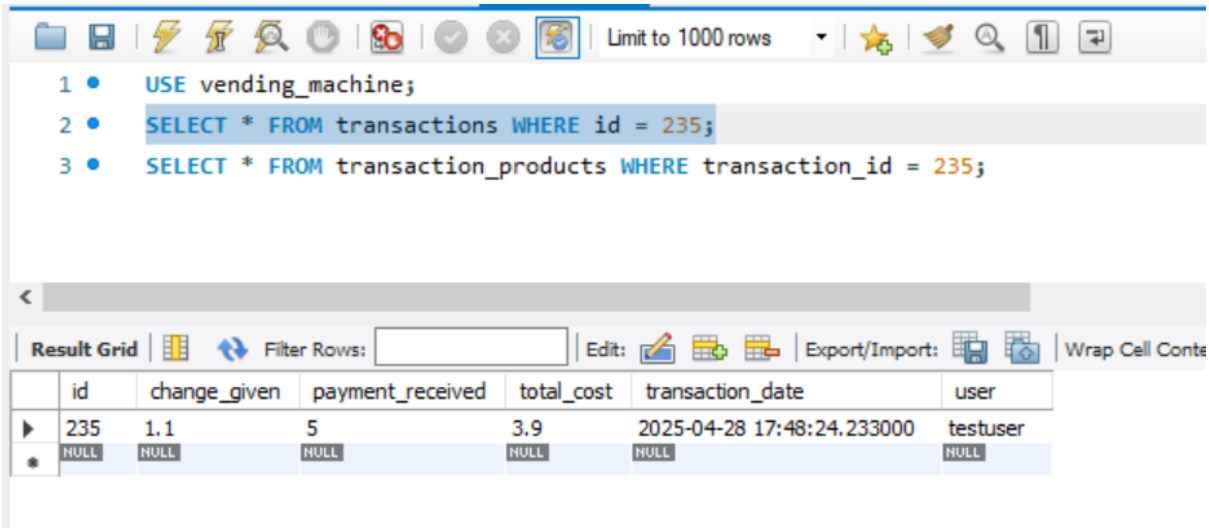
- Redirect unauthenticated users.
- Block unauthorised roles (e.g. users accessing admin page).
- Correctly handle OAuth2 login and session persistence.

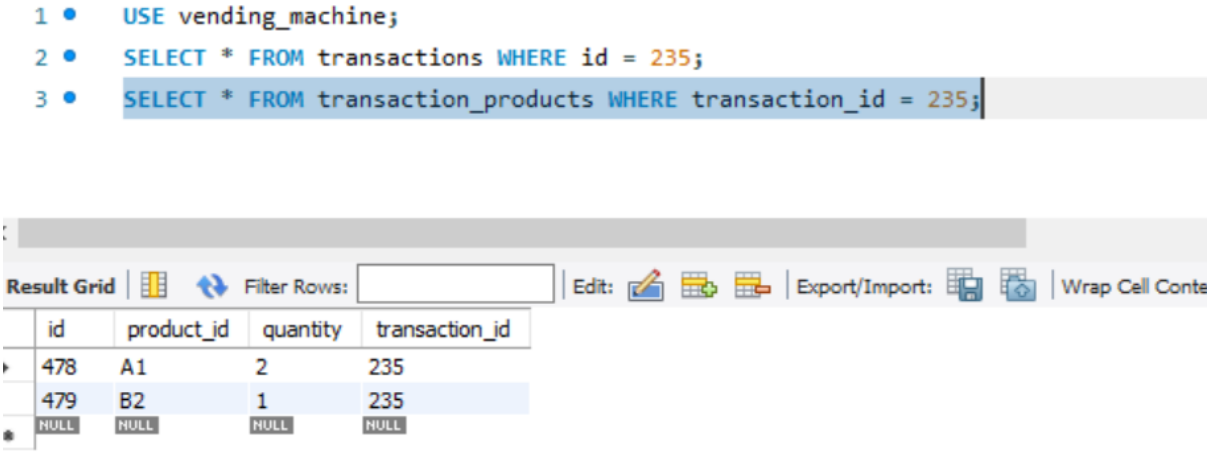
Database Validation

Database records were inspected using MySQL Workbench to validate:

- Transactions were correctly saved into the 'transaction' and 'transaction_product' tables.
- Stock levels updated appropriately after purchase.
- Users registered via standard and OAuth2 login were properly inserted into the user table.

Using the transaction from the Transaction Creation Test earlier (ID=235):





Test Result: Pass. The new transaction is correctly stored in 'transactions' table and the product entries are correctly stored in 'transaction_products'.

Summary of Backend Testing

Backend testing confirmed that:

- API endpoints operate according to specification.
- Input validation prevents bad requests.
- Role-based access controls are enforced securely.
- Data persistence behaves consistently across transactions, user registration and analytics fetching.

All critical backend functionalities have been verified through black-box testing methods, providing strong assurance of server-side stability.

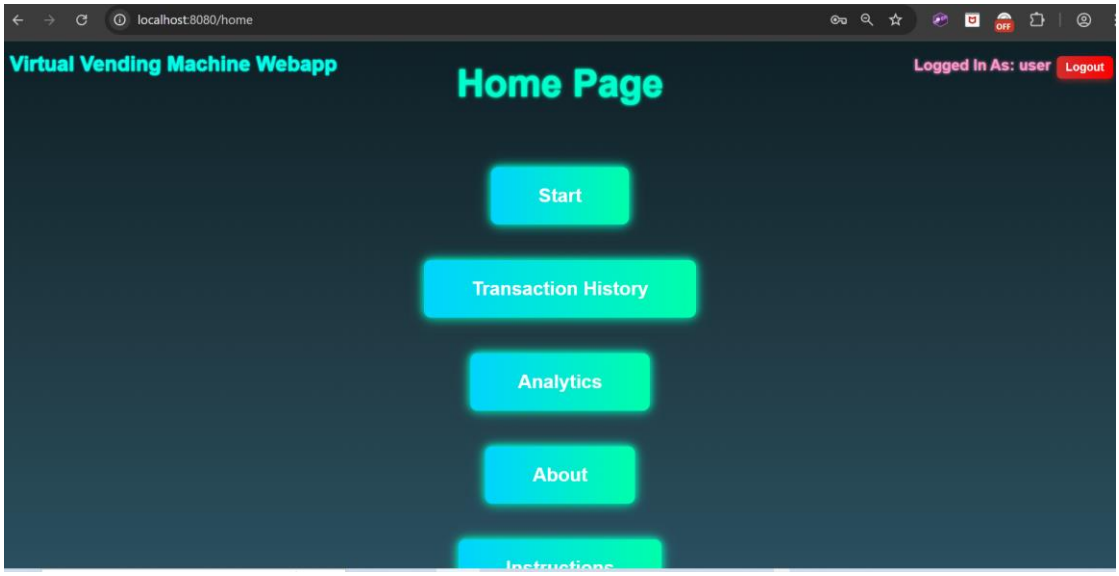
6.3 Frontend Testing

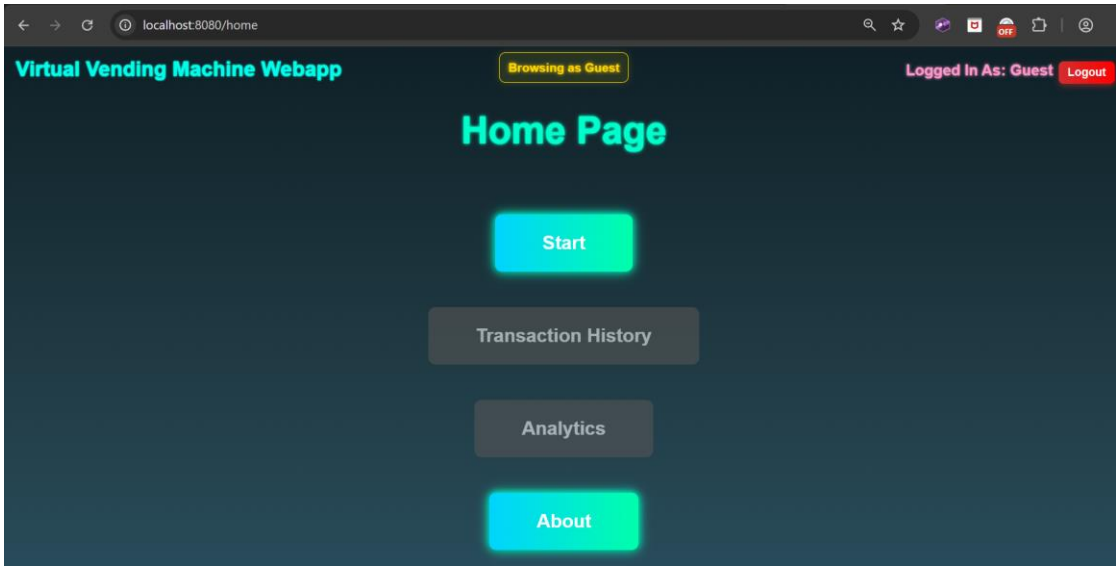
Frontend testing was focused on verifying that all web pages, UI elements and dynamic JavaScript functions performed as intended, under various user scenarios. Testing was carried out manually using Chrome Developer Tools and standard browser-based interaction tests.

Homepage Testing

The homepage (‘/home’) was tested for correct rendering and login state detection:

- As a guest user, the "Analytics" and "Transactions History" buttons were disabled and styled differently to show inaccessibility.
- As a logged-in user, the buttons were enabled and functional, leading correctly to their respective pages.

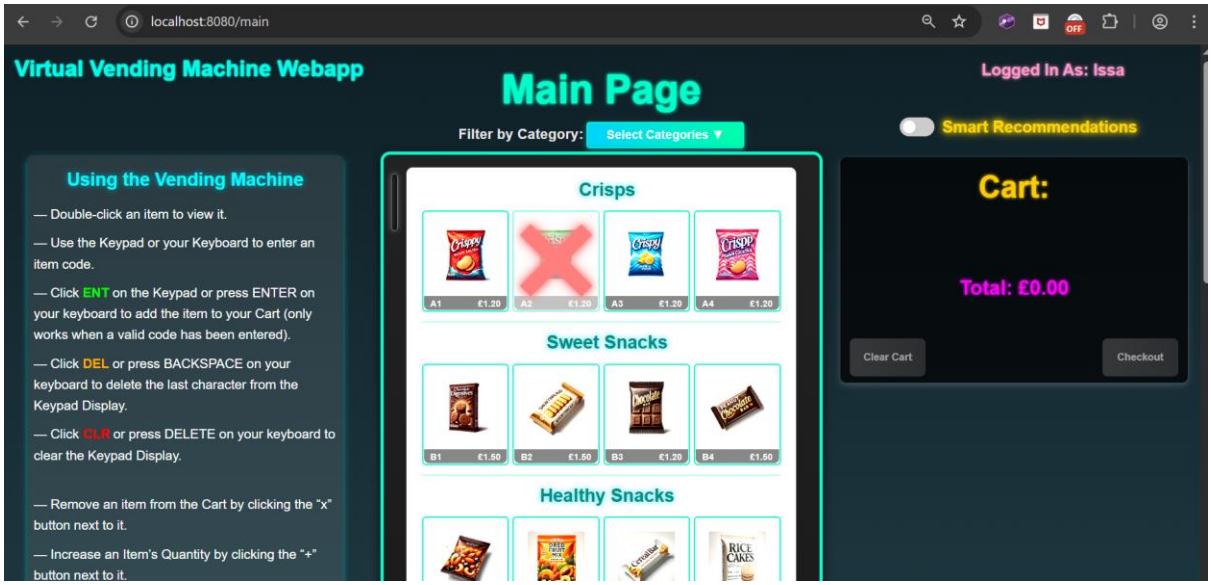


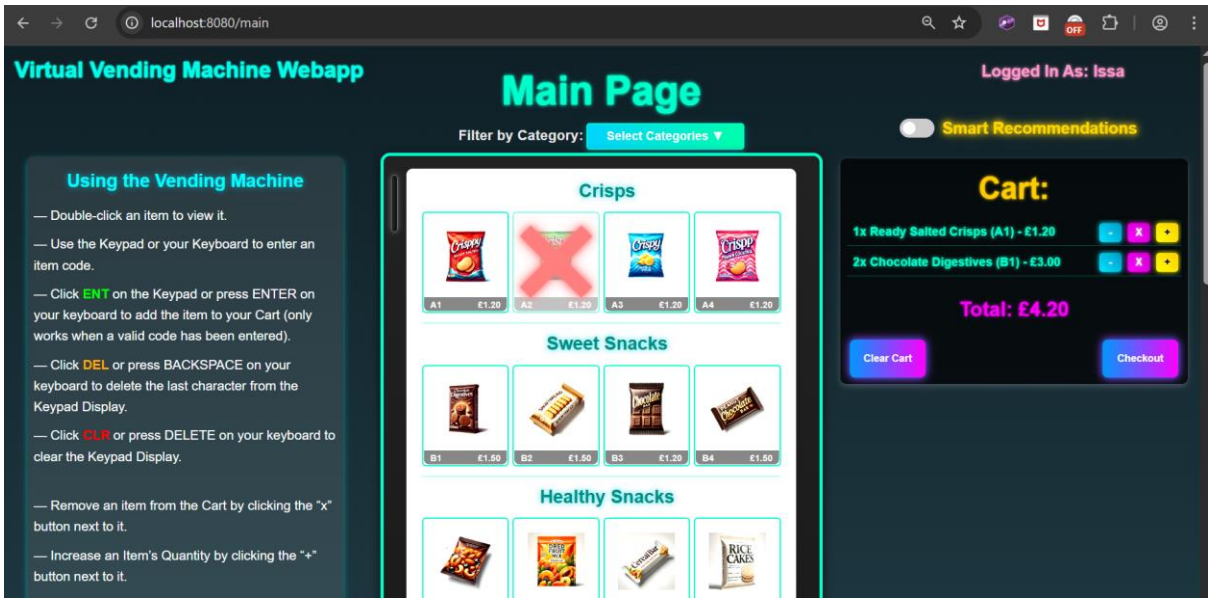


Main Vending Machine Page Testing

The main vending machine page (‘/main’) was tested for correct product grid rendering and cart functionality:

- The full grid of products loaded correctly based on backend data.
- Products showed an "Out of Stock" indicator when applicable.
- Adding products updated the cart dynamically without page reload.
- Removing products or adjusting quantities in the cart updated the total cost in real time.
- The "Checkout" button only activated when at least one item was in the cart.

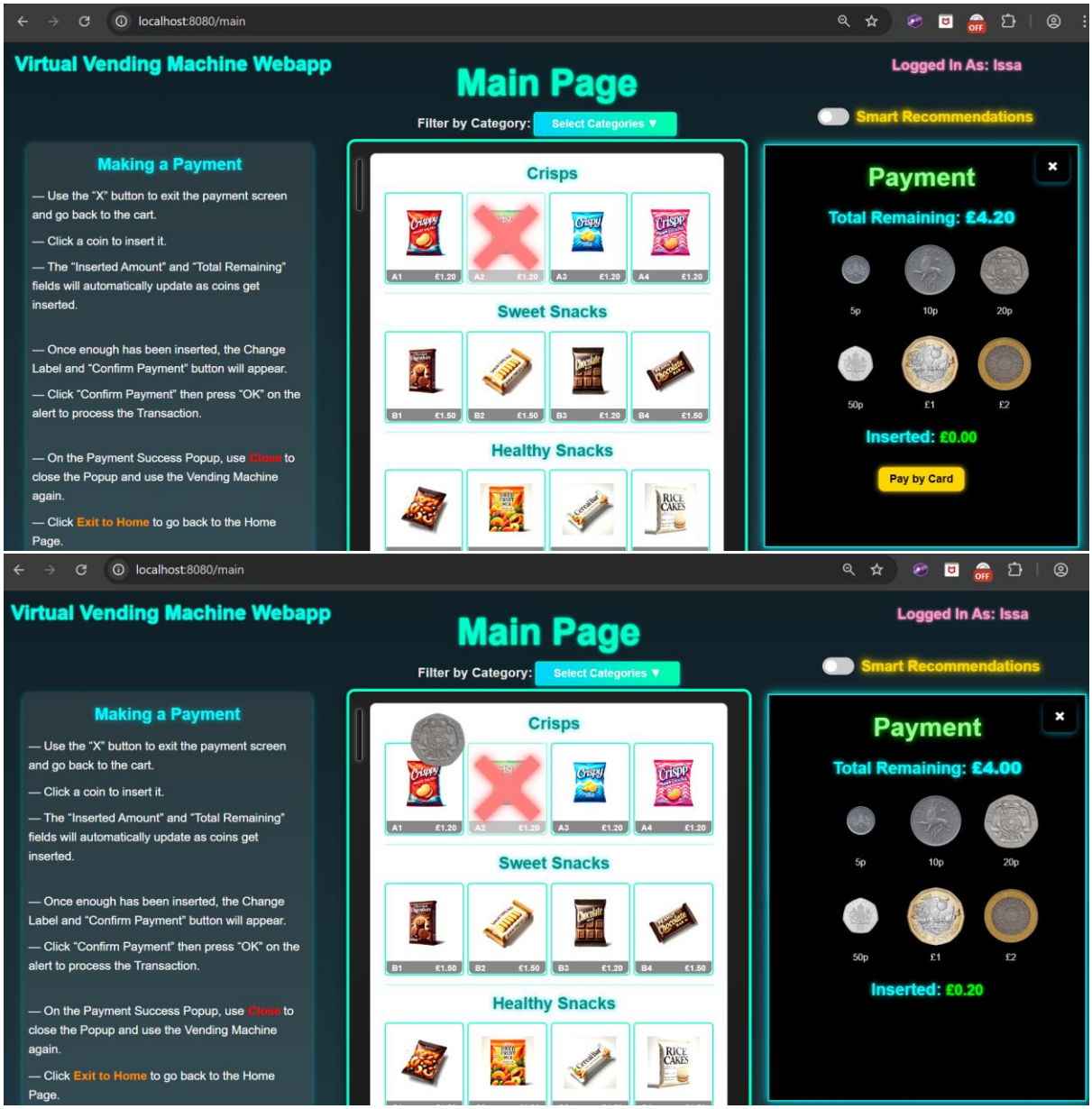




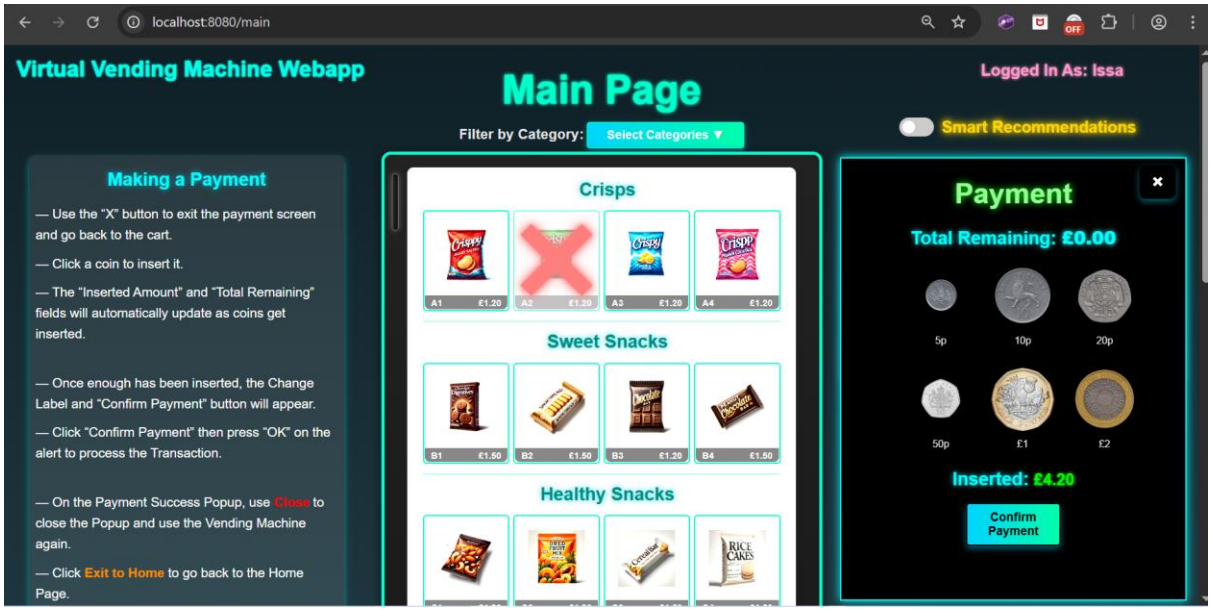
Payment Flow Testing

The payment interface was tested across both coin insertion and card payment options:

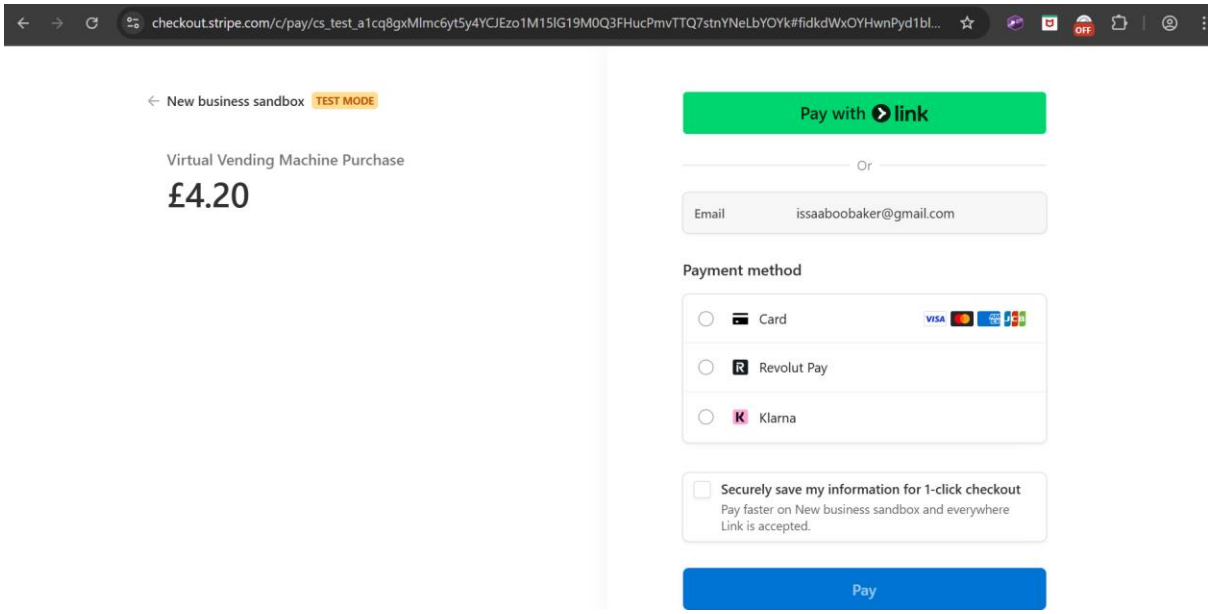
- Coin insertion correctly animated the coin into the vending machine and updated "Inserted" and "Remaining" payment amounts dynamically.
- The "Confirm Payment" button only appeared once enough funds were inserted.
- Card payment option triggered correct redirection to Stripe's Checkout page via the "Pay by Card" button, with correct amount and user email pre-filled.



(Clicked a 20p coin. Coin is being animated towards the slot and the inserted and total amounts have been updated.)



(Full payment has been made. Inserted amount is the full amount (£4.20), remaining amount is £0. "Confirm Payment" button has now appeared.)

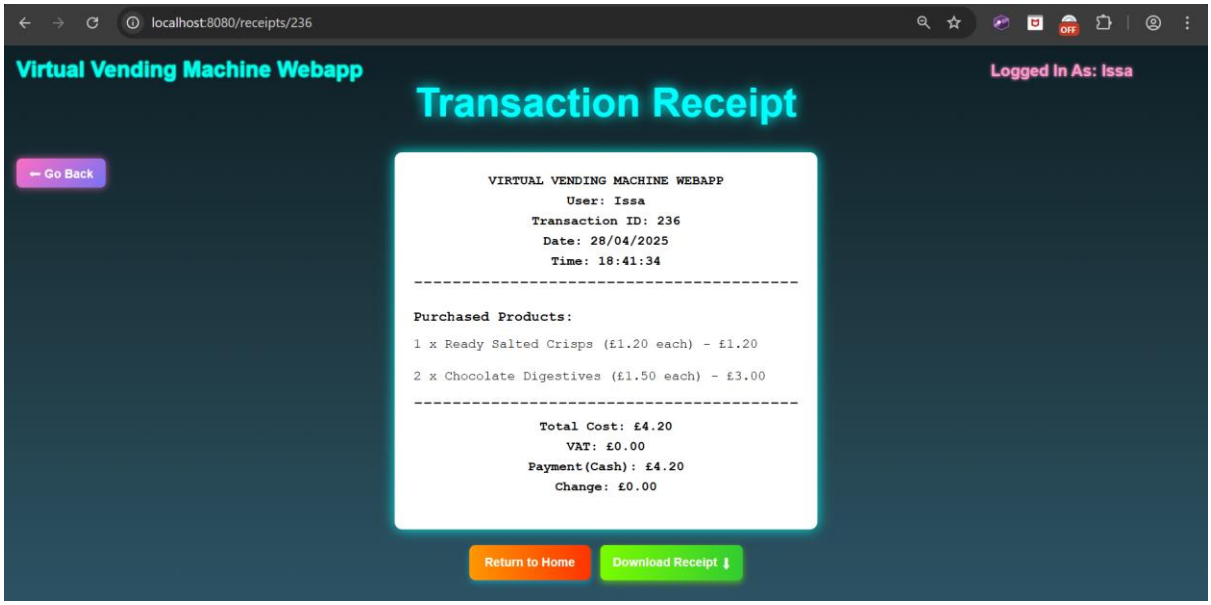


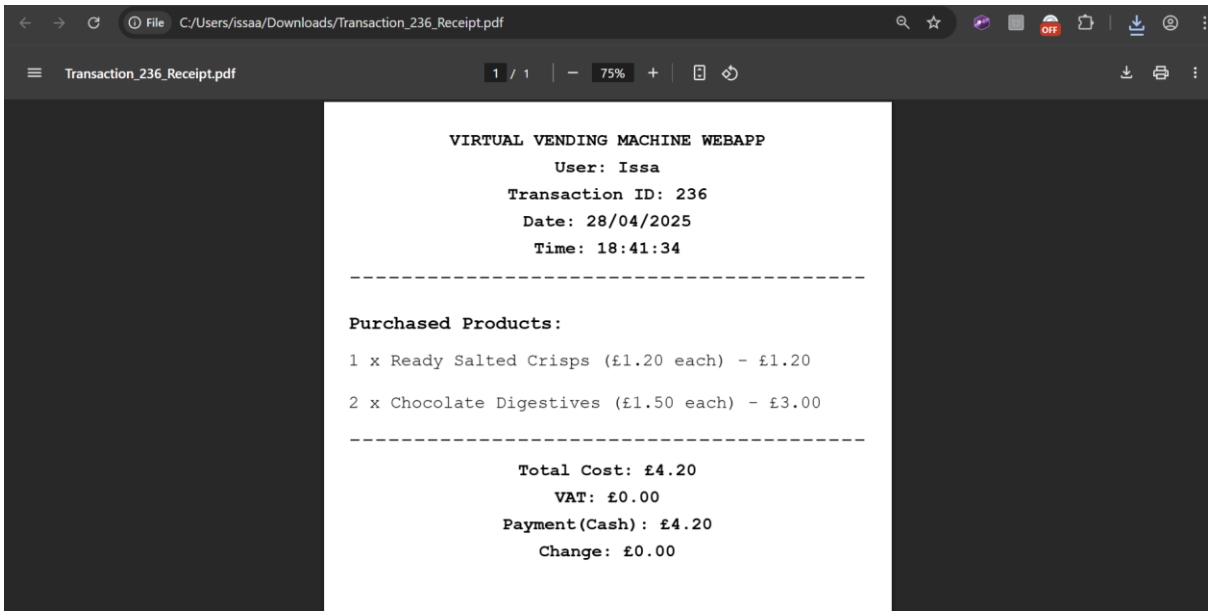
(Pay by Card button was clicked. Page has been redirected to Stripe checkout with the correct user email and amount to pay already filled in.)

Receipt Page Testing

Post-transaction receipt pages (‘/receipts/{transactionId}’) were tested:

- Receipt loaded correctly with full purchase summary, including item names, prices, payment and change.
- "Download Receipt" button generated a PDF file containing an identical receipt layout.



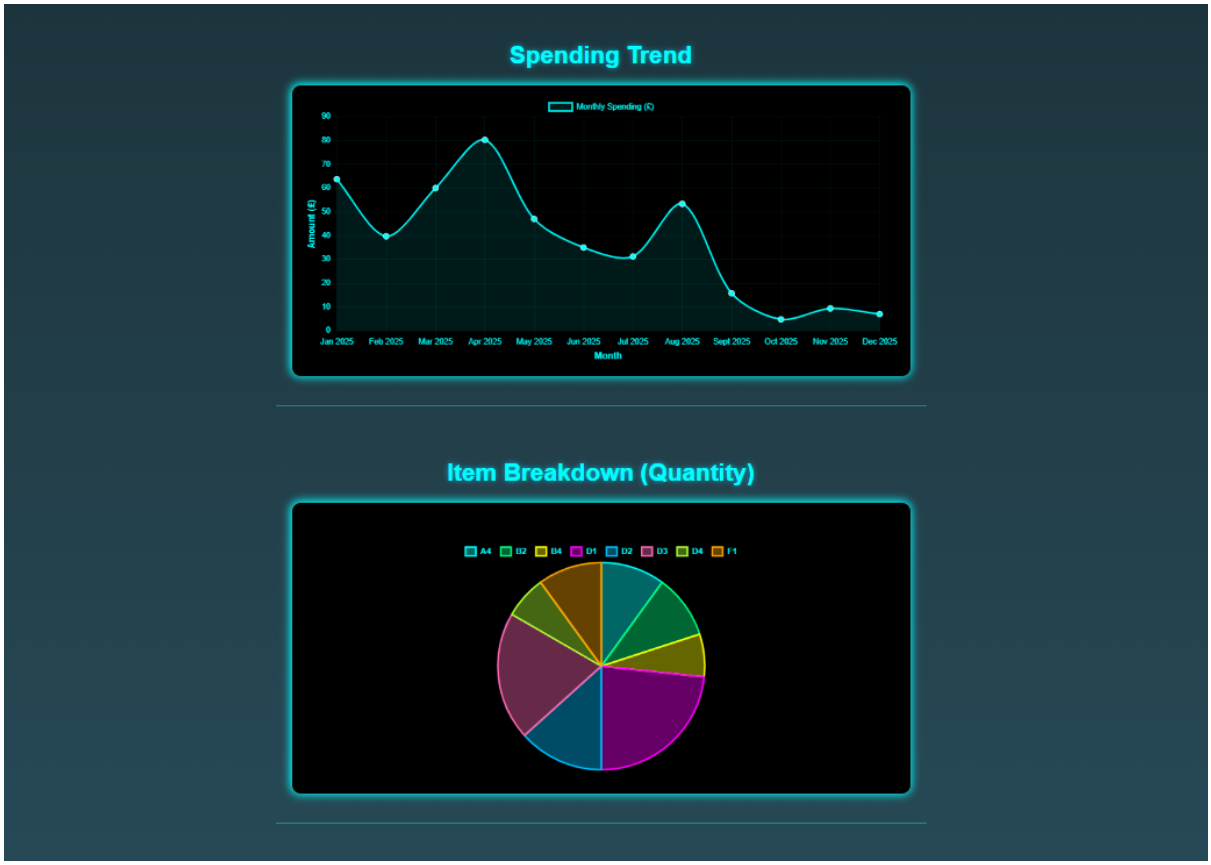


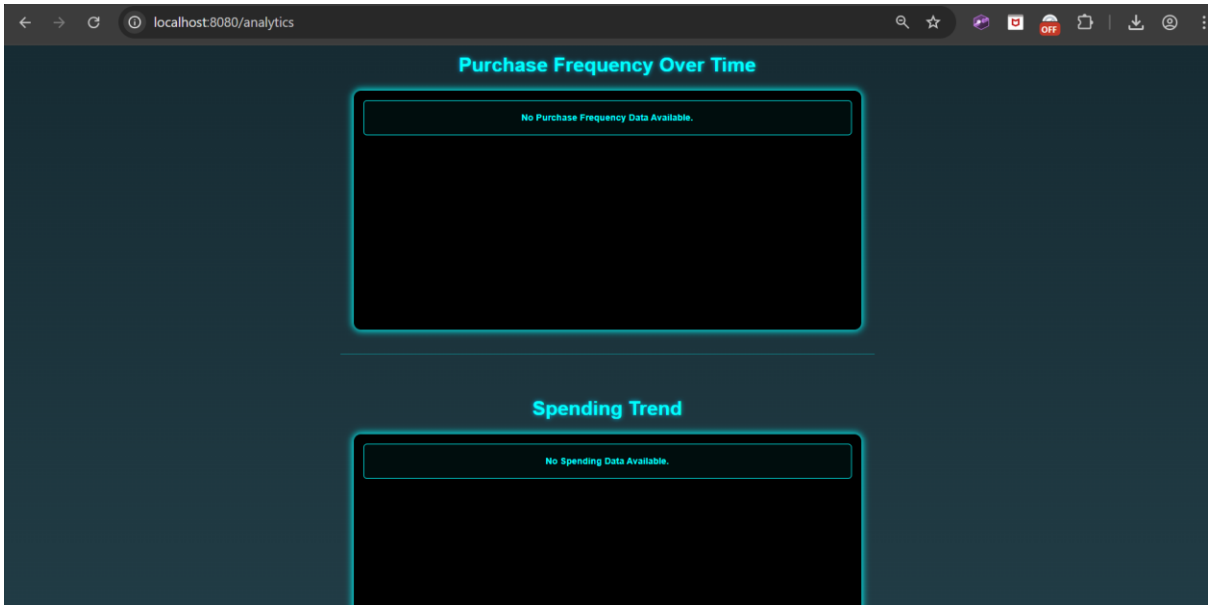
(Downloaded receipt format is the same as it is displayed on the page.)

Analytics Page Testing

Analytics visuals were checked after successful API testing:

- All graphs (Purchase Frequency, Spending Trend, Item Breakdown) correctly rendered using Chart.js.
- If data was missing (e.g., no purchases), fallback "No Data Available" messages appeared instead of blank charts.



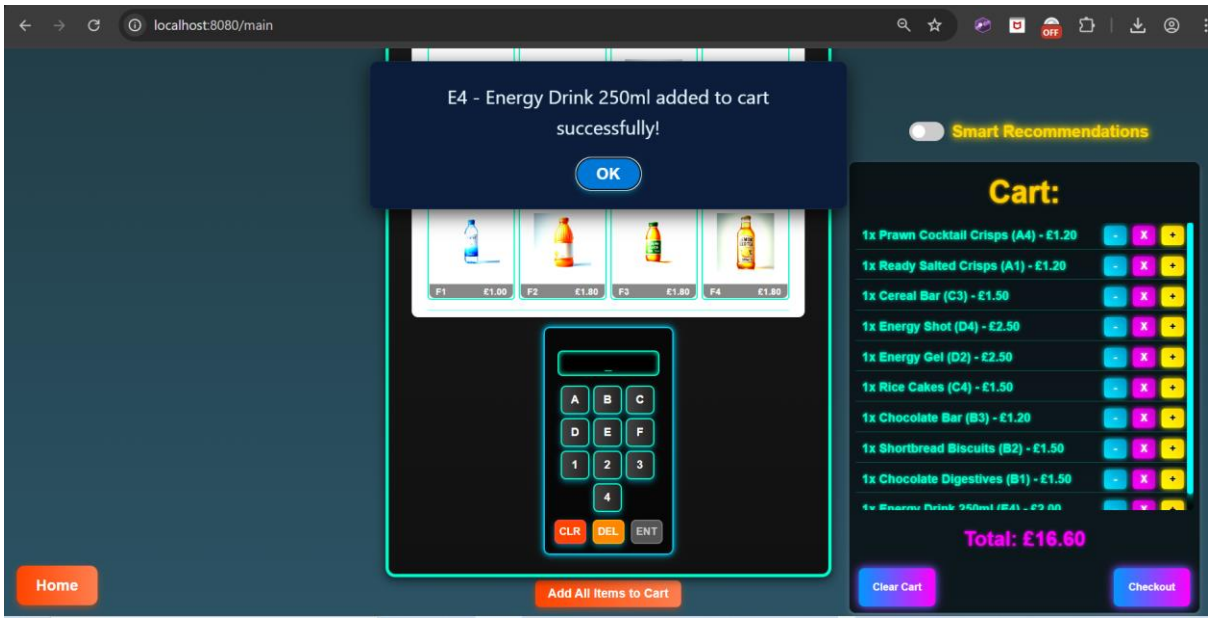


(When a brand-new user with no transactions history accesses the analytics page.)

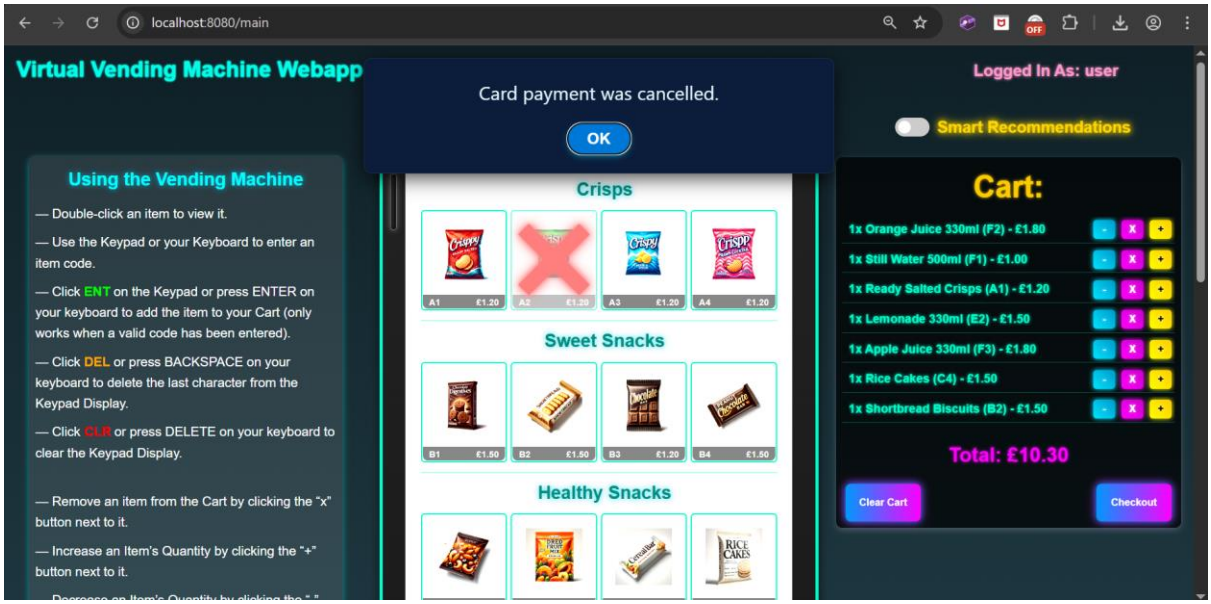
Exploratory Testing

Extensive exploratory interaction (adding and removing multiple products quickly, reloading pages midway through payment, cancelling Stripe payment sessions) confirmed that:

- No unhandled errors occurred.
- Cart recovery and transaction flows were stable and consistent across unexpected user actions.



(Repeatedly typing in item codes into the keypad and pressing enter, all very quickly: handled by the UI with no issues)



(After cancelling a Stripe card payment: handled by the UI, appropriate alert displayed, full cart remained.)

Summary of Frontend Testing

All frontend pages, animations, interactions and visuals behaved exactly as expected across major browsers. Dynamic AJAX updates, Chart.js graphs and page transitions worked without crashes or critical errors, demonstrating a stable and user-friendly interface.

6.4 Unit Testing

Unit testing was added to the Virtual Vending Machine Web Application to test the core backend service logic, independently of the database and frontend. The two most critical backend service classes 'ProductService' and 'TransactionService' were selected for testing, due to their importance to the system's operations (stock control, transaction creation, etc).

Unit tests were written using JUnit 5 and Mockito and executed inside IntelliJ IDEA. Mock repositories were injected into the service classes to isolate the methods and validate their individual functionality.

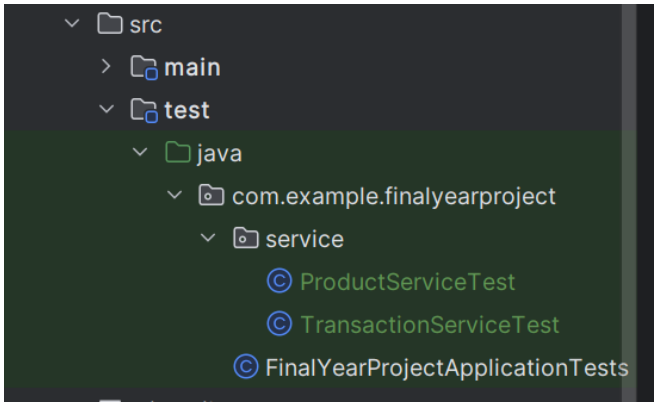
Test Setup and Structure

Test Classes Created:

- 'ProductServiceTest.java'
- 'TransactionServiceTest.java'

Test Location:

- Created in 'src/test/java/com/example/finalyearproject/service/' following standard project structure.



Each test class:

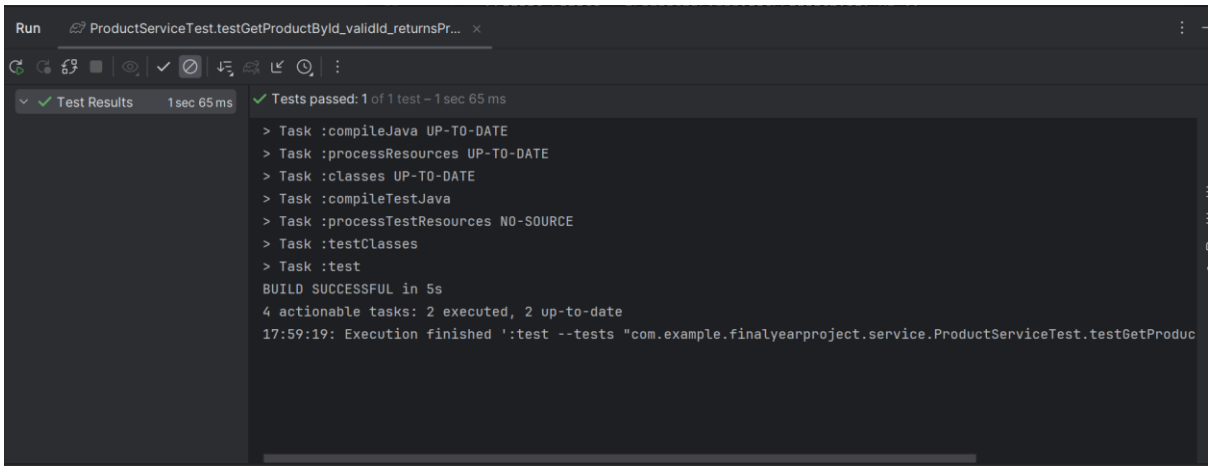
- Uses '@ExtendWith(MockitoExtension.class)'.
- Mocks the relevant repository ('ProductRepo', 'TransactionRepo') using '@Mock' annotation.
- Injects the service being tested using '@InjectMocks'.

Testing ‘ProductService’

1. Test: ‘getProductById()’

- Objective: Verify that given a valid product ID, the correct Product is returned.

```
new *
@BeforeEach
void setUp() {  //set up a sample product for each test
    sampleProduct = new Product();
    sampleProduct.setId("A1");
    sampleProduct.setStock(10);
}

new *
@Test  //test to find a product by its ID
void testGetProductById_validId_returnsProduct() {
    when(productRepo.findById("A1")).thenReturn(Optional.of(sampleProduct));

    Product result = productService.getProductById("A1");

    assertNotNull(result);
    assertEquals( expected: "A1", result.getId());
    verify(productRepo, times( wantedNumberOfInvocations: 1)).findById("A1");
}
```



- Test Result: Pass.

2. Test: ‘updateStock()’

- Objective: Verify that stock updates correctly and handles negative values.

```
new *
@Test //test to update/increase a product's stock
void testUpdateStock_increaseStock_success() {
    when(productRepo.findById("A1")).thenReturn(Optional.of(sampleProduct));

    productService.updateStock( id: "A1",  quantity: 5);
    assertEquals( expected: 15, sampleProduct.getStock());
}
```

```
Run  ProductServiceTest.testUpdateStock_increaseStock_succ...
Test Results 917 ms
Tests passed: 1 of 1 test – 917 ms
> Task :compileJava UP-TO-DATE
> Task :processResources UP-TO-DATE
> Task :classes UP-TO-DATE
> Task :compileTestJava UP-TO-DATE
> Task :processTestResources NO-SOURCE
> Task :testClasses UP-TO-DATE
> Task :test
BUILD SUCCESSFUL in 2s
4 actionable tasks: 1 executed, 3 up-to-date
18:03:16: Execution finished ':test --tests "com.example.finalyearproject.service.ProductServiceTest.testUpdateSto
```

```
new *
@Test  //test to update/decrease a product's stock to a negative value to check it is handled
void testUpdateStock_negativeStock_throwsException() {
    when(productRepo.findById("A1")).thenReturn(Optional.of(sampleProduct));

    assertThrows(IllegalArgumentException.class, () -> {
        productService.updateStock( id: "A1",  quantity: -20);
    });
}
```

```
Run  ProductServiceTest.testUpdateStock_negativeStock_thro...
Test Results 960 ms
Tests passed: 1 of 1 test – 960 ms
> Task :compileJava UP-TO-DATE
> Task :processResources UP-TO-DATE
> Task :classes UP-TO-DATE
> Task :compileTestJava UP-TO-DATE
> Task :processTestResources NO-SOURCE
> Task :testClasses UP-TO-DATE
> Task :test
BUILD SUCCESSFUL in 2s
4 actionable tasks: 1 executed, 3 up-to-date
18:04:38: Execution finished ':test --tests "com.example.finalyearproject.service.ProductServiceTest.testUpdateSto
```

- Result: Pass.

Testing TransactionService

3. Test: ‘createTransaction()’

- Objective: Verify that a transaction is created when given valid cart and payment details.

```
@Test //test for creating a transaction entry
void testCreateTransaction_validInput_createsTransaction() {
    Map<String, Integer> cart = new HashMap<>(); //create a cart
    cart.put("A1", 2);

    Product p = new Product(); //create the product in the cart
    p.setId("A1");
    p.setPrice(1.5);
    p.setStock(5);
    when(productRepo.findAllById(Set.of("A1"))).thenReturn(List.of(p));
    when(productRepo.findById("A1")).thenReturn(Optional.of(p));
    when(transactionRepo.save(any(Transaction.class))).thenAnswer(invocation -> {
        Transaction tx = invocation.getArgument( index: 0);
        tx.setId(1L); //fake Transaction ID to prevent it being null
        return tx;
    });

    Transaction result = transactionService.createTransaction(cart, paymentReceived: 5.0, user

    assertNotNull(result);
    assertEquals( expected: 3.0, resu
    assertEquals( expected: 5.0, resu
    assertEquals( expected: "user", result.getUser());
    assertEquals( expected: 1, result.getTransactionProducts().size());
}
```

```
com.example.finalyearproject.service.TransactionServiceTest

@InjectMocks
private TransactionService transactionService

FinalYearProject.test
```

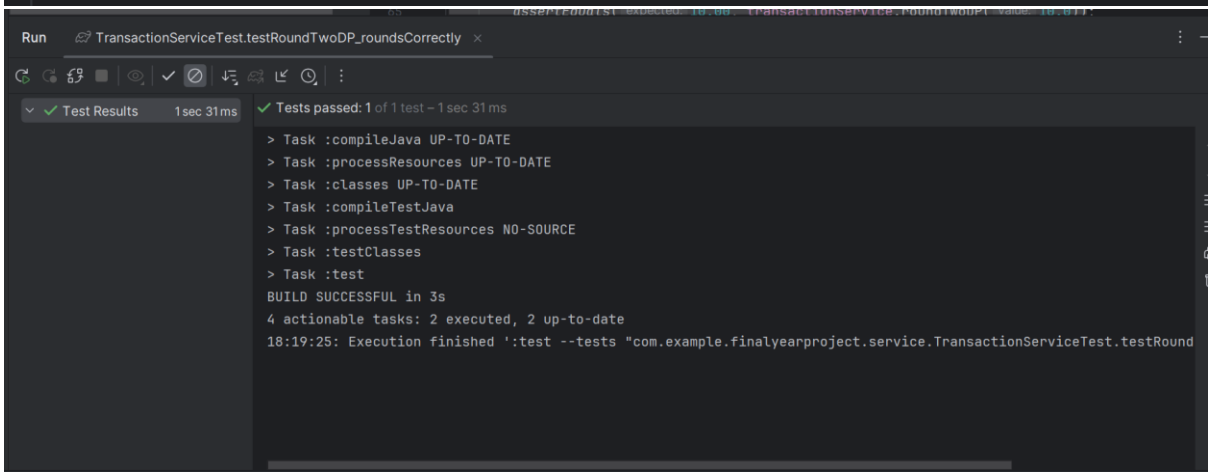
```
Run  TransactionServiceTest.testCreateTransaction_validInput...

Test Results  1 sec 226 ms
Tests passed: 1 of 1 test – 1 sec 226 ms

> Task :compileJava UP-TO-DATE
> Task :processResources UP-TO-DATE
> Task :classes UP-TO-DATE
> Task :compileTestJava
> Task :processTestResources NO-SOURCE
> Task :testClasses
New Transaction 1 saved for user: user with 1 product entries.
> Task :test
BUILD SUCCESSFUL in 5s
4 actionable tasks: 2 executed, 2 up-to-date
18:17:01: Execution finished ':test --tests "com.example.finalyearproject.service.TransactionServiceTest.testCreat
```

- Test Result: Pass.

4. Test: ‘roundTwoDP()’

- Objective: Verify that decimal numbers are correctly rounded to two decimal places.

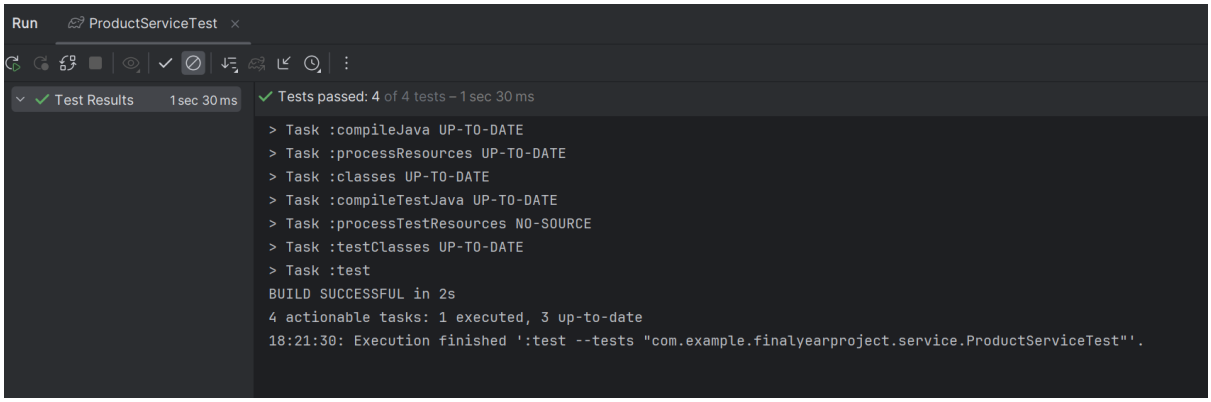
```
new *
@Test //test for the rounding helper method (required for all money values)
void testRoundTwoDP_roundsCorrectly() {
    //expect all numbers to be correctly rounded AND 2 decimal places
    assertEquals( expected: 2.67, transactionService.roundTwoDP( value: 2.666));
    assertEquals( expected: 5.99, transactionService.roundTwoDP( value: 5.994));
    assertEquals( expected: 10.00, transactionService.roundTwoDP( value: 10.0));
}
}
```

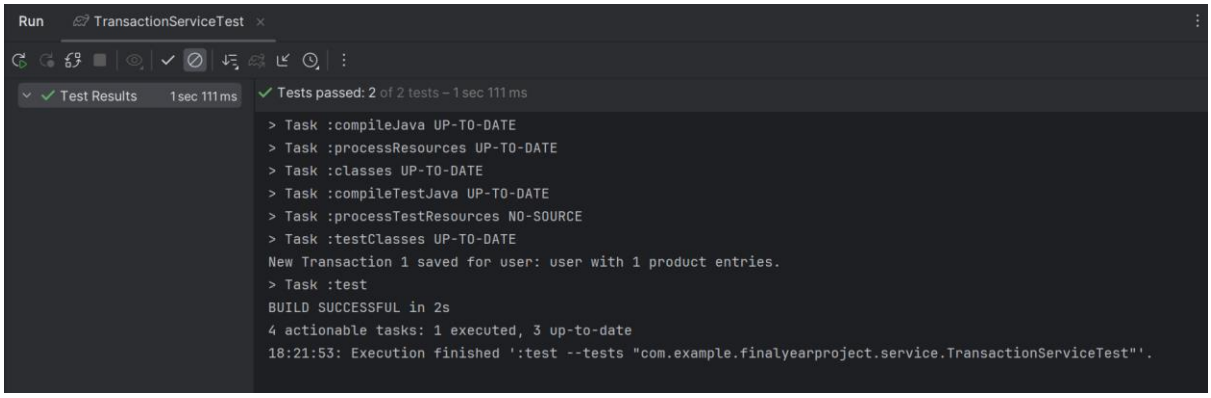


- Test Result: Pass.

Unit Testing Summary

- After writing all tests, the full test suite was run using “Run All Tests” option in IntelliJ IDEA.
- All tests passed together successfully, ensuring that critical individual backend service methods operate as intended independently of the live database.





Full testing of additional features such as transaction history, analytics, card payments and the smart recommendation algorithm are included in ***Appendix D – Additional Features Testing****.*

7. Evaluation

7.1 System Performance Evaluation

The Virtual Vending Machine Web Application demonstrated a high level of technical performance across backend processing, frontend usage and data communication during testing and exploratory stages. Backend API endpoints consistently returned results within acceptable response times (under 500ms for most requests during local development), ensuring smooth data retrieval and transaction processing. RESTful API interactions, particularly for transaction creation, payment session initiation and analytics data retrieval were handled asynchronously, allowing the frontend to remain responsive even during database reliant operations.

Frontend performance was optimised through efficient JavaScript event handling and AJAX requests, avoiding full page reloads during user interactions such as cart updates, coin insertion and payment confirmation. During exploratory testing with rapid user input or stress test scenarios (e.g. fast multiple item selection and removal), the UI remained stable without crashes or excessive delays.

While scalability for larger datasets (e.g. thousands of transactions) was not formally load-tested due to project scope, the system architecture (class design, separation of concerns, REST APIs, DTOs for lightweight data transfer) provides a strong foundation for future scalability improvements if required.

7.2 Feature Evaluation

The finalised system successfully delivered all core features (and some additional features) outlined in the initial project aims. The main vending machine interface provided two responsive and usable product selection systems and payment methods, closely simulating the actions of a physical vending machine. Features such as dynamic cart management, real-time stock updates and both coin and card payment methods worked together smoothly to provide the core experience. The success modal and receipt generation added an engaging end to each purchase.

The Transaction History page added significant value by allowing logged-in users to track previous purchases. Sorting and filtering functionality were implemented in a dynamic and user-friendly way, and testing confirmed this worked reliably with varied transaction data. The “View Receipt” function enabled quick access to past receipts, and all data was protected by secure user-based filtering.

The Admin Dashboard proved to be a powerful control panel for potential system maintainers. Stock management was quick and easy to use, and the auto-restock system worked exactly as intended, only activating at the correct threshold levels. The interface allowed full batches of stock setting adjustments, undo/reset of changes and confirmed admin-only access through route protection. System logs and database testing verified that changes were correctly committed to the backend.

The Smart Recommendations was designed to be one of the more technically advanced features. It adapted in real time based on a user's purchase behaviour and demonstrated a ML type multi-factor scoring algorithm, combining basic collaborative filtering with user-specific heuristics (e.g. category preference, spending etc). Testing showed that these recommendations dynamically changed between users and even evolved over time, which confirmed the effectiveness of the algorithm logic. The frontend visual feedback, including tooltips and glowing highlights further enhanced the user experience gained from the recommendations.

Overall, every feature that was intended to be delivered was fully implemented and verified through testing, and each contributed meaningfully to the overall experience and functionality of the system.

7.3 Frontend Usability and UI Design

An important focus from the beginning was to create a strong user interface, as many existing vending machine simulations were found to be visually outdated or difficult to use. I placed strong emphasis on building a user-friendly experience that looked professional but also engaging and functioned well.

All core user actions: browsing, adding to cart and completing a purchase were designed to feel smooth, with real-time updates and UI transitions. Features like coin animations, product highlights, and glowing alerts helped keep the user attentive through each stage. The layout remained consistent across pages, and important elements were always visible.

All of this helped to maintain the high standards of the UI throughout development and allowed me to keep it consistent through all pages and features.

7.4 Limitations and Improvements

While the Virtual Vending Machine Web Application met its core objectives and delivered a wide range of features, several limitations were encountered due to time constraints and the chosen decision to prioritise more critical functionalities.

Recommendation Algorithm Scope
The recommendation system successfully incorporates multiple scoring factors but remains relatively basic in its collaborative filtering logic. A more sophisticated algorithm (e.g. matrix factorisation or user clustering) could provide more detailed and personalised results.
Improvement: In future iterations, this could be extended with machine learning libraries or database-driven collaborative analysis.

Admin Functionality Depth
While the admin dashboard allows full stock control and auto-restock configuration, there is no interface for managing users or creating new products. I was aware of this but had to decide to leave it out due to time constraints and the need to prioritise implementing other features.
Improvement: There are numerous potential future improvements e.g. could include full CRUD control for products, live monitoring dashboards and user management tools.

Limited Real-World Integration
The Stripe card payment system works in test mode only. Real payments and secure web deployment were outside the scope for this project.
Improvement: With further time, if the system was to be deployed and commercialised, we could use a live Stripe account for real card payments.

Testing Coverage
Comprehensive manual testing was completed across the entire project, mostly during development (and unfortunately not documented) but automated testing (unit, integration) was limited to only the critical backend services.
Improvement: Expanding JUnit test coverage would increase reliability and make future updates safer.

Mobile Responsiveness
The frontend UI was designed primarily for desktops. Although some UI components adapt to smaller screens, it is not fully optimised for mobile or tablet use.
Improvement: Additional CSS media queries and layout adjustments would allow full cross-device support.

These limitations reflect executive decisions on prioritising certain aspects during the development process. With finite time and a large system scope, features were prioritised based on value, technical complexity, available time and my own relevant knowledge, with the primary focus placed on delivering a complete, well-tested and user-friendly core experience.

7.5 Requirements Summary

Overall, all functional requirements were met. Every feature proposed was successfully implemented in and everything worked as intended. The extent to which some were met can be questioned, for example "**5. Change Dispensing"**, the change for transactions was correctly calculated and displayed but wasn't shown in coin denominations and there was no simulation of coins being dispensed. Additionally, "**10. Analytics Page**" and "**11.Product Recommendations**" were fairly limited based on their full potentials (as mentioned above) however these were extended features and weren't prioritised and they were still implemented to an acceptable level, so these were met to a justifiable extent.

Most non-functional requirements were met but "**19. Testing and Validation**" wasn't met to the standard I expected due to a lack of Junit tests created. As mentioned above, the reason for this was a lack of time and a word count restriction which limited how many I could document in this report. Aside from this the other non-functional requirements were met to a high standard.

8. Contextual Discussion

The need for this project was originally created by identifying a clear gap in quality across existing vending machine simulation systems. From the start, I noticed that most were outdated in design, lacked features, and didn't make full use of the web development stack. My goal was to change that by building something modern, functional and full stack, with both user experience and technical architecture equally prioritised.

The system's features mirror real-world expectations: responsive UI, smooth product selection, secure payments, admin controls, restocking and transaction logging. These fit in well with the kinds of systems seen in commercial or smart retail environments. While the current project runs in a test and development setup, there is definite potential for launching and commercialisation in the future. With further work e.g. secure deployment, full mobile responsiveness and real payment mode enabled via Stripe this system could be a blueprint for a fully scaled, interactive selling platform. With modern advancements in technology and constant evolution of AI the Virtual Vending Machine has potential to be used as a real-life purchase system, for online shopping, smart kiosks or even hybrid vending machines (mechanically works like a standard vending machine but the UI and advanced features are taken from the virtual vending machine).

From an ethical and security standpoint, I ensured user privacy was respected throughout. All authenticated routes were protected using Spring Security and all passwords were encrypted and securely stored. For OAuth2 logins, users were assigned secure fallback credentials and session cookies were used securely for login persistence. Although the project is only a simulation, these were still essential considerations to take seriously and would be treated the same if it was a fully deployed project.

In terms of sustainability, this system is naturally lightweight: it uses minimal server-side processing and avoids unnecessary loads through efficient API usage and frontend rendering. If deployed, its energy usage would be extremely low, especially compared to larger frameworks or services that remain constantly turned on. By offloading some tasks to the frontend and using lazy AJAX fetching, performance and power efficiency were naturally maximised.

In summary, I developed this project to address a real need but also made sure that every technical decision I made considered future possibilities, user safety and sustainability, not just what was convenient during development and testing.

9. Personal Reflection

My final year project was the most challenging but rewarding experience of my academic journey. From initial planning through to finalisation and testing, I was responsible for managing the entire lifecycle of a full-stack web application something I had not attempted at this level before. It significantly improved my skills in backend development, UI design, database integration and overall software architecture. Aside from technical and scalable motivations for this project, personal was always my biggest motivation: I was determined to put my strengths and skills to good use by creating a fully functional and usable system.

One of the biggest lessons learnt was the importance of balancing my ambitions with time management and other constraints. Throughout the development process, I had to make difficult decisions about which features to prioritise and where to compromise on complexity and detail. While I had initially considered implementing more advanced features (e.g. more admin controls, detailed analytics and deeper ML-based recommendation), I instead focused on ensuring the key functionality was implemented to a high standard and well tested.

My choice of languages, technologies and tools to use for this project was based on my valuable experience and deep understanding of them, through all my academic studies, e.g. IntelliJ IDEA, MySQL, SpringBoot, Java. Some other things I chose to implement based on what I have learned this academic year and wanted to start building a strong understanding of them, e.g. AJAX, Thymeleaf, ML. Finally, I have also explored things that are completely brand new to me, e.g. Stripe, Postman. Through all of this I am proud to showcase my existing talents and newly acquired ones in my final project of my university journey.

This project helped me to understand the value of testing and version control as part of a development workflow. Manual and API testing using Postman and DevTools became an essential part of my process, and unit testing with JUnit gave me an insight into how automated validation strengthens long-term maintainability of real-world systems.

Overall, I am proud of the outcome and believe I have developed invaluable practical skills and professional habits that will benefit me in all my potential future projects and industrial roles. Regardless of the final mark for the system (good or bad) I view this project and everything else I have done alongside it as a huge success.

10. Conclusion

The development of the Virtual Vending Machine Web Application brought together everything I've learned during my Computer Science degree from backend architecture and API design to frontend UI creation, data management testing and security. My goal from the beginning was to build something modern, functional and full stack, with both technical performance and user experience treated equally as priorities. Looking back, I believe this goal was met across every major aspect of the project.

Through my work, I gained a deeper understanding of the end-to-end process of designing and delivering a real software product. I learned the importance of architectural structure, asynchronous data transfer, session handling and validation and how all of these concepts interact in a full-stack environment. Testing gave me a new insight into the importance of stability and edge case handling, and implementing a scoring algorithm for product recommendations showed me how even basic ML ideas can add significant value when applied practically.

Every planned feature was delivered, tested and documented. The system supports browsing, purchasing, payment via coins or card, automatic stock updates, admin controls, transaction history and analytics - all built on a clean MVC architecture with secure access control and dynamic, user-driven interactivity. The Smart Recommendations feature pushed the system further into intelligent design, and despite being relatively basic (debatable), it lays strong foundations for more advanced techniques in the future.

If this project were to be extended, there's real potential for deployment and commercial use. Real payment integration, full mobile responsiveness, detailed stock analytics and deeper recommendation logic using collaborative filtering or AI would be prominent areas for future development. Even beyond web use, the structure of this project could be adapted to hybrid vending machines or smart retail kiosks.

Aside from all the development I have spent lots of time and effort on the documentation aspect: all code is heavily filled with comments, loads of GitLab commits made frequently and consistently throughout the whole development process with detailed commit messages. The rest of my time has been dedicated to the deliverables: a project log (uploaded to GitLab every week) giving insights to the exact progress I have made every week with precise explanations of all improvements, the interim report and this final report dissertation.

To conclude everything, I'm happy with the final product I have built and the skills I've developed along the way. This project wasn't just an academic exercise it was a project that allowed me to apply everything I know to solve a real problem in a professional and creative way. It represents a major step forward in my journey as a developer and a milestone I'll carry into my future work. From all my ideas and plans I have successfully created the Virtual Vending Machine Webapp.

11. Bibliography

[1] N. Ratnasri and T. Sharmilan, “Vending Machine Technologies: A Review Article,” *International Journal of Sciences: Basic and Applied Research*, vol. 58, no. 2, pp. 160–166, 2021. [Online]. Available: https://www.researchgate.net/profile/Tharaga-Sharmilan/publication/357577286_Vending_Machine_Technologies_A_Review_Article/links/61d544a5b8305f7c4b23188d/Vending-Machine-Technologies-A-Review-Article.pdf

[2] A. Solano, N. Duro, R. Dormido, and P. González, “Smart vending machines in the era of Internet of Things,” *Future Generation Computer Systems*, vol. 76, pp. 215–220, Nov. 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X16304757

[3] M. Seim, *Exploring Design Principles for Self-Service Technologies: The Case of a Ticket Vending Machine*, M.S. thesis, Dept. of Information Science and Media Studies, Univ. of Bergen, Norway, 2014. [Online]. Available: https://bora.uib.no/bora-xmlui/handle/1956/8522

[4] PlayHop, "Vending Machine Simulator," PlayHop, [Online]. Available: https://playhop.com/app/339874.

[5] D. Roberts, "VIRTUAL VENDING MACHINE: An interactive vending machine. Weekly game design project," Dennis Roberts, [Online]. Available: https://dennisroberts.com/filter/GAME/VIRTUAL-VENDING-MACHINE.

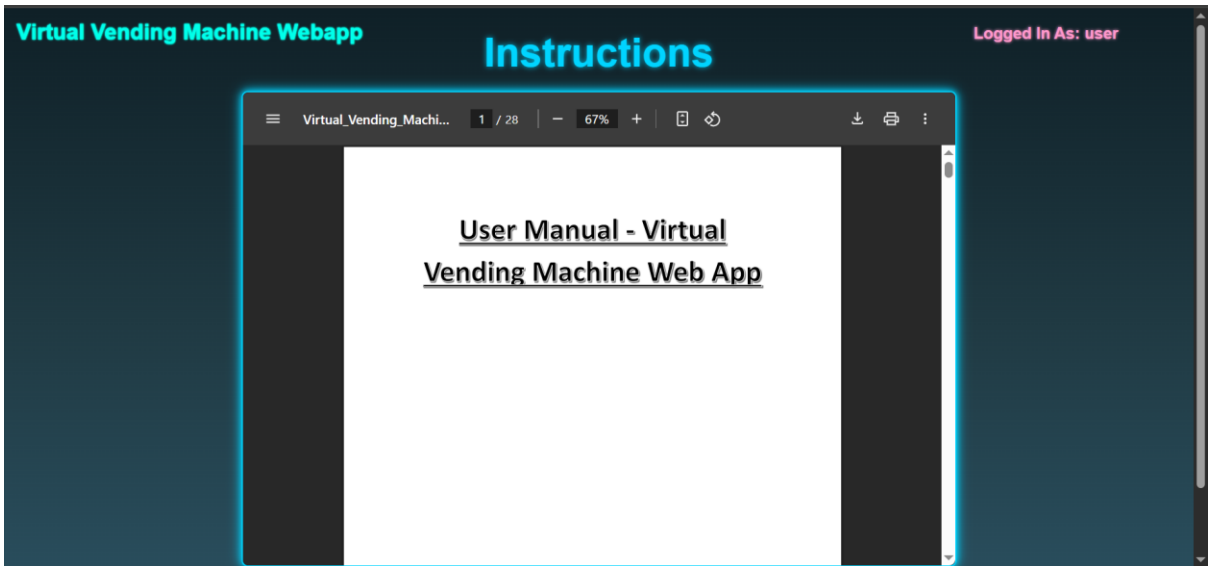
[6] ArcadeSpot, "Surprise Eggs Vending Machine," ArcadeSpot, [Online]. Available: https://arcadespot.com/game/surprise-eggs-vending-machine/.

12. Appendices

Appendix A – Extra Documentation

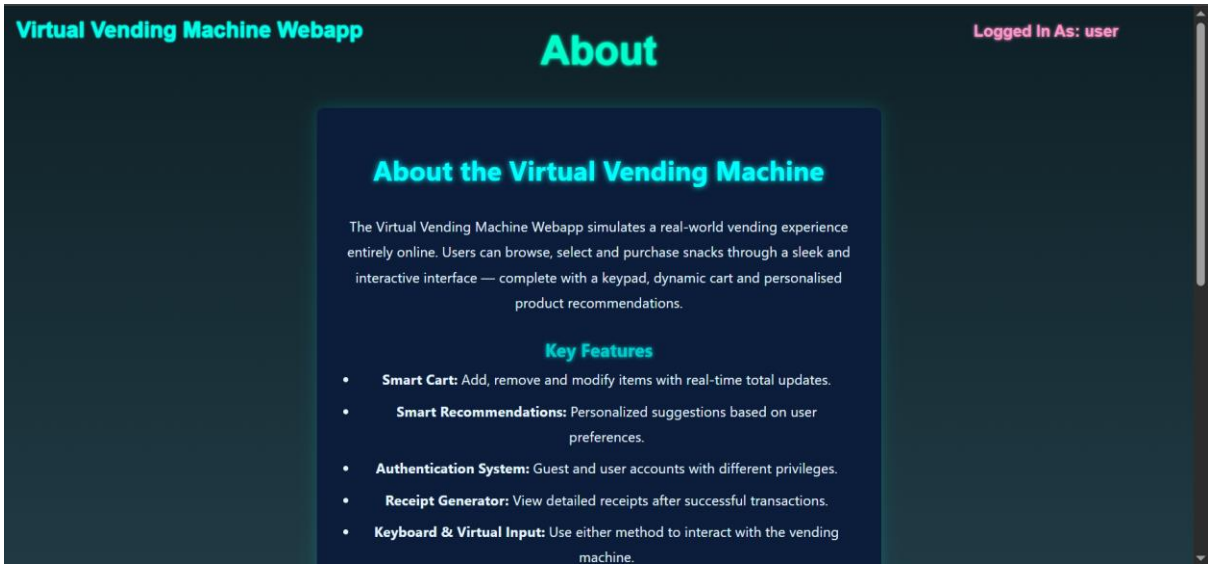
The project is supported by a detailed **README file** on GitLab, which outlines all technical requirements and startup instructions for running the application.

A full User Manual is included in the '/resources/static/pdf/' directory. It explains how to navigate the application, covering all key features and UI interactions. The manual is also directly accessible within the UI on the 'Instructions Page'.

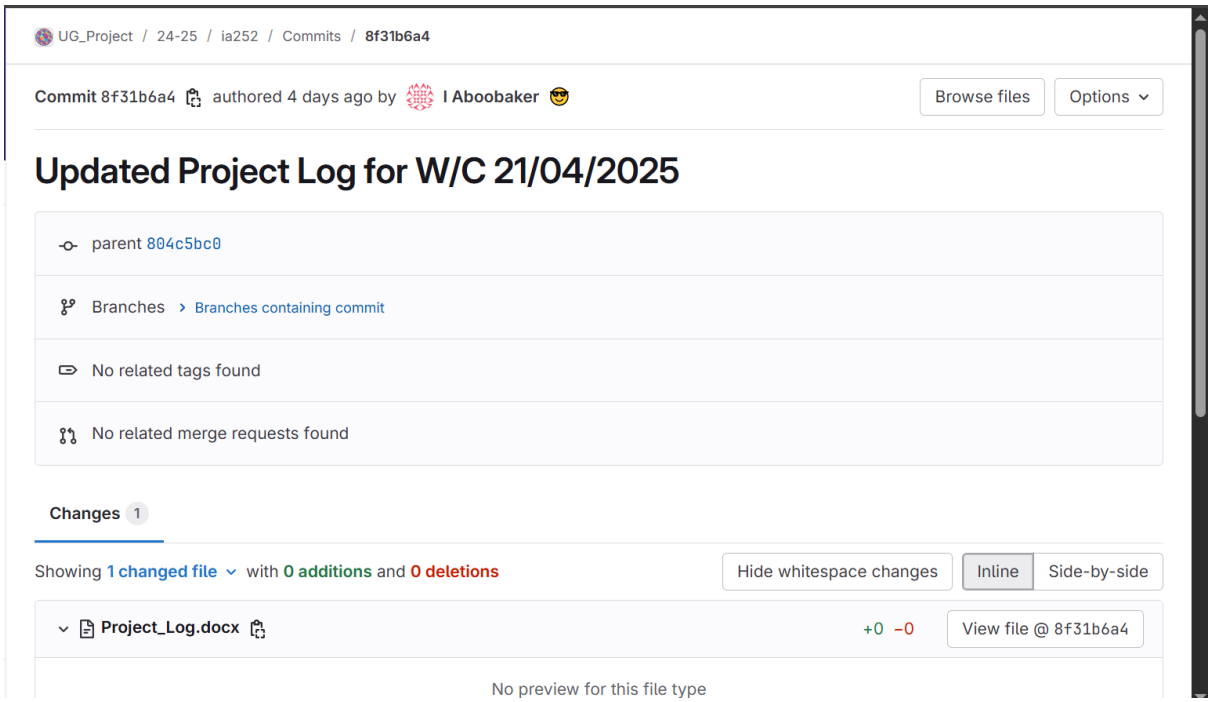


It comes with a "Fullscreen" button to get a full view of the manual and a "Download" button to download a copy of the User Manual as a PDF, so it is always readily available for offline use.

In addition, the ‘About Page’ offers a general overview of the system, including useful context and key features in a simplified and engaging format.



The final piece of extra documentation is the Project Log, which has been uploaded to GitLab every week. Each new entry provides a detailed and specific summary of what was completed during that week.



(An example upload from last week.)

Project Log entries have been added consistently, every week without failure, since 21/10/2024.

Appendix B - Implementation Diagrams, Code Snippets and Screenshots

This appendix contains supplementary diagrams, screenshots and code snippets that demonstrate key implementation aspects of the Virtual Vending Machine Web Application. It includes project structure overviews, dynamic cart and payment handling functions, and visual representations of the main vending interface, payment modal, transaction confirmation and receipt pages.

Figure B1 - Project Folder Structure

An overview of the project's backend and frontend structure as seen in the development environment (IntelliJ IDEA).

Fig. B1.1: 'src' folder view, controllers and models:

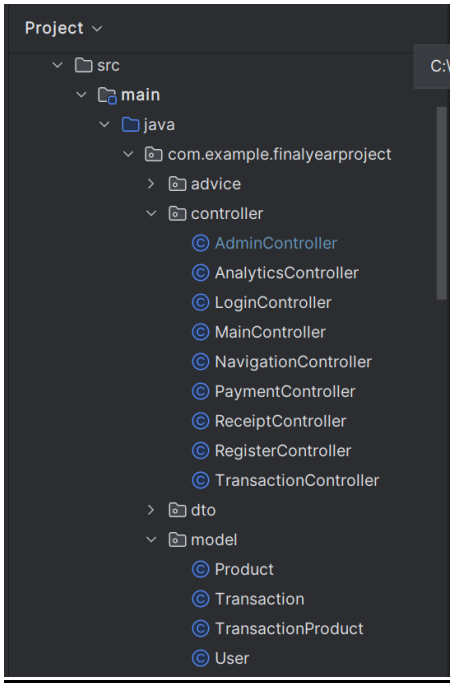


Fig. B1.2: Repositories, Security, Services and Application:

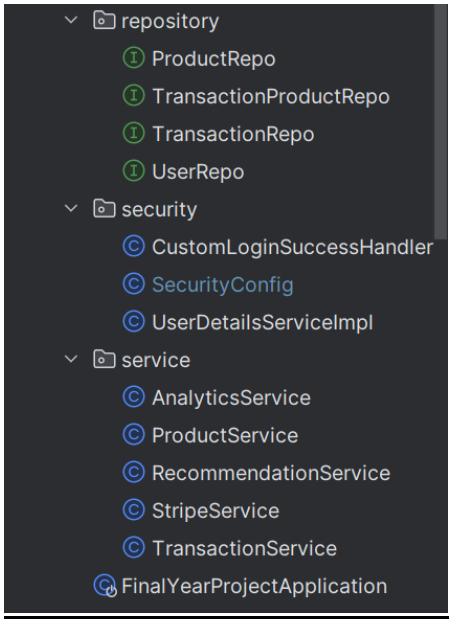
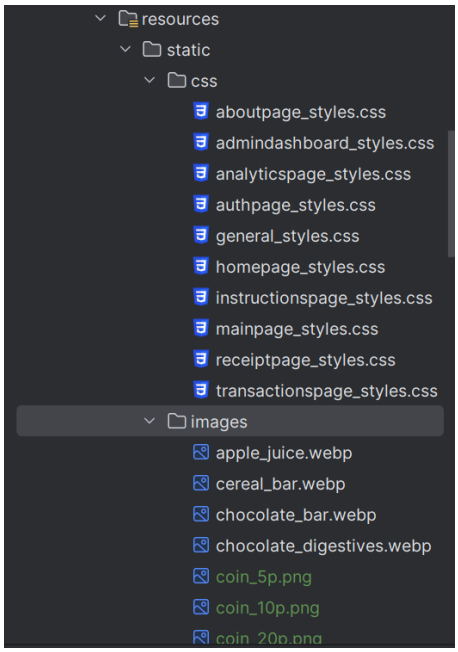


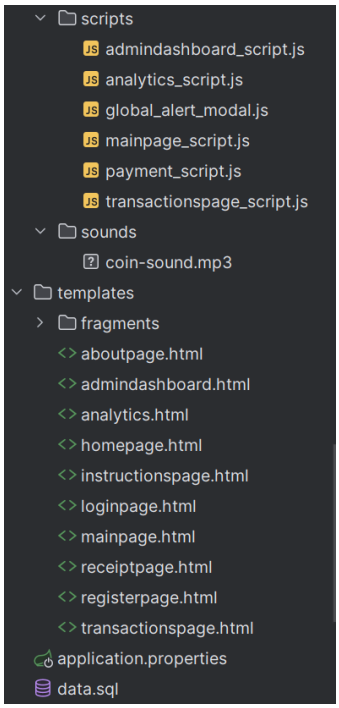
Fig. B1.3: 'resources' folder (frontend), CSS and Images:



(Note: 'images' contains all product images and all coin icons. There's too many to fit in one screenshot so not all of them are shown).

(Note: All Product Images and Coin Icon Images were AI generated images.)

Fig. B1.4: Scripts (JS), Sounds, Templates (HTML), 'application.properties' and 'data.sql':



(Note: 'data.sql' is a file used to insert product data upon running the application. 'fragments' contains a HTML fragment which is a modal globally used in all pages to show alerts (in place of standard JS alerts).

Fig. B1.5: Overall directory view:

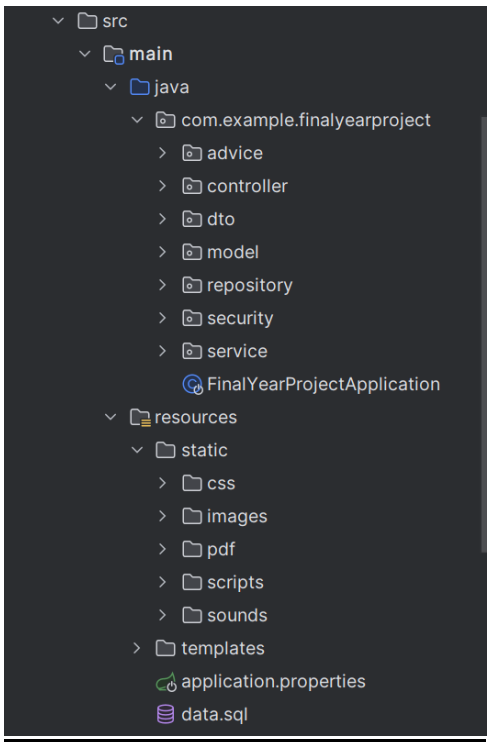


Figure B2 – JavaScript Cart 'addItemToCart(product)' and 'updateCartDisplay() Functions

Code snippet showing the functions that handle adding products to the cart and updating the virtual cart in real-time:

```
//function to dynamically add an item to the cart
async function addItemToCart(product) {
    let enoughStock = await checkStock(product);  //call stock check
function and await it

    if (!enoughStock) return;  //if not enough stock return from
function, the alert will already be sent by the checkStock function

    //if enough stock proceed with adding item to cart (or updating
quantity)
    if (cartItems[product.id]) {  //checking if the product already
exists in the cart
        cartItems[product.id].quantity += 1; //increment its
quantity if  already in the cart
    } else {
        cartItems[product.id] = {  //add as new item if not already
in cart
            name: product.name,
            price: parseFloat(product.price), //parse the price as a
float
            quantity: 1,
            imageUrl: product.imageUrl  //also store image url
        };
    }

    showAlert(`${product.id} - ${product.name} added to cart
successfully!`); //successful alert

    updateCartDisplay();  //calling function to update the cart
display after adding an item
}
```

(The helper function 'checkStock()' will make an AJAX call to the backend with all products and their quantities currently in the cart to double check that there is enough stock for each. If it fails, this function will exit and an alert will be displayed.)

```
//function to update cart display
function updateCartDisplay() {
    let cartList = document.getElementById("cartList");
```

```
    let checkoutButton = document.getElementById("checkoutBtn");
    let clearCartButton = document.getElementById("clearCartBtn");

    cartList.innerHTML = "";  //clears cart to avoid duplication

    for (let itemCode in cartItems) { //iterate through cart items
        let item = cartItems[itemCode];
        let itemTotalPrice = (item.price *
item.quantity).toFixed(2); //handle item(s) price

        let listItem = document.createElement("li"); //create new
html list tag
        listItem.classList.add("cart-item");

        //create text container for product name & price
        let itemText = document.createElement("span");
        itemText.classList.add("cart-item-text");
        itemText.textContent = `${item.quantity}x ${item.name}
(${itemCode}) - £${itemTotalPrice}`;

        //create button container
        let buttonContainer = document.createElement("div");
        buttonContainer.classList.add("cart-item-buttons");

        //create reduce button
        let reduceBtn = document.createElement("button");
        reduceBtn.innerText = "-";
        reduceBtn.classList.add("reduce-btn");
        reduceBtn.onclick = function() {
            reduceItem(itemCode); //call function to reduce item
        };

        //create remove button
        let removeBtn = document.createElement("button");
        removeBtn.innerText = "X";
        removeBtn.classList.add("remove-btn");
        removeBtn.onclick = function() {
            removeItem(itemCode); //call function to remove item
        };

        //create Increase button
        let increaseBtn = document.createElement("button");
        increaseBtn.innerText = "+";
        increaseBtn.classList.add("increase-btn");
        increaseBtn.onclick = function() {
            increaseItem(itemCode);
        };

        //append buttons to button container
        buttonContainer.appendChild(reduceBtn);
        buttonContainer.appendChild(removeBtn);
        buttonContainer.appendChild(increaseBtn);

        //append text and button container to list item
        listItem.appendChild(itemText);
```

```
        listItem.appendChild(buttonContainer);

        cartList.appendChild(listItem); //append list item to cart
list
    }

    updateTotal();  //call function to update total price

    if (Object.keys(cartItems).length === 0) {  //if cart is empty
        checkoutButton.disabled = true;   //disable checkout button
        clearCartButton.disabled = true;  //disable clear cart
button
    } else {                            //if cart is not empty
        checkoutButton.disabled = false;  //enable checkout button
        clearCartButton.disabled = false; //enable clear cart button
    }
}
```

(The helper function 'updateTotal()' updates the dynamic running total of the cart by multiplying the price of each item in the cart by its quantity and summing all of these up.)

Figure B3 – JavaScript Payment 'insertCoin(value)' and 'updateInsertedAmount' Functions

Code snippet showing how coin insertion is simulated, animated into the vending machine and how the inserted payment total is updated dynamically:

```
//function to simulate inserting a coin into the machine, using
animations
function insertCoin(value, event) {
    const coinElement = event.target;  //get the coin element
    const animatedCoin = coinElement.cloneNode(true);  //clone the
coin
    animatedCoin.classList.add("coin-fly");  //add the fly class to
the clone
    animatedCoin.style.opacity = "1"; //ensure full visibility

    document.body.appendChild(animatedCoin);  //add the coin clone
to page

    const coinRect = coinElement.getBoundingClientRect();

    //set the dimensions of the cloned coin based on the original
coin's size
    animatedCoin.style.width = `${coinRect.width}px`;
    animatedCoin.style.height = `${coinRect.height}px`;

    //position the clone at the same position as the original
```

```
    animatedCoin.style.left = `${coinRect.left}px`;
    animatedCoin.style.top = `${coinRect.top}px`;

    //get the vending machine position
    const vendingMachine = document.querySelector(".vending-
machine");
    const vendingRect = vendingMachine.getBoundingClientRect();

    //calculate x and y distance to the vending machine (add on
small margins for error)
    const offsetX = vendingRect.left - coinRect.left + 10;
    const offsetY = vendingRect.top - coinRect.top + 20;

    //animate the coin movement by translating to the offset
position
    setTimeout(() => {
        animatedCoin.style.transform = `translate(${offsetX}px,
${offsetY}px)`;
    }, 10);

    //when the coin reaches the desired position: make it disappear
and trigger the machine glow
    setTimeout(() => {
        animatedCoin.remove(); //remove the coin
        vendingMachine.classList.add("glow-pink");  //add the glow
        setTimeout(() => vendingMachine.classList.remove("glow-
pink"), 400);  //remove the glow after the timeout
    }, 800);

    //play the sound
    const sound = new Audio('/sounds/coin-sound.mp3');
    setTimeout(() => {
        sound.play();
    }, 1000);  //play after 2 seconds

    //hide the card button after coin insertion (don't allow coins
and card combined for payments)
    const cardSection =
document.getElementById("cardPaymentSection");
    if (cardSection) {
        cardSection.style.display = "none";
    }

    updateInsertedAmount(value); //update the payment values
}
```

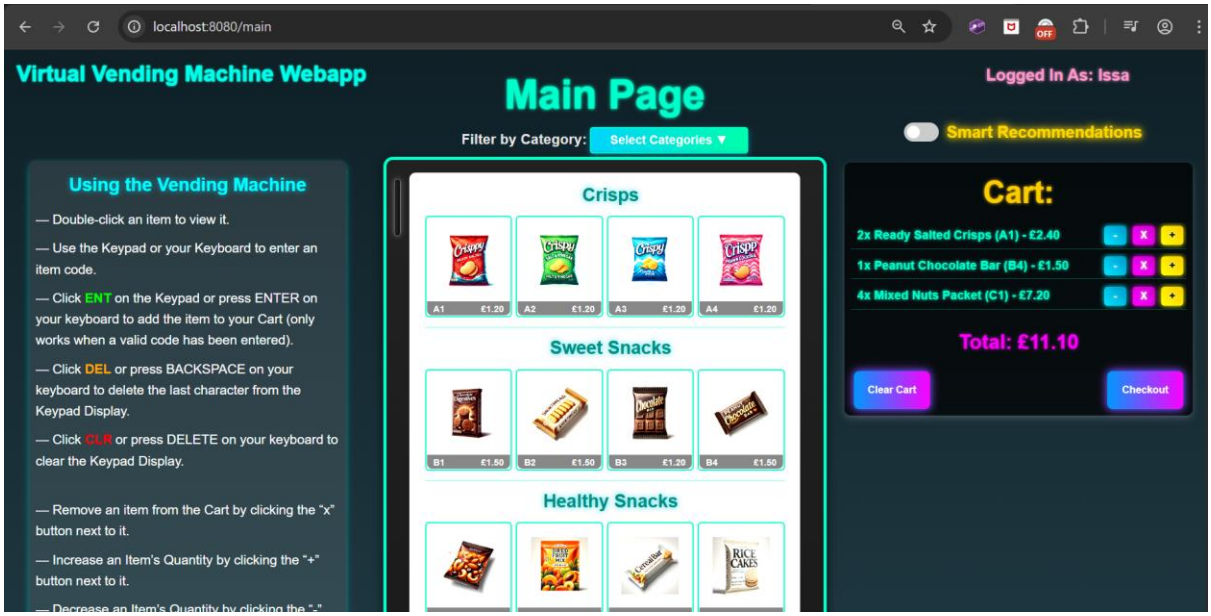
(Animates coin movement into the coin slot by calculating the position of the coin slot based on its distance, creating a clone coin of the one to be inserted and translating it to the desired position [with CSS animation styles].)

```
//function to update the amounts after a coin is inserted for
payment
function updateInsertedAmount(amount) {
    if (remainingAmount > 0) {      //only work if remaining amount
exceeds 0
        insertedAmount = +(insertedAmount + amount).toFixed(2);
//increment inserted amount by the coin value and round to prevent
Floating Point Error
        remainingAmount = +(remainingAmount - amount).toFixed(2);
//decrement remaining amount by the coin value and round to prevent
Floating Point Error

        //checking if payment can be completed
        if (remainingAmount < 0) {  //if overpaid
            changeAmount = Math.abs(remainingAmount); //convert to
positive value
            remainingAmount = 0;


//display change and confirm button elements
            document.getElementById("changeAmount").innerText =
`£${changeAmount.toFixed(2)}`;
            document.getElementById("changeContainer").style.display
= "block";
            document.getElementById("confirmPayment").style.display
= "block";
        }
        else if (remainingAmount === 0) {  //if paid exactly display
only the confirm button (no change)
            document.getElementById("confirmPayment").style.display
= "block";
        }

        //dynamically updates inserted and remaining amounts in the
UI
        document.getElementById("insertedAmount").innerText =
`£${insertedAmount.toFixed(2)}`;
        document.getElementById("remainingAmount").innerText =
`£${remainingAmount.toFixed(2)}`;

    }
}
```

(Updates all HTML elements after a coin insert, including the remaining amount to pay and the total inserted. Also checks if the full amount has been paid, if so, it displays the confirm payment button. Additionally, if the payment has exceeded the total to pay, it will display the change value too.)

Figure B4 – Main Vending Machine Page

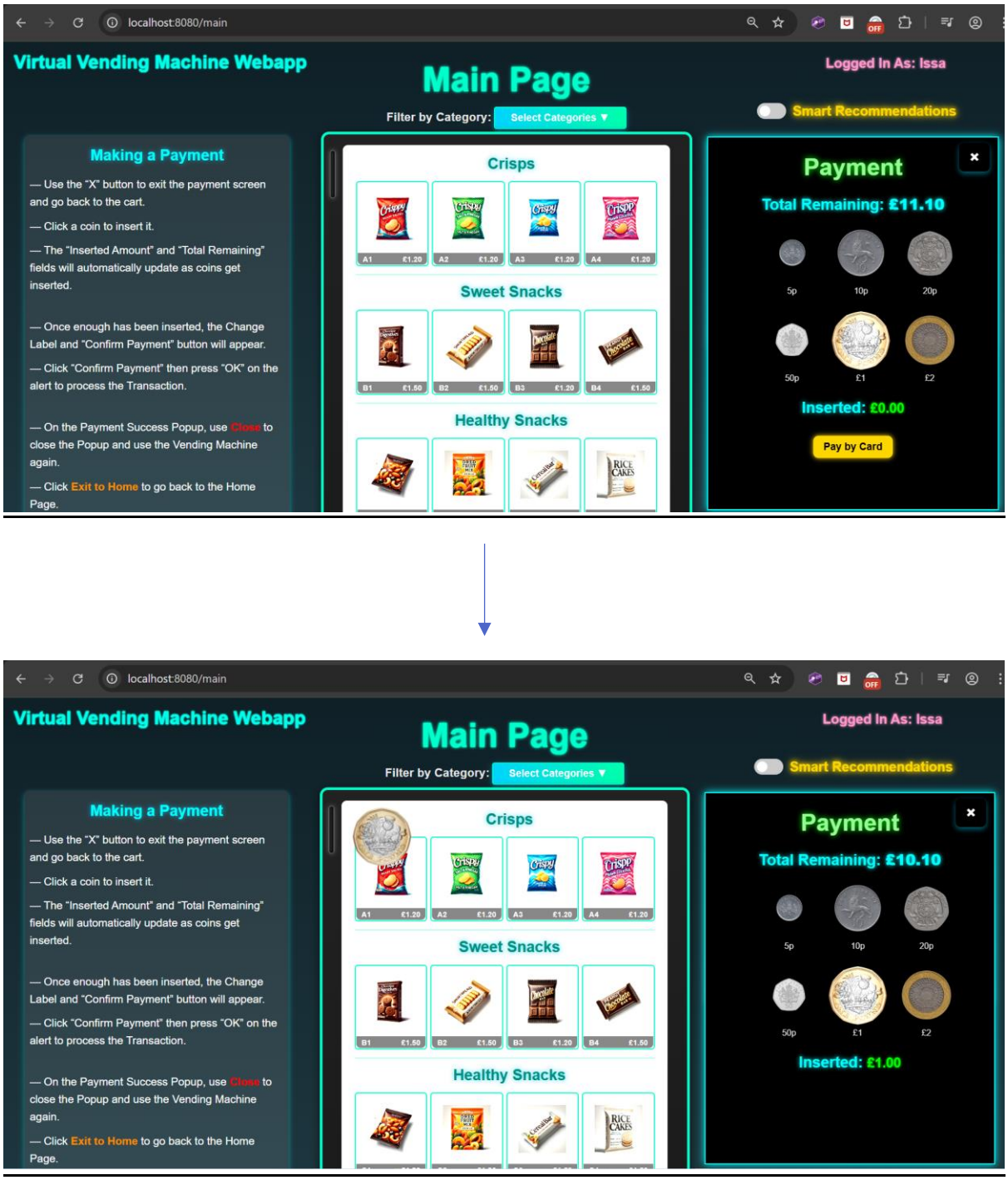
Screenshot showing the vending machine product grid layout, the dynamic cart next to it and the on-page instructions to accompany them.



(Note: The "Select Categories" toggle can be used to filter between different product categories and the vending machine grid will automatically change based on filter choices.)

Figure B5 - Payment Interface Screenshot (Coin Insertion Section)

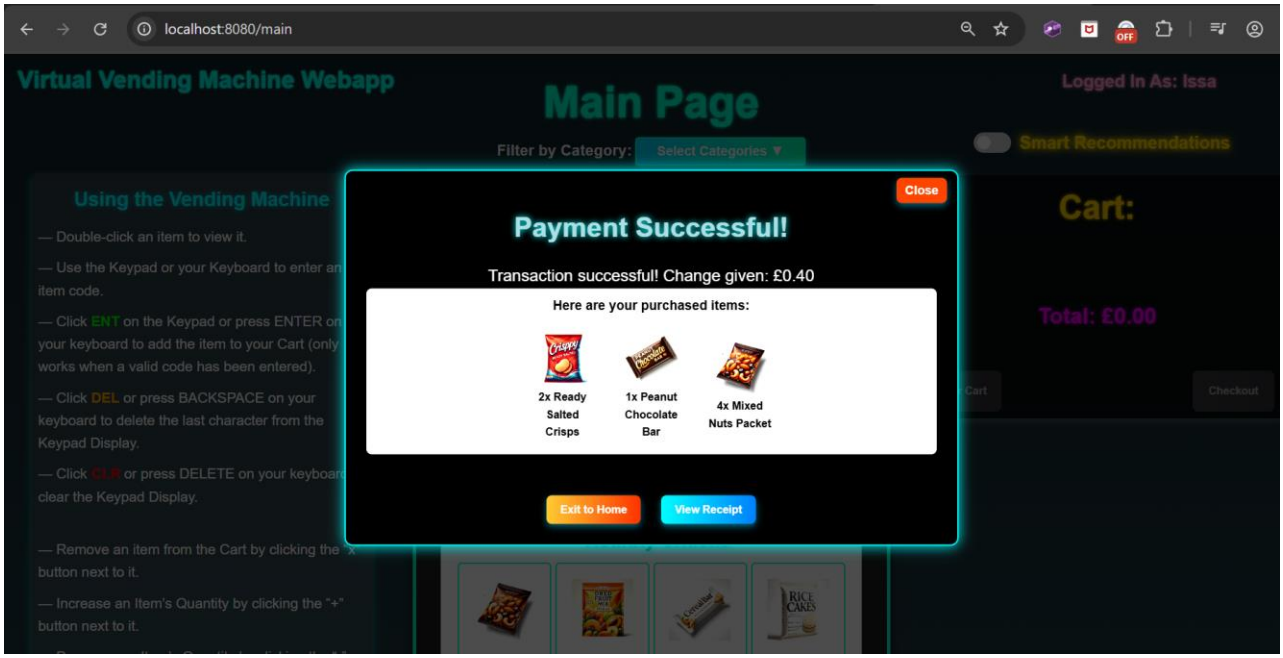
Screenshot showing the payment panel interface with available coin options, inserted amount tracking and coin movement animations.



(£1 Coin has been clicked and is visually moving towards the coin slot. The "Total Remaining" and "Inserted:" amounts have been updated. This all happens in real-time.)

Figure B6 – Successful Transaction Modal

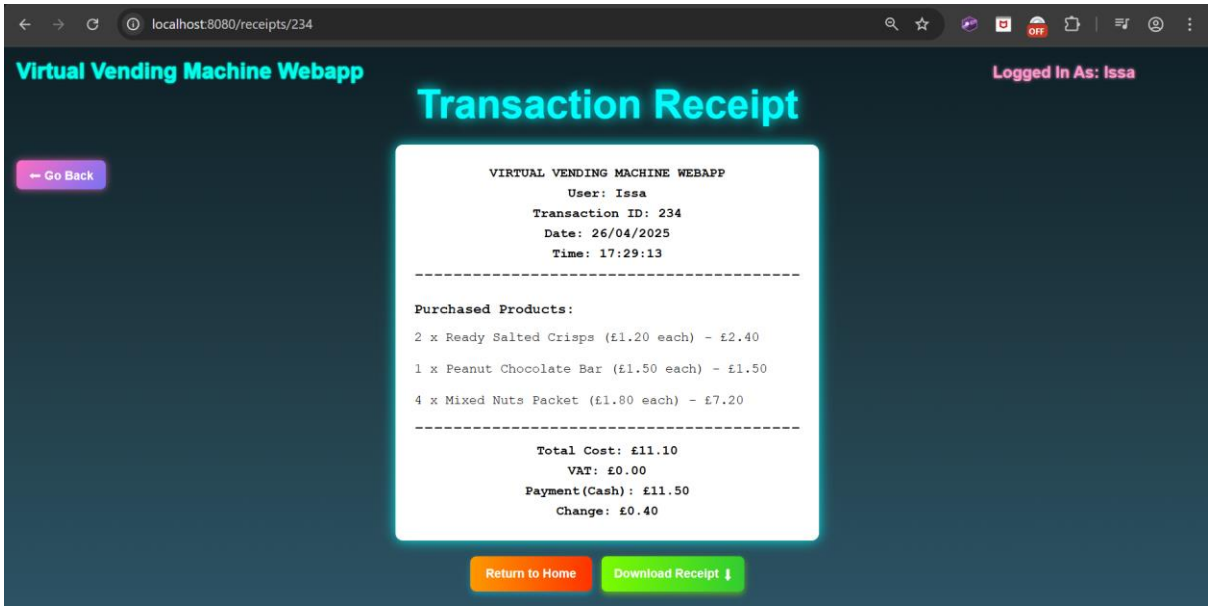
Screenshot showing the purchase summary modal after a completed payment, listing purchased items, payment totals and receipt options.



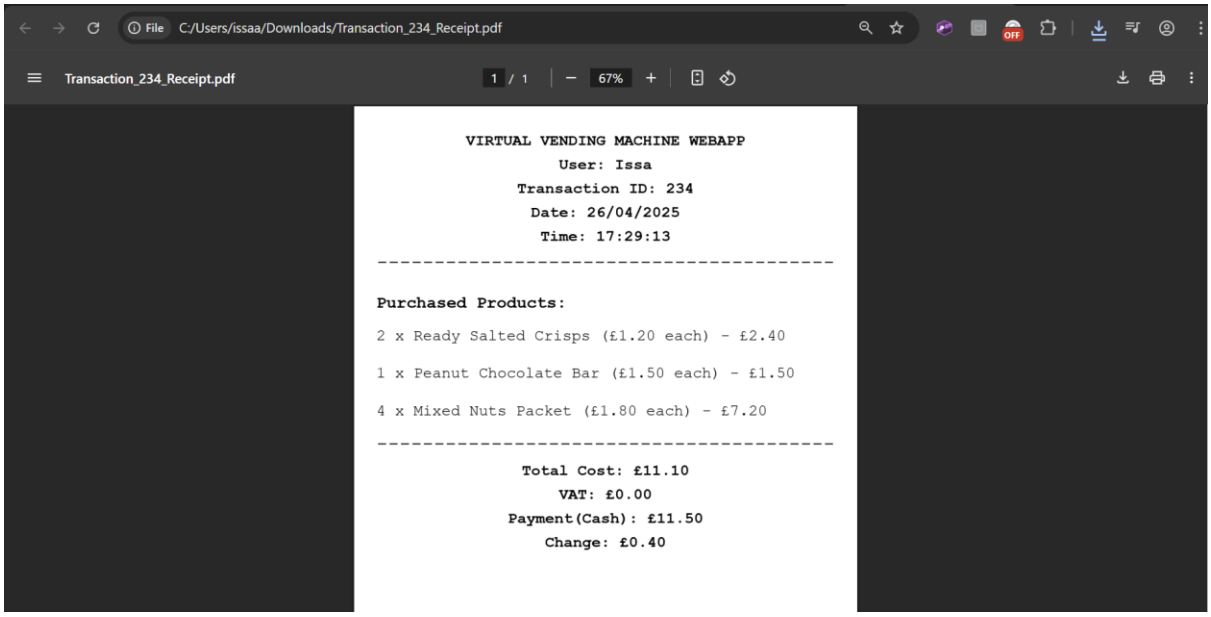
(User can choose to view the receipt, return to home page or close the modal and purchase again.)

Figure B7 – Receipt Page After Transaction

Screenshot showing the detailed digital receipt generated after a transaction, including all purchase and payment information.



Then the receipt after downloading it as a PDF:



(The format is identical to as it is on the page for consistency.)

Figure B8 – Frontend User Interface Flow Diagram

Explores the journey of a user interacting with the vending machine from product selection through to receipt viewing.

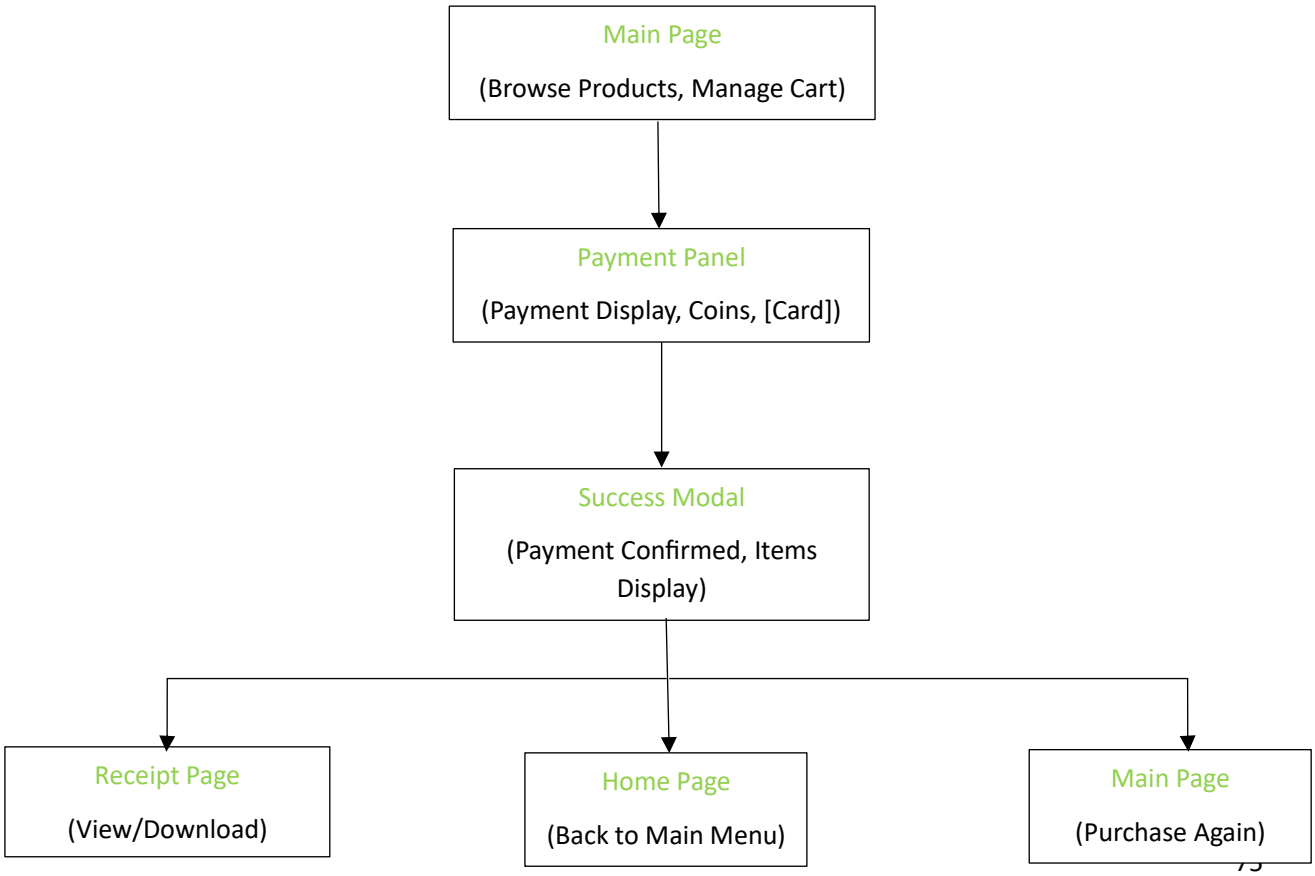
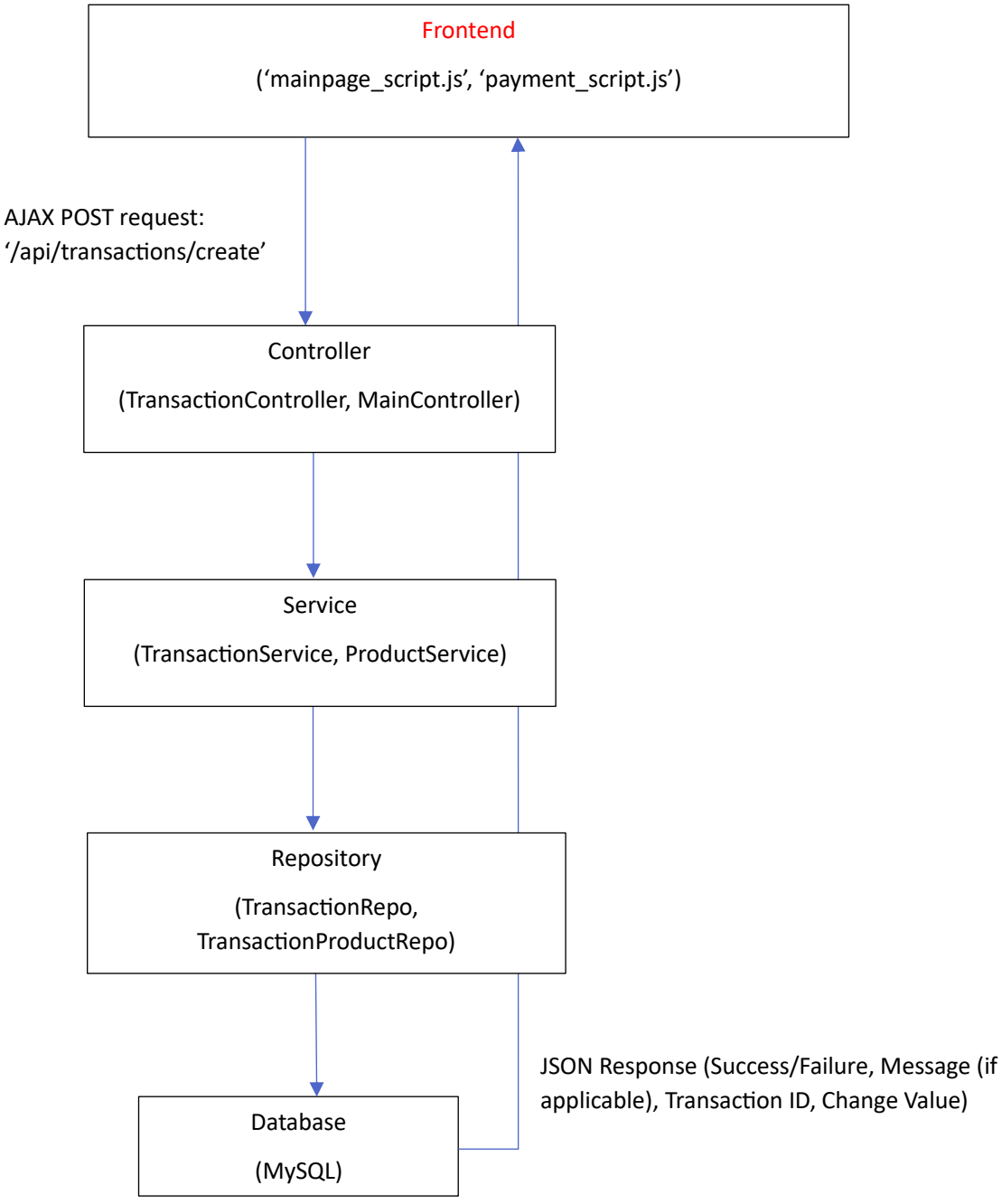


Figure B9 – AJAX Data Communication Flow Diagram

Shows how asynchronous AJAX requests manage dynamic transaction processing and receipt generation between the frontend and backend components.



Appendix C - Additional Features Implementation

Transactions History Page

The Transaction History Page provides logged-in users the ability to view a list of all their previous transactions, offering basic sorting and advanced filtering options. This feature showcases full-stack development, including secure backend data handling and dynamic frontend rendering using AJAX and JavaScript.

Backend Implementation

The backend logic for transaction history is primarily handled by the 'TransactionController', 'TransactionService', and 'TransactionRepo' classes:

- **Endpoints**:
  - '@GetMapping("/api/transactions/user")':
    Retrieves all transactions associated with the currently authenticated user. The controller extracts the username from the Spring Security session and fetches their transactions using the method: 'TransactionService.getTransactionsByUsername(username)'.
  - '@GetMapping("/api/transactions/filter")':
    Provides an endpoint for filtered transaction searches. Query parameters include:
    - 'transactionId'
    - 'minTotalCost'
    - 'maxTotalCost'
    - 'minPayment'
    - 'maxPayment'
    - 'minChange'
    - 'maxChange'
    - 'startDate'
    - 'endDate'
    - 'username'
  - The 'TransactionService.getFilteredTransactions(username, filters)' method dynamically builds a database query based on these filter values and fetches matching transactions.

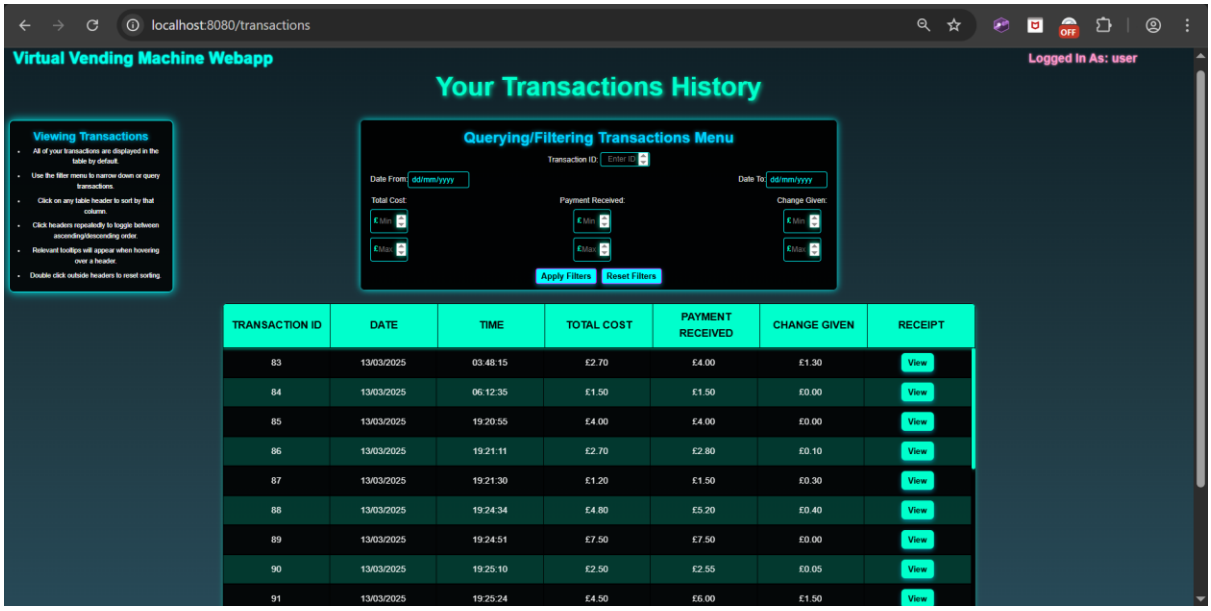
- **Repository Access**:
  - ‘TransactionRepo’ interacts with the database. Based on the parameters it receives from the controller method when called, it specifically queries to fetch only the required transactions based on the filters.
  - Securely ensures only the transactions belonging to the authenticated user are fetched, by enforcing username as a required parameter in the queries.
- **Security Enforcement**:
  - All access is restricted to authenticated users.
  - Guest users cannot access this page: the URL of this page is blocked for them and the button to access the page from the homepage is disabled.
  - Usernames are verified at every database query level to prevent cross-account data leaks.

This structure ensures user privacy while enabling dynamic, efficient fetching of transaction data and allowing easy and secure data filtering/querying.

Frontend Implementation

The frontend of the Transaction History page is implemented using a custom HTML template, JavaScript interactions and dynamic AJAX calls:

- **HTML Structure (‘transactionspage.html’)**:
  - Renders a clean table layout with headers:
    - “Transaction ID”
    - “Date”
    - “Time”
    - “Total Cost”
    - “Payment Received”
    - “Change Given”
    - “Receipt”
  - A "Filter Transactions" menu section allows users to input:
    - Transaction ID (search specific ID)
    - Payment Details [cost, amount paid and change] Range (Min/Max)
    - Date Range (Start and End)
  - Clear and Apply buttons to reset or submit filters.



*[**Figure C1 – Transactions History Page Screenshot**]*

- **JavaScript Logic (‘transactionspage_script.js’)**:
  - ‘fetchUserTransactions()’:
    - Sends an AJAX GET request to ‘/api/transactions/user’.
    - On success, calls ‘populateTransactionTable()’ to dynamically build the table rows with the data inserted in.
  - ‘populateTransactionTable(transactions)’:
    - Iterates through the received transaction array and injects rows into the HTML table.
  - ‘applyFilters()’:
    - Triggered by clicking the “Apply Filters” button in the menu.
    - Gathers filter inputs from the user and sends an AJAX GET request to ‘/api/transactions/filter’, attaching filter parameters.
    - Uses ‘populateTransactionTable(transactions)’, passing in an array of only filtered transactions, to update the transaction table based on filtered results.
  - ‘clearFilters()’:
    - Resets all filter input fields and reloads the default transaction list.
  - Validation is client-side to prevent empty or invalid filter fields from being sent unnecessarily.

[Figure C2 – Code Snippets: 'fetchUserTransactions()' and 'populateTransactionTable()']

```
//function to fetch transactions from the controller using AJAX and
update the table
async function fetchUserTransactions(username) {
    try {
        let response = await
fetch(`/api/transactions/user?username=${encodeURIComponent(username)}`
); //AJAX request

        if (!response.ok) {
            showAlert("Failed to fetch transactions."); //handle bad
response
        }

        let transactions = await response.json();
        populateTransactionTable(transactions);  //call function to
populate the table, passing in the transactions list
        currentTransactions = transactions; //store the transactions so
they can be sorted
    } catch (error) {
        console.error("Error fetching transactions:", error); //error
handling
    }
}

//function to populate the transactions table
function populateTransactionTable(transactions) {
    let tableBody = document.querySelector(".transactions-table
tbody");
    tableBody.innerHTML = ""; //clear all previous content in the table

    if (transactions.length === 0) { //check if there's no transactions
for the user
        tableBody.innerHTML = "<tr><td colspan='6'>No Transactions
Found.</td></tr>"; //add in a message
        return;
    }

    transactions.forEach(transaction => {  //iterate for each
transaction
        let row = document.createElement("tr");  //create a row

        //separating out date and time
        let transactionDate = new Date(transaction.transactionDate);
        let formattedDate = transactionDate.toLocaleDateString();
        let formattedTime = transactionDate.toLocaleTimeString();

        //add in all the transaction data
        row.innerHTML = `
            <td>${transaction.id}</td>
            <td>${formattedDate}</td>
            <td>${formattedTime}</td>
            <td>£${transaction.totalCost.toFixed(2)}</td>
            <td>£${transaction.paymentReceived.toFixed(2)}</td>
            <td>£${transaction.changeGiven.toFixed(2)}</td>
            <td><button
onclick="viewReceipt(${transaction.id})">View</button></td>
```

```
        `;

        tableBody.appendChild(row);  //append the row to the table
    });
}
```

(These functions fetch all user transactions from the backend controller via the API and dynamically populate the HTML transaction table without needing a page reload.)

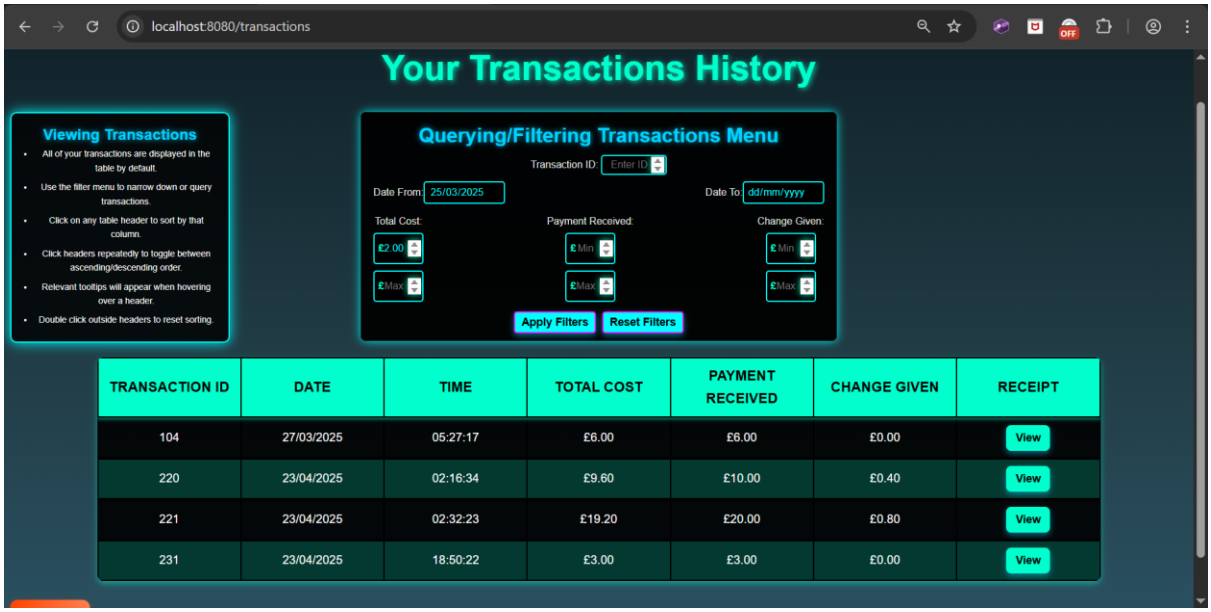
*[**Figure C3 – Code Snippet: 'applyFilters()' Function**]*

```
//function to apply filters by querying to the backend and displaying
the response using AJAX
function applyFilters() {
    if (!validateFilters()) return; //if any invalid data dont apply
filters

    //retrieve filter values from the HTML elements
    let transactionId = document.getElementById("transactionId").value;
    let startDate = document.getElementById("startDate").value;
    let endDate = document.getElementById("endDate").value;
    let minTotalCost = document.getElementById("minTotalCost").value;
    let maxTotalCost = document.getElementById("maxTotalCost").value;
    let minPayment = document.getElementById("minPayment").value;
    let maxPayment = document.getElementById("maxPayment").value;
    let minChange = document.getElementById("minChange").value;
    let maxChange = document.getElementById("maxChange").value;

    //create the URL parameters for the query
    let queryParams = new URLSearchParams({
        transactionId, startDate, endDate, minTotalCost, maxTotalCost,
        minPayment, maxPayment, minChange, maxChange, username
    });

    //send an AJAX fetch request to the controller method endpoint with
the query parameters sent over
    fetch(`api/transactions/filter?${queryParams.toString()}`)
        .then(response => response.json())  //parse the response
        .then(data => {

            currentTransactions = data;  //store the filtered
transactions in case they need to be sorted

            if (currentSortColumn) {   //check if there is currently a
sort active
                sortTableData(currentSortColumn, currentSortDirection);
//apply active sort to filtered data
            } else {
                populateTransactionTable(currentTransactions);
//otherwise display the filtered transactions in the table in default
order
            }

        })
        .catch(error => console.error("Error fetching transactions:",
```

```
error));  //handle errors
}
```

(This function gathers the user's filter inputs, sends a dynamic AJAX request to the backend and updates the transaction table with only the filtered results.)

*[**Figure C4 – Transactions History Table Filtering**]*



(This shows the same transactions table but filtered by “Date From” 25/03/2025 and minimum “Total Cost” as £2.00.)

- **Sorting Feature**:
  - Clicking on any table header triggers ‘sortTableByColumn(columnIndex, order)’.
  - Sorting order toggles between ascending and descending each time the same header is clicked.
  - Transactions are sorted by “Transaction ID” ascending by default. Double clicking anywhere off the table headers will reset the sorting order back to this.
  - Sorting is fully dynamic and entirely client-side only (no additional backend data or AJAX fetch is needed).

(Extra note: the JavaScript logic was specifically refined to allow both Filtering and Sorting to be applied at the same time)

*[**Figure C5 – Transactions History Table Sorting**]*

(Currently being sorted by “Total Cost” ascending. If we click again, as prompted by the tooltip the sort order will flip to descending.)

- **Receipt Viewing**:
  - Each transaction has a "View" button which redirects the user to the ‘/receipts/{transactionId}’ URL where a full detailed receipt is displayed on the Receipt Page. This re-uses all previous Receipt Viewing logic and allows users a way to access receipts for old transactions.

Card Payments (Stripe)

The Virtual Vending Machine Web Application includes an optional Card Payment method, allowing users to pay securely through Stripe as an alternative to the default coin insertion system. Stripe integration adds a professional payment flow to the system and demonstrates advanced full-stack development practices, including session handling and third-party API interaction. Since the system currently requires only simulated payments, it uses Stripe in test mode, meaning no real cards can be used and no actual money is paid.

Backend Implementation

The backend logic for card payments is handled through the 'PaymentController' and 'StripeService' classes:

- **Session Creation (Backend)**:
  - Endpoint: '@PostMapping("/api/payment/create-session")'
  - The 'createCheckoutSession()' method in 'PaymentController' accepts:
    - The amount to be paid (in pounds) and the username.
  - It calls the service method 'StripeService.createCheckoutSession(amount)', which:
    - Converts the amount into the smallest currency unit (pence) as required by Stripe (e.g. £3 becomes 300 pence).
    - Sets up a Stripe Checkout Session with the necessary payment configurations.
    - Provides two redirect URLs depending on the result:
      - Success: Appends a URL parameter: 'payment=success'
      - Failure/Cancellation: Appends a URL parameter 'payment=cancel'
    - Returns the created Checkout Session URL back to the frontend.
  - On the frontend, users are automatically redirected to the external secure Stripe Checkout page.
- **'StripeService'**:
  - Handles all direct interactions with the Stripe API.
  - Creates a 'SessionCreateParams' object containing payment details, success/cancel URLs and session mode (PAYMENT).
  - Uses Stripe's Java SDK to generate and return the Checkout session.

[Figure C9 – Stripe Session Creation in Backend ('StripeService.java')]

(The Java method that creates and configures the Stripe payment session.)

```
//method to create the Stripe card payment (checkout) session
public Session createCheckoutSession(double amount, String username)
throws StripeException {
    Stripe.apiKey = secretKey; //set the key

    //retrieve the user's email from the database
    String customerEmail = userRepo.findByUsername(username)
            .map(User::getEmail)
            .filter(this::isValidEmail) //check it's a valid email
address, using the helper method
            .orElse("guest@virtualvendingmachine.com");  //if invalid
or null, fallback to this default email (primarily for guests)

    //initialise the payment form
    SessionCreateParams params = SessionCreateParams.builder()
            .setMode(SessionCreateParams.Mode.PAYMENT)

.setSuccessUrl("http://localhost:8080/api/payment/payment/success")
//define success URL

.setCancelUrl("http://localhost:8080/api/payment/payment/cancel")
//define the cancel/failure URL
            .setCustomerEmail(customerEmail)  //pre-fill the user's
email in the payment form
            .addLineItem(
                    SessionCreateParams.LineItem.builder()
                            .setQuantity(1L)
                            .setPriceData(

SessionCreateParams.LineItem.PriceData.builder()
                                            .setCurrency("gbp") //set
currency as £
                                            .setUnitAmount((long)
(amount * 100)) //convert to pence (required for Stripe)
                                            .setProductData(

SessionCreateParams.LineItem.PriceData.ProductData.builder()

.setName("Virtual Vending Machine Purchase") //set the name for the
form
                                                        .build())
                                            .build())
                            .build())
            .build();

    return Session.create(params);  //return the created session
}
```

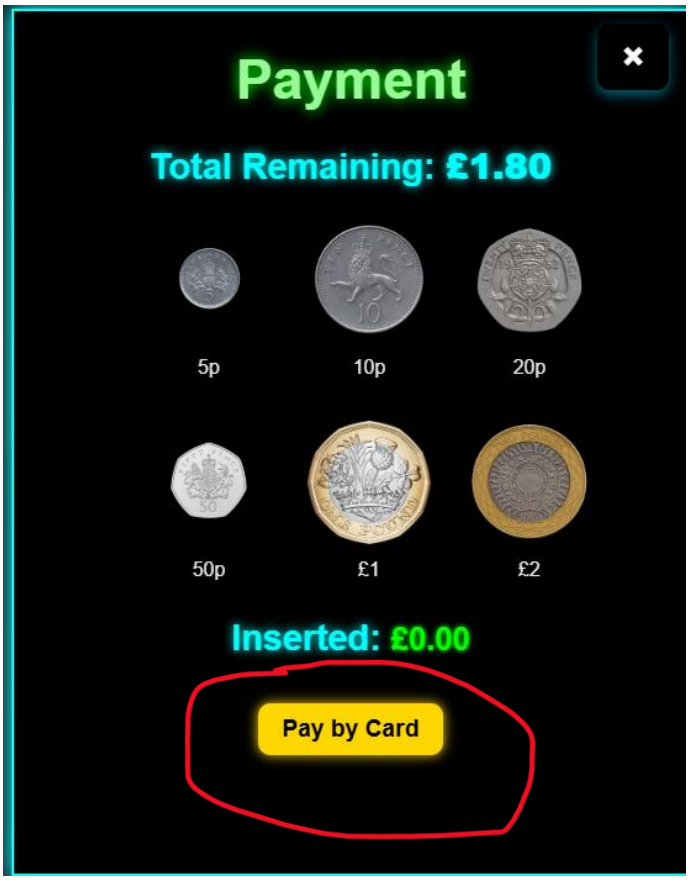
- **Security Considerations**:

- Only using Stripe test mode for now: no real card details or payments.
- All requests to start a payment are verified server-side, not entirely frontend initiated.
- Session and amount validation protects against tampering.

Frontend Implementation

The frontend integration for card payments was carefully implemented within the existing vending machine UI and payment flow:

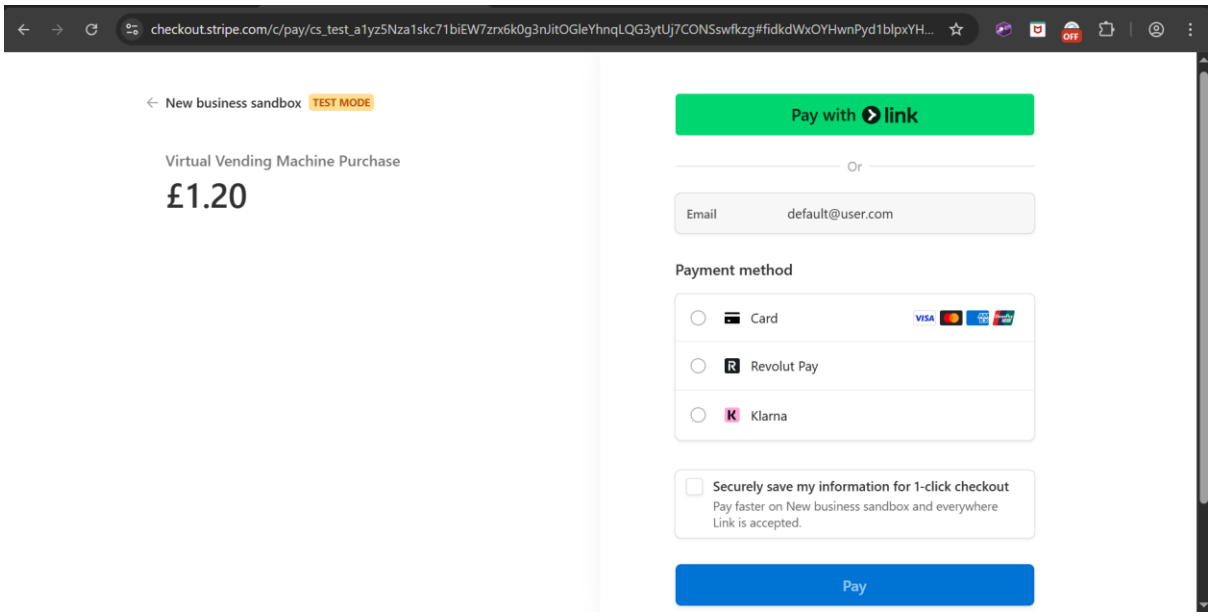
- **HTML Button ('mainpage.html')**:
  - A "Pay by Card" button exists within the payment modal screen, underneath the coins.
  - The button is only visible when starting a payment and disappears as soon as coin insertion begins (to stop combined coin and card payments).



*[**Figure C6 - Payment Modal showing Pay by Card Button**]*
Screenshot of the payment modal where both coin and card options are initially visible.

- **JavaScript Logic ('payment_script.js')**:
  - 'initiateCardPayment()' is called when the card button is clicked:
    - Gathers the remaining amount to pay and the username, both from elements on the page.
    - Saves the current cart contents and payment total into Local Storage in the browser (so they can remain beyond page reloads/redirects).
    - Sends an AJAX POST request to '/api/payment/create-session' to create a new Stripe Checkout Session.
    - Upon receiving the session URL, it redirects the user automatically to Stripe's secure payment page.

*[**Figure C7 – Stripe Checkout Page**]*
Screenshot of the Stripe secure payment page after redirection.



*[**Figure C8 – JavaScript Code Snippet for 'initiateCardPayment()'**]*

The function that sends the AJAX request to create a Stripe session and redirects user:

```
//function to process the Stripe card payment functionality
function initiateCardPayment() {
    const totalText = document.getElementById("remainingAmount");
//retrieve the amount to pay from the element
    const amount = parseFloat(totalText.textContent.replace(/[£,]/g,
""")); //format it as a pure number (float)

    //save the cart and required amount into storage, so they aren't
lost when the page reloads
    localStorage.setItem("cartItemsBackup", JSON.stringify(cartItems));
    localStorage.setItem("requiredAmountBackup", amount);

    const username = document.querySelector(".user-info
span")?.textContent?.trim(); //get the username from the page

    //fetch request to backend
    fetch("/api/payment/create-session", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({
            amount: amount,
            username: username
        })
        // use actual required amount
    })
        .then(res => res.json())
        .then(data => {
            if (data.url) {
                window.location.href = data.url;
            } else {
                showAlert("Payment initiation failed."); //error
handling
            }
        })
        .catch(err => {
            console.error("Stripe payment error:", err);
            showAlert("An error occurred starting card payment.");
//error handling
        });
}
```

- **Post-Payment Handling**:
  - After the payment on Stripe, the user is redirected back:
    - If successful (‘payment=success’ in URL):
      - The saved cart and total amount are restored from Local Storage which is then cleared when no longer needed.
      - The ‘processTransaction()’ method is called, reusing the standard purchase completion logic.
      - The URL is reset back to normal (remove the parameters) using ‘resetUrl()’.
    - If cancelled/failed (‘payment=cancel’ in URL):
      - The cart is restored from local storage, and a cancellation alert is shown to the user.
      - Again, the URL is reset to normal, removing the parameters.
  - Users never lose their cart until their payment is completed: Local Storage provides temporary state preservation.

Final Note: Stripe test mode is currently being used to **simulate** card payments. To complete the payment they must enter the default card number: 4242 4242 4242 4242, use any 3 digit number for the CVC, use any valid expiry date (any date after the current date), use any postcode and name and finally any valid email address (although this gets auto filled using the user’s registered email address from the database, or a default one if theirs is invalid or they are a guest user). Entering these correctly will simulate a valid payment through the Stripe page, redirect back to the main vending machine page and be treated as a successful transaction.

Analytics Page

The Analytics Page provides logged-in users with a detailed view of their purchasing behaviour and trends, using visuals for summary cards, dynamic charts and smart recommendation insights. This feature demonstrates full-stack data aggregation, AJAX-driven frontend updating and visualisation through HTML elements and Chart.js (an open-source JavaScript library for chart visualisation).

Backend Implementation

The backend logic for the Analytics Page is handled through the AnalyticsController, AnalyticsService, and separate DTO classes:

- **Endpoints**:
  - '@GetMapping("/analytics/summary")':
    - Returns an 'AnalyticsSummaryDTO', containing total number of purchases, total spent, most active day of the week and the number of unique items purchased.
  - '@GetMapping("/analytics/frequency")':
    - Returns an array of 'PurchaseFrequencyDTO' entries, showing the number of purchases made per month.
  - '@GetMapping("/analytics/spending")':
    - Returns an array of 'SpendingTrendDTO' entries, showing monthly total spending.
  - '@GetMapping("/analytics/item-breakdown")':
    - Returns an array of 'TopProductQuantityDTO' objects, showing total quantity bought for each different product.
  - '@GetMapping("/analytics/smart-insights")':
    - Returns a 'RecommendationDTO', containing top category, average spending and lists of recommended products based on collaborative filtering and price range logic (relates to Smart Recommendations feature).
- **AnalyticsService**:
  - Each controller method calls a corresponding method in 'AnalyticsService', which queries data from 'TransactionRepo' and 'TransactionProductRepo' methods, calculates required metrics and maps results into the appropriate DTOs (or arrays of DTOs) and returns them back.
- **DTOs**:
  - 'AnalyticsSummaryDTO' (for summary cards):

```
//new dto class for combining data into single objects to be used for the analytics page
public class AnalyticsSummaryDTO {
```

```
    //attributes
    private int totalPurchases;
    private double totalSpent;
    private String mostActiveDay;

    private int uniqueItemsPurchased;

    //constructors
    public AnalyticsSummaryDTO() {

    }

    public AnalyticsSummaryDTO(int totalPurchases, double
totalSpent, String mostActiveDay, int uniqueItemsPurchased)
{
        this.totalPurchases = totalPurchases;
        this.totalSpent = totalSpent;
        this.mostActiveDay = mostActiveDay;
        this.uniqueItemsPurchased = uniqueItemsPurchased;
    }

    //getters and setters
    public int getTotalPurchases() {
        return totalPurchases; }
    public void setTotalPurchases(int totalPurchases) {
        this.totalPurchases = totalPurchases; }

    public double getTotalSpent() {
        return totalSpent; }
    public void setTotalSpent(double totalSpent) {
        this.totalSpent = totalSpent; }

    public String getMostActiveDay() {
        return mostActiveDay; }
    public void setMostActiveDay(String mostActiveDay) {
        this.mostActiveDay = mostActiveDay; }

    public int getUniqueItemsPurchased() {
        return uniqueItemsPurchased;
```

```
    }

    public void setUniqueItemsPurchased(int
uniqueItemsPurchased) {
        this.uniqueItemsPurchased = uniqueItemsPurchased;
    }
}
```

*[**Figure C10 – Analytics Summary DTO Structure ('AnalyticsSummaryDTO.java')**]*
(Shows example backend model for analytics summary data.)

- 'PurchaseFrequencyDTO' (for bar chart data)
- 'SpendingTrendDTO' (for line graph data)
- 'TopProductQuantityDTO' (for pie chart breakdown)
- 'RecommendationDTO' (for smart insights and recommendation cards)

- **Security Enforcement**:
  - All analytics endpoints are protected by Spring Security and require user authentication.
  - Guest users cannot access the Analytics page (URL is restricted in 'SecurityConfig' and each analytics controller method validates against Guest users).

Frontend Implementation

The frontend Analytics Page ('analyticspage.html') is dynamically populated through AJAX requests and Chart.js visualisations. The page uses a modular loading system, where a master function 'loadAllAnalytics()' triggers individual AJAX fetches for each data category on page load:

- **Summary Cards**:
  - Cards display total purchases, total spent, most active day, and unique items.
  - AJAX fetch from '/analytics/summary' endpoint.
  - DOM updated via 'renderAnalyticsSummary(data)' JavaScript function.

```
//function to display the analytics data
function renderAnalyticsSummary(data) {
    document.getElementById("totalPurchases").innerText =
data.totalPurchases;
```

```
    document.getElementById("totalSpent").innerText =
`£${data.totalSpent.toFixed(2)}`;
    document.getElementById("activeDay").innerText =
data.mostActiveDay;
    document.getElementById("uniqueItems").innerText =
data.uniqueItemsPurchased;
}

//function to retrieve the summary data from the controller endpoint
using AJAX
function loadAnalyticsSummary() {
    fetch('/analytics/summary')
        .then(response => response.json())
        .then(data => renderAnalyticsSummary(data))
        .catch(err => console.error("Error loading analytics summary:",
err));
}
```

*[**Figure C11 – JavaScript Load and Render Functions for Analytics Summary**]*

(Shows the load function 'loadAnalyticsSummary()' and the render function 'renderAnalyticsSummary(data)' for the analytics summary cards visual.)

- **Purchase Frequency Chart**:
  - Bar chart displaying the number of purchases per month.
  - Data loaded via '/analytics/frequency' endpoint.
  - Rendered with Chart.js using custom animations and consistent neon-themed styling.

```
//function to create and display the purchase frequency chart (bar
chart)
function renderPurchaseFrequencyChart(data) {
    const canvas = document.getElementById('purchaseFrequencyChart');
    const ctx = canvas.getContext('2d');

    const msg = document.getElementById('noFrequencyDataMsg');
    const chart = document.getElementById('purchaseFrequencyChart');

    if (handleNoChartData(data, msg, chart)) return; //call to check
and handle empty chart data and if returns true, stop rendering (no
data)
```

```
    const labels = data.map(entry =>
        new Date(entry.month).toLocaleString('default', { year:
'numeric', month: 'short' })
    );
    const counts = data.map(entry => entry.count);

    const container = canvas.parentElement;
    if (data.length > 12) {
        container.style.overflowX = 'auto';
        container.style.width = '100%';
        canvas.width = data.length * 80;
    }

    new Chart(ctx, {
        type: 'bar',
        data: {
            labels: labels,
            datasets: [{
                label: 'Purchases per Month',
                data: counts,
                backgroundColor: 'rgba(0, 255, 255, 0.3)',
                borderColor: 'cyan',
                borderWidth: 2,
                hoverBackgroundColor: 'rgba(0, 255, 255, 0.7)',
                hoverBorderColor: '#0ff',
                barThickness: 40
            }]
        },
        options: {
            responsive: true,
            maintainAspectRatio: false,
            scales: {
                x: {
                    title: {
                        display: true,
                        text: 'Month',
                        color: 'cyan',
                        font: {
                            size: 16,
```

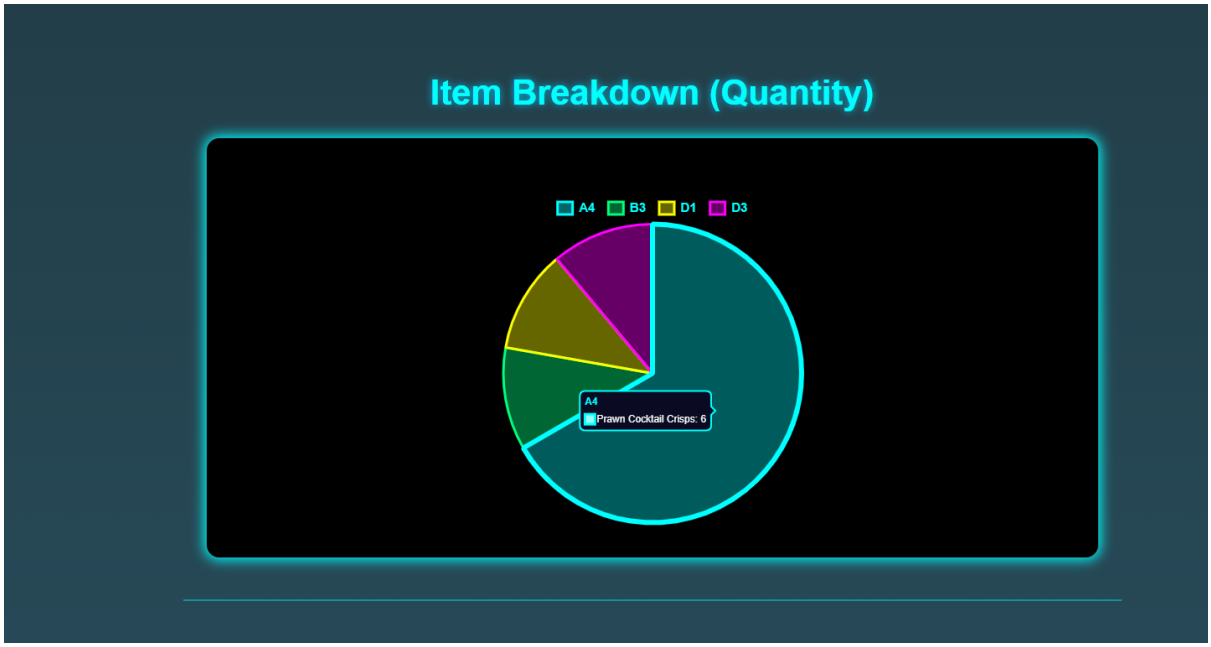
```
                        weight: 'bold'
                    }
                },
                ticks: {
                    color: '#0ff',
                    font: {
                        size: 14
                    }
                },
                grid: {
                    display: true,
                    color: 'rgba(0, 255, 255, 0.1)',
                    lineWidth: 1
                }
            },
            y: {
                beginAtZero: true,
                title: {
                    display: true,
                    text: 'Number of Purchases',
                    color: 'cyan',
                    font: {
                        size: 16,
                        weight: 'bold'
                    }
                },
                ticks: {
                    color: '#0ff',
                    font: {
                        size: 14
                    }
                },
                grid: {
                    display: true,
                    color: 'rgba(0, 255, 255, 0.1)',
                    lineWidth: 1
                }
            }
        },
```

```
            plugins: {
                legend: {
                    labels: {
                        color: '#00ffff',
                        font: {
                            size: 14
                        }
                    }
                },
                tooltip: {
                    backgroundColor: '#000',
                    borderColor: '#0ff',
                    borderWidth: 1,
                    titleColor: '#0ff',
                    bodyColor: '#fff',
                    titleFont: {
                        size: 16,
                        weight: 'bold'
                    },
                    bodyFont: {
                        size: 14
                    },
                    displayColors: false,
                    intersect: true,
                    mode: 'index',
                    caretSize: 6
                }
            },
            animation: {
                duration: 1000,
                easing: 'easeOutCubic'
            }
        }
    });
}
```

*[**Figure C12 – JavaScript Chart.js Rendering for Purchase Frequency**]*
(Code snippet showing chart config and rendering.)

- **Spending Trend Chart**:
  - Line graph showing total spending trend over months.
  - Data loaded via '/analytics/spending' endpoint.
  - Smooth curve animations and gradient fills using Chart.js.

- **Item Breakdown Pie Chart**:
  - Pie chart displaying the quantity of each purchased item.
  - Data fetched from '/analytics/item-breakdown' endpoint.
  - Each product has a unique colour for improved readability.

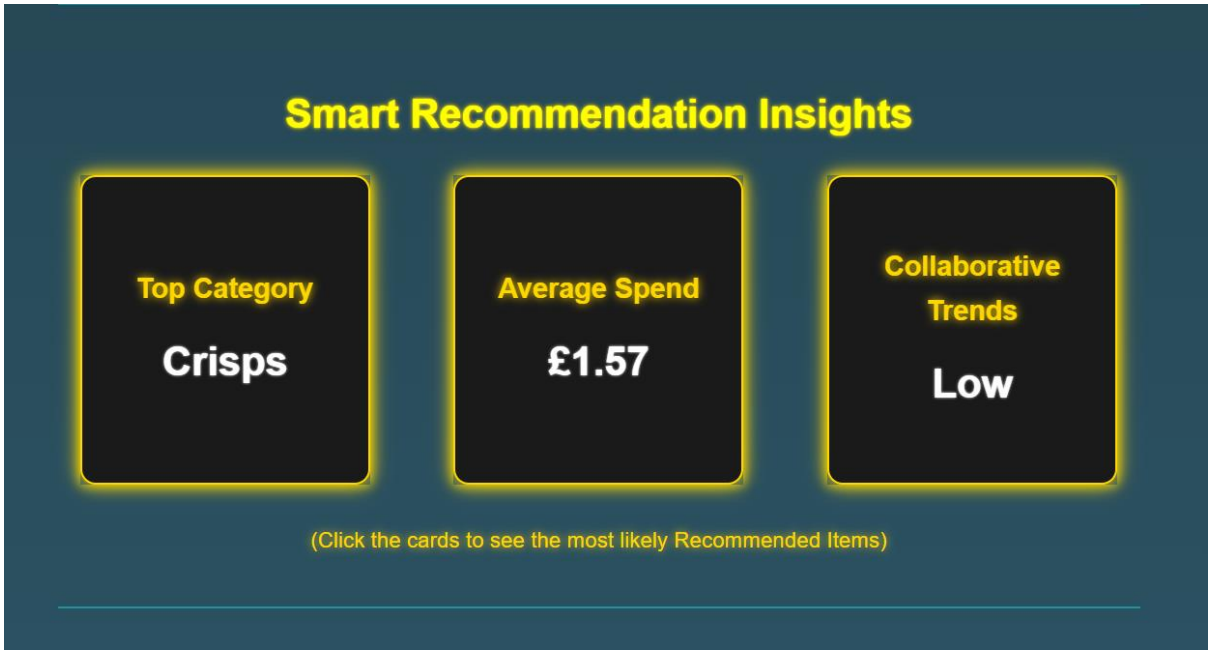


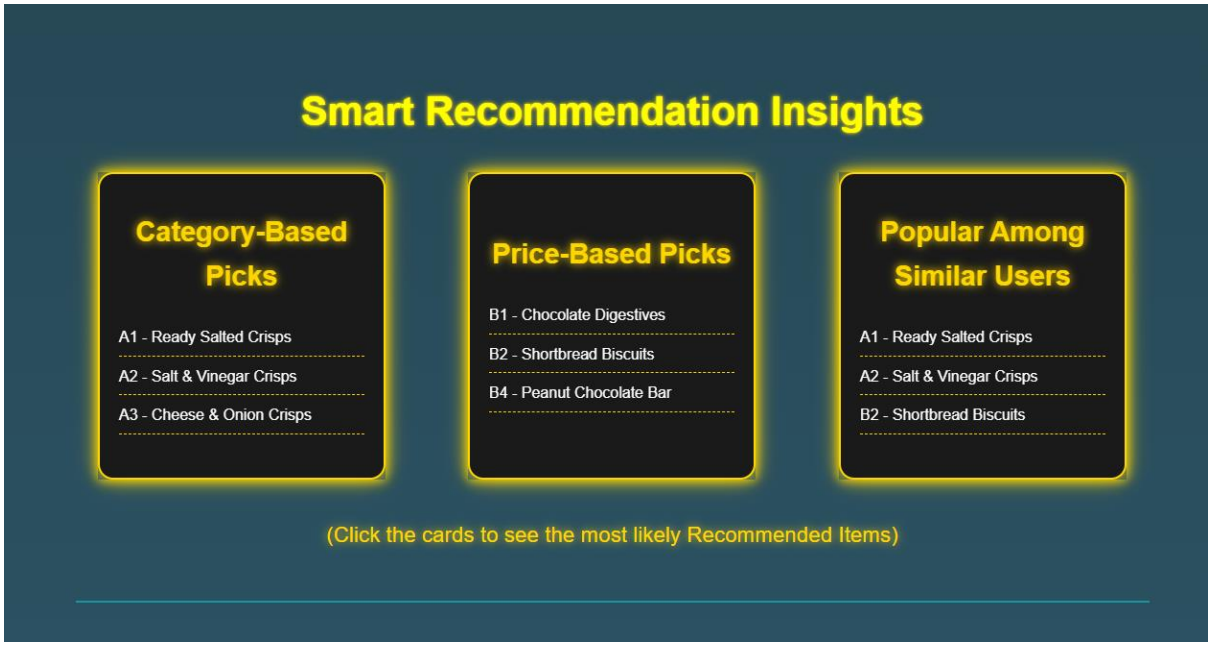
*[**Figure C13 - Pie Chart Showing Item Breakdown**]*
(Shows purchase quantity distribution across different items.)

- **Smart Recommendation Insights**:
  - Section displaying:
    - Top purchased category
    - Average spending value

- Collaborative purchase trends
- Potential recommended items based on all the above factors (Calculated and used in the Smart Recommendations algorithm).

- Dynamic AJAX call to '/analytics/smart-insights'.
- Flip-card styled UI elements (HTML/CSS) to show front and back content interactively.
- Relates directly to the Smart Recommendations feature.





*[**Figure C14 - Smart Recommendation Insights Cards**]*

(Flip cards showing top category, average spend and collaborative suggestions. Relates to the Smart Recommendations feature.)

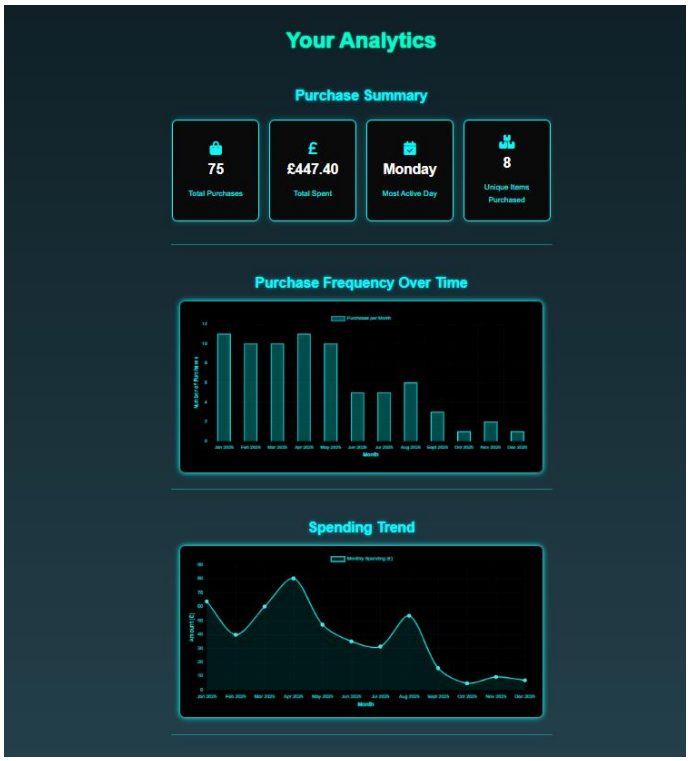
- **JavaScript:**
  - Each visual on the page has a load and a render function.
  - Load functions make AJAX requests to the backend and retrieve the required data. They then call the correct render function.
  - Render functions create the visual (cards/graph/charts) and populate it with the fetched data.
  - An overall load function 'loadAllAnalytics()', which is run when the page loads, calls all individual load functions.
  - A helper function 'handleNoChartData()' is used for all visuals to check if data was loaded properly. If not, it will add a relevant message in and prevent the page from crashing.

Visual Style and User Experience

- All charts and summaries adopt a futuristic neon aesthetic to remain consistent with the overall system theme.
- Loading states and errors are handled gracefully to ensure the page remains user-friendly even if data is missing.
- Guest users see a restricted message if they attempt to access the page.

*[**Figure C15 - Analytics Page Full Screenshot**]*

(A zoomed out view of the other analytics visuals: summary, purchase frequency and spending trend.)



Smart Recommendations Feature

The Smart Recommendations feature was developed to enhance the vending machine user experience by providing dynamic, intelligent and real-time product recommendations based on the user's previous purchases. It introduces basic machine learning concepts into the system by applying a hybrid multi-factor scoring system, adaptive weighting and simple collaborative filtering mechanisms.

Algorithm Overview

The recommendations are generated by an algorithm which is implemented in the backend in its own dedicated 'RecommendationService' class and is called by the controller method with the '/main/api/recommendations' endpoint.

The recommendation process is based on a scoring system that evaluates products using three weighted factors:

1. **Category-Based Similarity**:
    - Products belonging to the user's most frequently purchased category are given higher scores.
    - This is one of the best ways to pick out similar products that the users might like, based on their previous purchases.

2. **Price Range Similarity**:
    - Products are scored based on their similarity range to the user's average spending value.
    - This ensures that recommendations align with the user's typical affordability and preferences.
3. **Collaborative Filtering (Basic Version)**:
    - Products that have been purchased often by other users are given a popularity boost.
    - While full collaborative filtering typically requires user-product matrices and similarity calculations, this simple version captures some of that behaviour through advanced database queries.

Each factor contributes to a product's total recommendation score. Products are ranked by total score, and the top-ranked items are stored in a list and sent to the frontend to be highlighted on the vending machine frontend.

Technical Process

1. Identify Already Purchased Products:
    - Using a query method in 'TransactionRepo', a list of products already purchased by the user is retrieved to avoid recommending the same products.

2. Calculate Category Preferences:
    - A scoring map is built where each product category is scored based on the number of items the user has previously bought in that category.
    - More purchases in a category = a higher category preference score.

3. Score All Products:
    - Each available product is given an initial score based on multiple weighted factors:

Factor	Description	Contribution
Category-Based Similarity	Products matching the user's strongest categories receive a logarithmic scaled boost to prevent strong bias from dominant categories.	Moderate Weight (but scalable)
Price Range Similarity	Products priced near the user's average transaction amount get a proportional boost, based on how close the price gap is. Ensures affordability is respected.	Weak Weight (so boosted by a multiplier)
Collaborative Filtering Boost	Products frequently bought by other users are given an adaptive bonus, which scales depending on the user's purchase history.	Adaptive Weight (decided by a helper method)

Each product's final score is the sum of these individual factor scores.

4. Collaborative Filtering (Adaptive):
    - The collaborative filtering score boost value is decided by a helper method 'setAdaptiveCollabBoost'. This uses the user's transaction history depth to decide how strongly the collaborative filtering factor should affect a product's overall recommendation score:
        - Few previous transactions = larger boost to rely more on crowd popularity.
        - Loads of previous transactions = smaller boost, relying more on personalised factors.

5. Ranking and Selection:
    - Products are ranked by their total computed scores.
    - The top 5 highest-scoring products are selected.
    - Each recommended product is tagged with a reason (category, price, or collaboration) based on a weighted calculation of which scoring factor contributed most to its final score.

This structured yet flexible scoring system ensures recommendations are meaningful, contextually relevant and user-personalised without requiring complex machine learning libraries.

*[**Figure C15 – Recommendation Algorithm Code Snippet ('RecommendationService.java')**]*

(Code showing the main algorithm used to generate product recommendations.)

```
//method to smartly generate a list of recommended products and return
it to the controller
//uses an algorithm containing ML concepts for advanced and
personalised recommendations
public List<RecommendationDTO> generateRecommendations(String username)
{

    //retrieve a list of top products already purchased by the user,
using the repo method
    List<String> alreadyPurchased =
transactionRepo.findTopProductsByUser(username);

    //1.Score all categories based on user preference
    Map<String, Integer> categoryScore = new HashMap<>(); //initialise
a hashmap for scoring categories
    List<Transaction> transactions =
transactionRepo.findByUser(username); //retrieve all the user's
transactions

    //iterate for every user transaction
    for (Transaction t : transactions) {
```

```
        for (TransactionProduct tp : t.getTransactionProducts()) {
//iterate for every product in the transaction
            //attempt to find the product, if found total up the
quantity of each category
            productRepo.findById(tp.getProductId()).ifPresent(p ->
                    categoryScore.merge(p.getCategory(),
tp.getQuantity(), Integer::sum));
        }
    }

    //2.Score all products by match to user's preferences
    List<Product> allProducts = productRepo.findAll(); //get a list of
all products
    Map<String, Double> productScores = new HashMap<>(); //create new
hashmap for scoring products
    Map<String, Double> categoryContrib = new HashMap<>(); //hashmap to
track category-based contribution
    Map<String, Double> priceContrib = new HashMap<>(); //hashmap to
track price-based contribution
    Map<String, Double> collabContrib = new HashMap<>(); //hashmap to
track collaborative contribution

    double avgSpend = avgSpent(transactions); //get the user's average
spend
    boolean isNewUser = transactions.isEmpty(); //check if user is new
(has no transactions)

    int totalCategoryQuantity =
categoryScore.values().stream().mapToInt(Integer::intValue).sum();
//get total number of categories
    for (Product p : allProducts) { //iterate for all products
        if (alreadyPurchased.contains(p.getId())) continue; //if user
has already purchased the product, skip it

        double score = 0; //initialise a total score variable

        //initialise partial scores for category and price
        double catBoost = 0;
        double priceBoost = 0;

        //category-based scoring
        int catScore = categoryScore.getOrDefault(p.getCategory(), 0);
//get existing value or default to 0
        if (catScore > 0) {
            double raw = (double) catScore / totalCategoryQuantity;
//get a raw category score between 0-1
            catBoost = Math.log1p(catScore) * raw; //scale category
boost down using log function
            score += catBoost; //increment total score with category
boost
        }


        //price range similarity
        if (!isNewUser) { //only consider price match for experienced
users
            double priceDiff = Math.abs(avgSpend - p.getPrice());
//calculate difference between product price and user average spend
```

```
                priceBoost = 1.0 / (1 + priceDiff) * 1.4; //set the
price score boost based on closeness to average price (now with a 1.4
multiplier to give it more chance)
                score += priceBoost;  //increment the total score with
the price score boost
        }

        //store individual contributions for comparison later
        if (catBoost > 0) categoryContrib.put(p.getId(), catBoost);
        if (priceBoost > 0) priceContrib.put(p.getId(), priceBoost);

        productScores.put(p.getId(), score); //store the productID and
its corresponding score in the hashmap
    }

    //3.Collaborative filtering scoring boosts: higher recommendation
for items bought by similar users

    //retrieve a list of top products bought by other users (exclude
the current user and products they have already bought), using the repo
method
    List<String> collabItems =
transactionRepo.findPopularProductsNotInUser(username,
alreadyPurchased);

    for (String id : collabItems) { //iterate for each collaborative
suggested item
        double boost = setAdaptiveCollabBoost(transactions); //define
the collaborative boost value from the helper method
        double updatedScore = productScores.getOrDefault(id, 0.0) +
boost; //get the existing score (default to 0 if none) and add on the
collaborative score boost
        productScores.put(id, updatedScore); //boost their scores
        collabContrib.put(id, boost); //track the collab contribution
    }

    //4.Create a final list of the top scoring items and return it with
reasons based on contribution levels

    return productScores.entrySet().stream()
            .sorted((a, b) -> Double.compare(b.getValue(),
a.getValue())) //sort by score (descending order)
            .limit(5) //select the top 5
            .map(entry -> {
                String id = entry.getKey(); //get the id
                double total = entry.getValue(); //get the total score

                //fetch the contributions (default to 0 if none)
                double cat = categoryContrib.getOrDefault(id, 0.0);
                double price = priceContrib.getOrDefault(id, 0.0);
                double collab = collabContrib.getOrDefault(id, 0.0);

                String reason = "hybrid"; //fallback/default reason

                //calculate contribution percentages
                if (total > 0) {
                    double catPct = cat / total;
                    double pricePct = price / total;
```

```
                    double collabPct = collab / total;

                    if (catPct >= pricePct && catPct >= collabPct) {
                        reason = "category"; //set the reason if
category percent is dominant
                    } else if (pricePct >= catPct && pricePct >=
collabPct) {
                        reason = "price";  //set the reason if price
percent is dominant
                    } else {
                        reason = "collab";  //set the reason as collab
in all other cases
                    }
                }

                System.out.printf("Product %s => score %.2f (cat %.2f,
price %.2f, collab %.2f) -> reason: %s%n",
                        id, total, cat, price, collab, reason); //log
all scoring and reason data for testing

                return new RecommendationDTO(id, reason); //map to
RecommendationDTO including reason
            })
            .toList(); //return as a list
}
```

Machine Learning Concepts Integrated

Although a full ML pipeline (training, validation etc.) is not implemented due to time constraints, the algorithm integrates several important Machine Learning concepts:

- **User Feature Extraction**: (average spending, preferred categories)
- **Collaborative Filtering Principles**: boosting products popular with similar users.
- **Multi-Factor Weighted Scoring**: different factors involved, each with a different weight on the overall score.
- **Adaptive Boosting**: dynamic tuning of collaborative influence on the overall score, based on user transaction history.
- **Logarithmic Scaling**: reduces impact of outliers (heavy purchasers) in category weights.

While lightweight and domain-specific, this feature effectively implements a simplified personalised recommendation system.

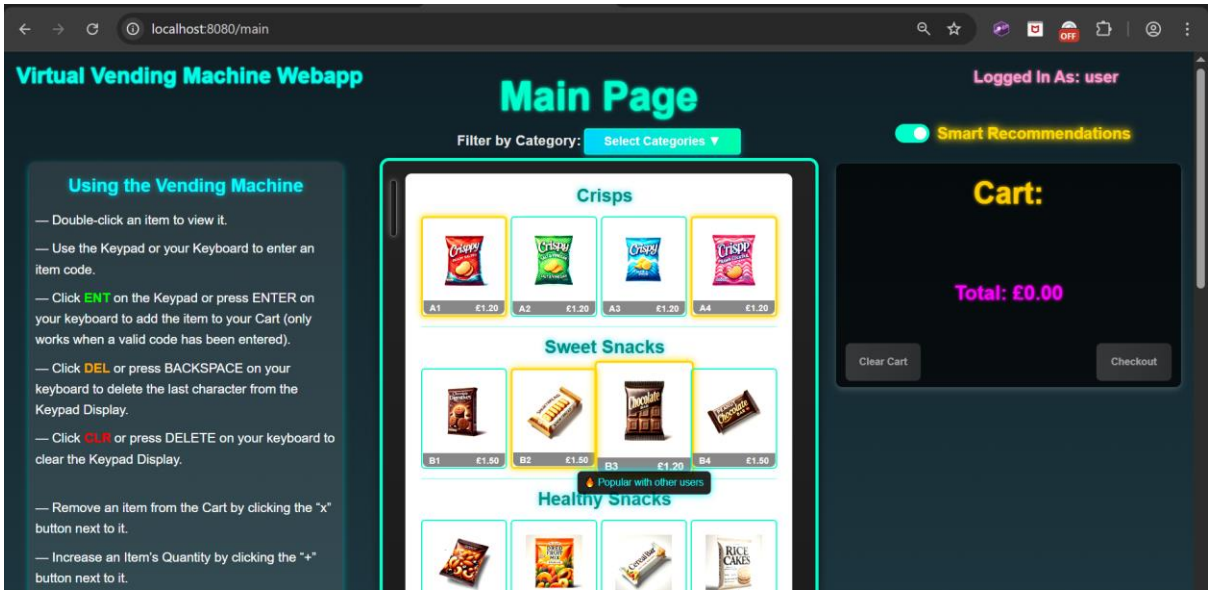
Visual Frontend Implementation

- The user can toggle a “Smart Recommendations” switch on the frontend to turn recommendations on/off.
- When turned on, a list of recommended products is fetched via AJAX, the frontend highlights the products using special visual indicators (CSS neon glow effects) on the vending machine grid.

- Reasons ("Matches your spending habits", "Based on your favourite categories" or "Popular with other users") are also attached to each recommended product as a tooltip, based on the dominant factor score.

*[**Figure C16 – Recommended Products Display Screenshot**]*

(Screenshot showing products visually highlighted as Smart Recommendations inside the vending machine.)



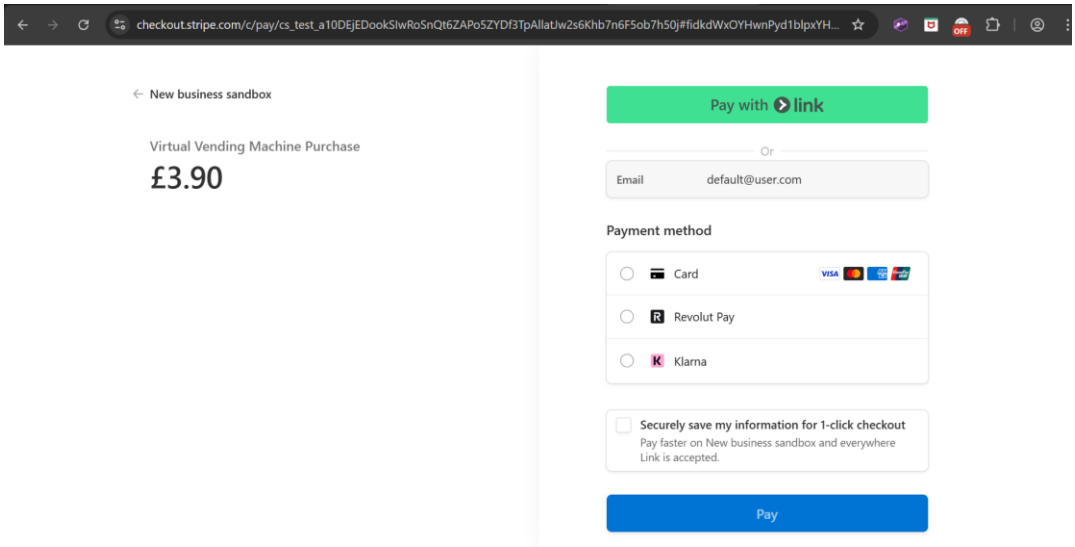
Technical Notes

- The recommendation toggle is disabled for guest users.
- Recommendations refresh dynamically every time the page is refreshed, a transaction is completed, or the toggle is activated.
- The frontend ('mainpage_script.js') ensures performance by only applying visual updates to specific recommended products without re-rendering the whole page.

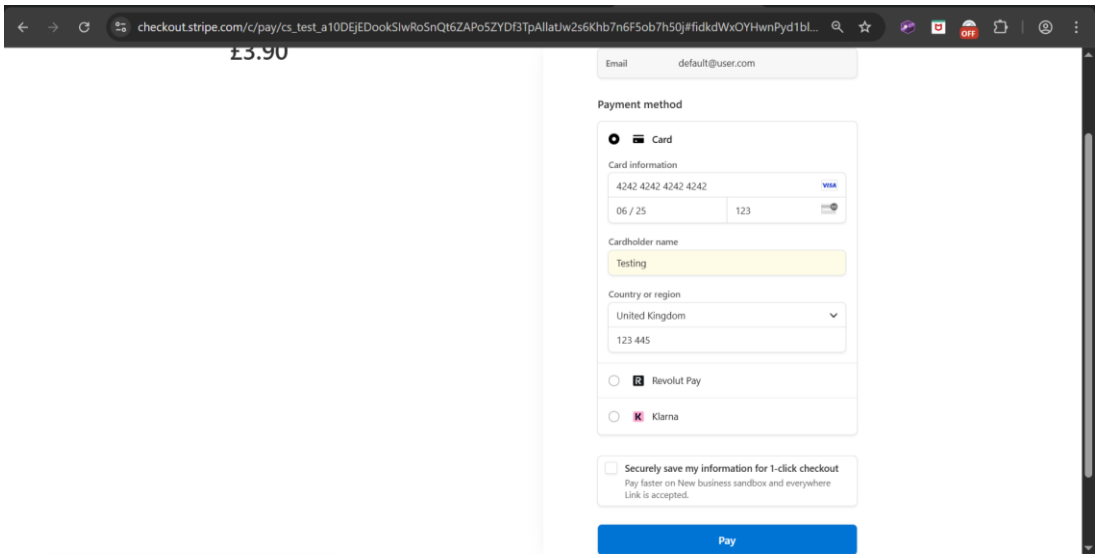
Appendix D – Additional Features Testing

Testing of all other additional features to ensure they work properly.

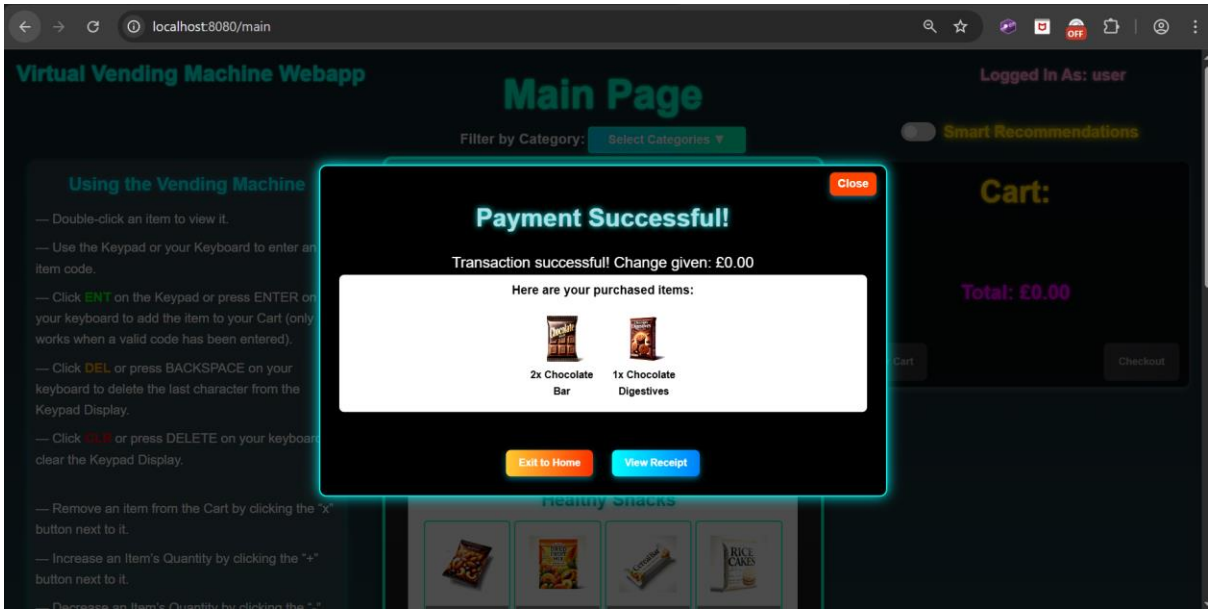
Stripe Card Payment Flow Testing



(After clicking the “Pay by Card” button we get successfully redirected to the Stripe Checkout page, with the user email and amount to pay already filled in.)



(Filled out the form using the default Stripe Test Mode card number, a valid expiry date and a made-up CVC, name and postcode.)



(After clicking “Pay” we are redirected to the main vending machine page, and the success modal appears, indicating a successful purchase.)

(Custom system logs just confirm that the transaction was properly created and the stock for the two items was correctly deducted.)

Transaction History Testing

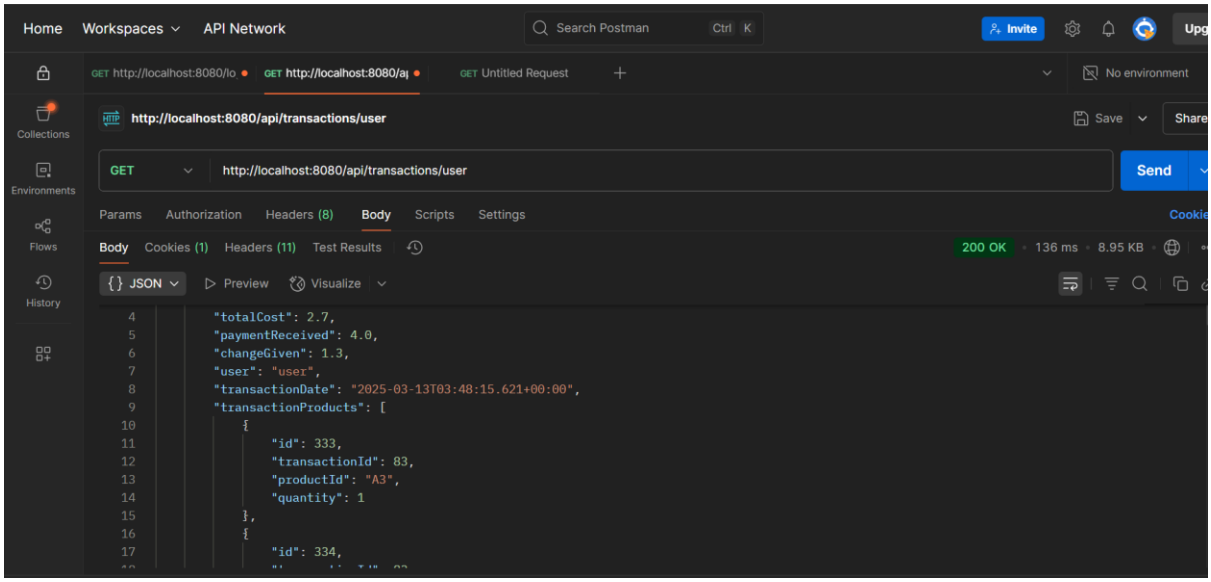
The Transaction History page was tested to ensure users could successfully view and interact with their previous transactions. Testing focused on correct data retrieval, table rendering, sorting correctly and filtering accuracy.

Page Load Testing

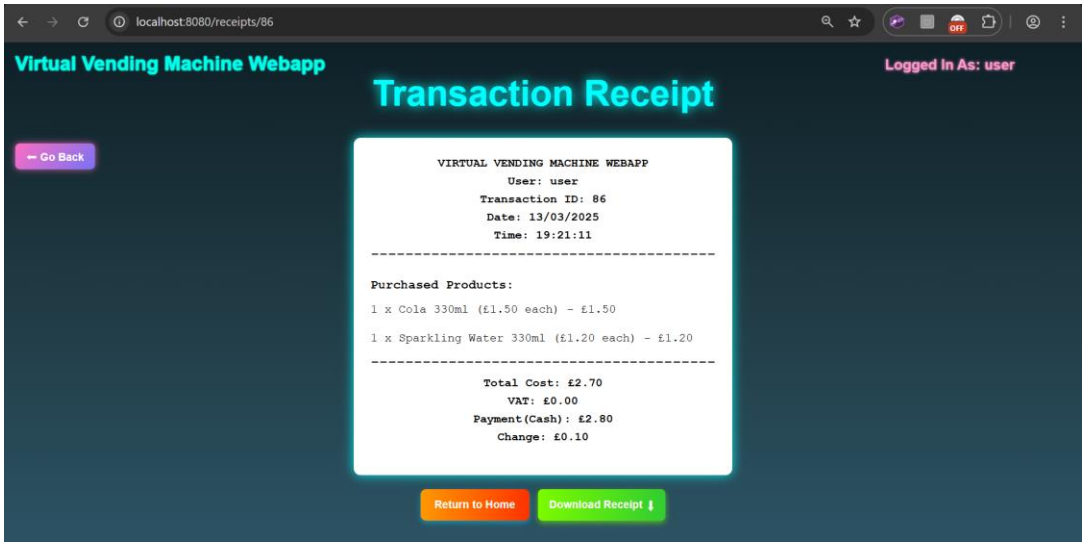
- Upon accessing the page as a logged-in user, a table populated with all of the user’s past transactions correctly appeared.
- Each transaction displayed accurate details: Transaction ID, Date, Time, Total Cost, Payment Received and Change Given.

- The "View Receipt" button next to each transaction correctly navigated to the associated receipt page.

(The page table shows all transactions for the user in the order of their Transaction ID. The table is scrollable to fit any number of transactions within a fixed space.)



(This screenshot shows a snippet of the JSON data returned (transactions) when for the API endpoint: ‘api/transactions/user’ which is the one we redirect to upon loading this page.)



(This screenshot shows successful navigation to the receipt page for transaction 86 by clicking the “View” button in the table.)

Sorting Testing

Sorting was tested by clicking on the table headers (e.g., Transaction ID, Total Cost). Each click alternated the sorting order between ascending and descending, and the visual sorting direction indicator correctly updated.

TRANSACTION ID	DATE	TIME	TOTAL COST	PAYMENT RECEIVED ▲	CHANGE GIVEN	RECEIPT
223	23/04/2025	18:28:07	£0.00	£0.00	£0.00	View
224	23/04/2025	18:29:04	£0.00	£0.00	£0.00	View
225	23/04/2025	18:29:16	£0.00	£0.00	£0.00	View
228	23/04/2025	18:43:06	£0.00	£0.00	£0.00	View
229	23/04/2025	18:43:13	£0.00	£0.00	£0.00	View
230	23/04/2025	18:49:15	£0.00	£0.00	£0.00	View
94	13/03/2025	19:33:23	£1.20	£1.20	£0.00	View
103	20/03/2025	17:38:52	£1.20	£1.20	£0.00	View
219	23/04/2025	02:16:06	£1.20	£1.20	£0.00	View
226	23/04/2025	18:31:20	£1.20	£1.20	£0.00	View
227	23/04/2025	18:42:36	£1.20	£1.20	£0.00	View

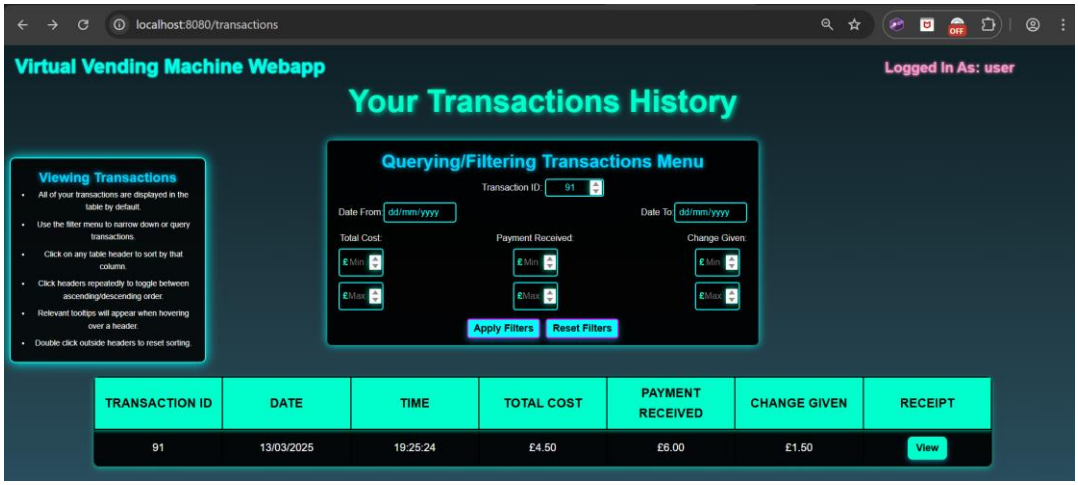
(Sorts by Payment Received ascending when we first click it.)

TRANSACTION ID	DATE	TIME	TOTAL COST	PAYMENT RECEIVED ▼	CHANGE GIVEN	RECEIPT
93	13/03/2025	19:26:00	£38.90	£39.35	£0.45	View
221	23/04/2025	02:32:23	£19.20	£20.00	£0.80	View
92	13/03/2025	19:25:40	£12.60	£13.00	£0.40	View
220	23/04/2025	02:16:34	£9.60	£10.00	£0.40	View
89	13/03/2025	19:24:51	£7.50	£7.50	£0.00	View
91	13/03/2025	19:25:24	£4.50	£6.00	£1.50	View
100	16/03/2025	02:21:08	£5.70	£6.00	£0.30	View
104	27/03/2025	05:27:17	£6.00	£6.00	£0.00	View
88	13/03/2025	19:24:34	£4.80	£5.20	£0.40	View
83	13/03/2025	03:48:15	£2.70	£4.00	£1.30	View
85	13/03/2025	19:20:55	£4.00	£4.00	£0.00	View

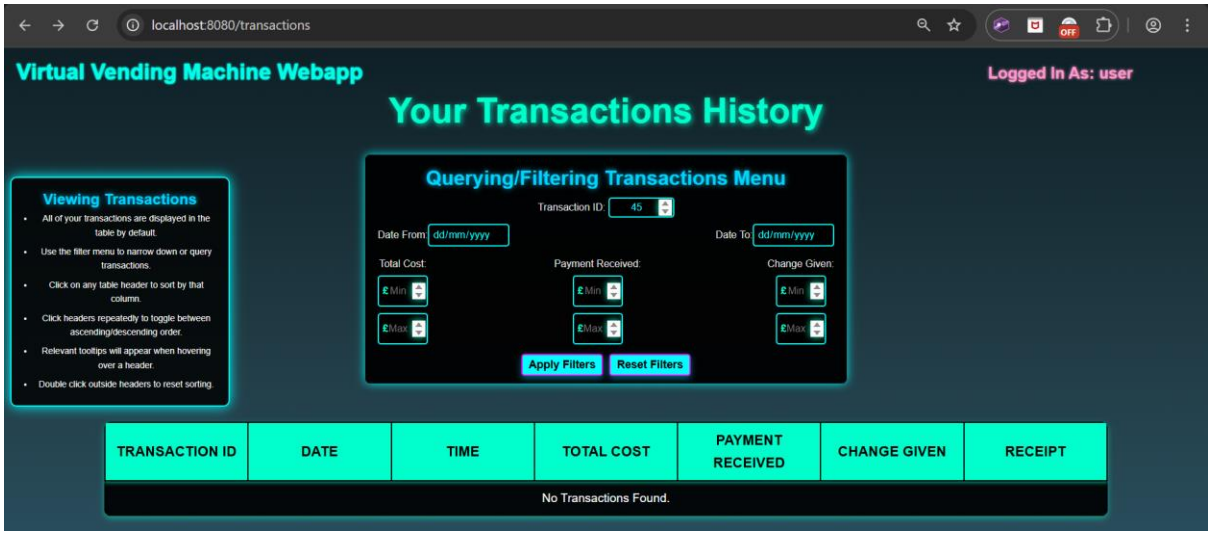
(Clicking it again flips the direction to descending.)

Filtering Testing

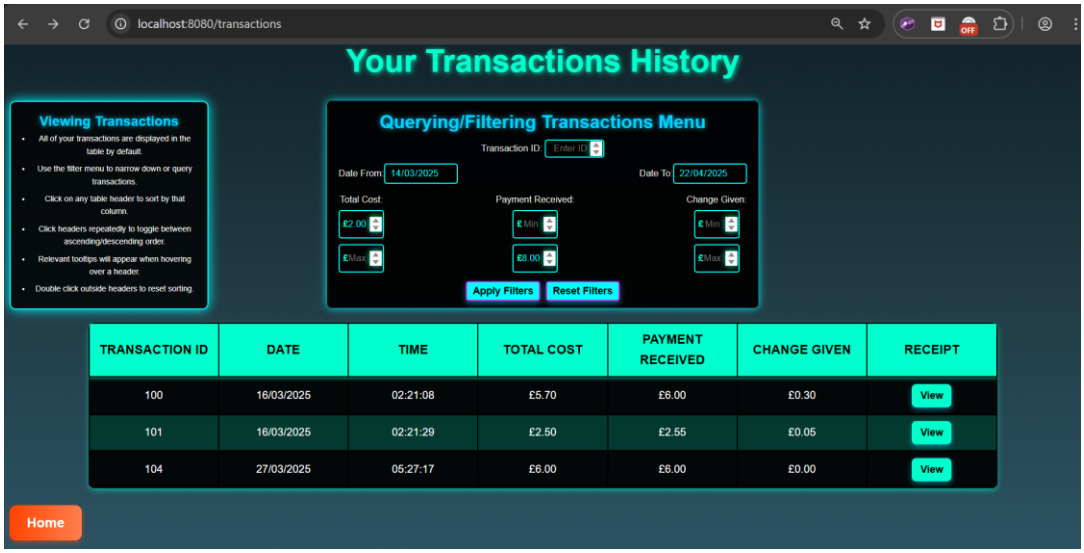
- Filtering was tested using the "Filter Transactions" menu:
    - Filtering by date range correctly narrowed results.
    - Filtering by minimum and maximum payment amounts produced the expected subset of transactions.
    - Searching by Transaction ID returned the specific transaction or an appropriate "No Transactions Found" message when it doesn’t exist for the user.
- After applying filters, sorting could still be applied dynamically on the filtered results, ensuring full interaction flexibility.



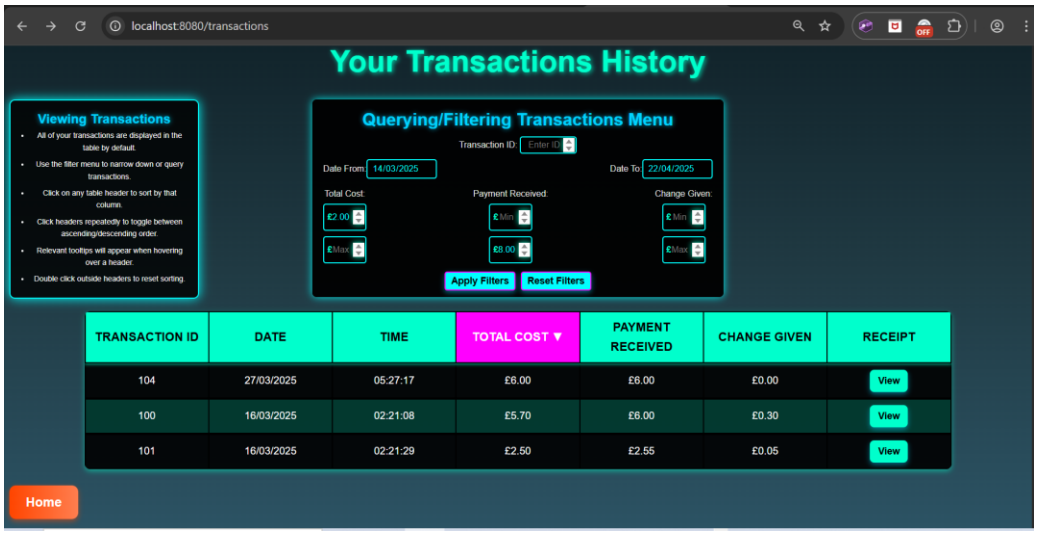
(After filtering for Transaction 91 [made by this user].)



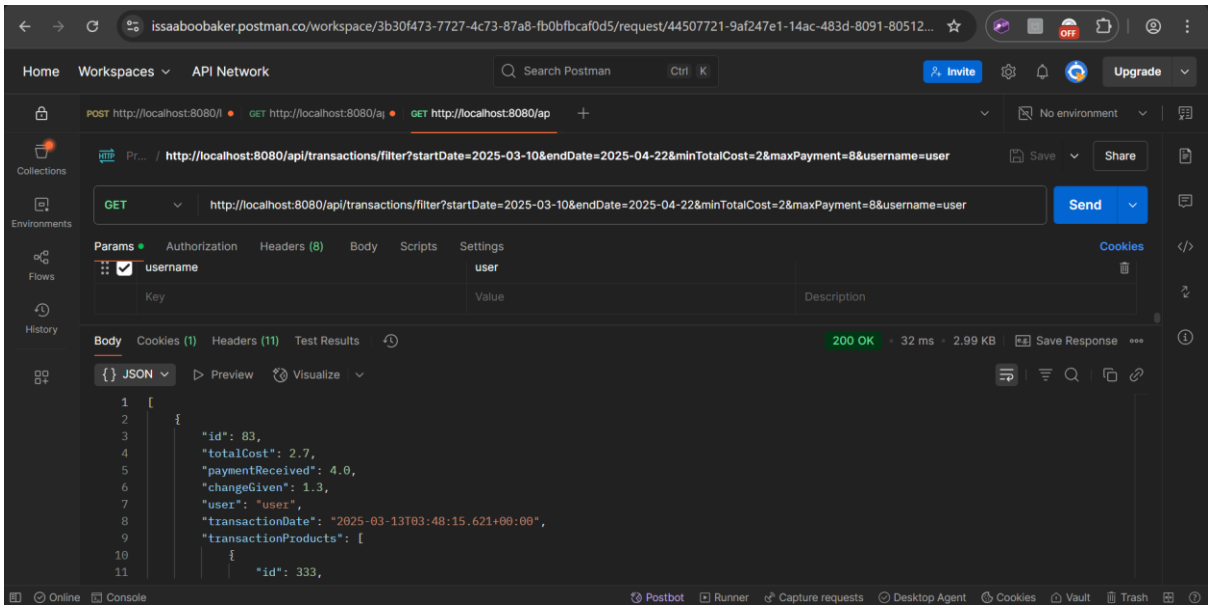
(After filtering for Transaction 45 [not made by this user].)



(After filtering for transactions with minimum Total Cost of £2.00, maximum payment received of £8.00 and dates range of 14/03/2025 to 22/04/2025.)



(Also sorting the previous filtered table by total cost descending, showing sorting and filtering can be combined.)



(Testing the URL for the above filtered table with all the filters shown as URL parameters. The request was successful and the correct JSON response with the filtered transactions is sent back.)

Interaction Stability Testing

- Rapidly applying and clearing filters multiple times did not cause any crashes or layout bugs.
- Filtering can be applied to a sorted table, and sorting can be applied to a filtered table. After resetting one of them, the table still maintains the other.
- Submitting invalid or empty filter parameters was handled gracefully, by defaulting the filter menu values.

Summary of Transaction History Testing:

- Data loading, sorting and filtering worked without errors.
- Transaction data integrity between frontend and backend was preserved.
- Dynamic interactions were handled robustly without UI glitches or fails.

Admin Controls Testing

The Admin Dashboard (‘/admin/dashboard’) was tested to ensure that stock management, auto-restock configuration and general admin page security worked correctly. Testing was carried out through combining frontend interaction validation with backend database inspection.

Admin Dashboard Access

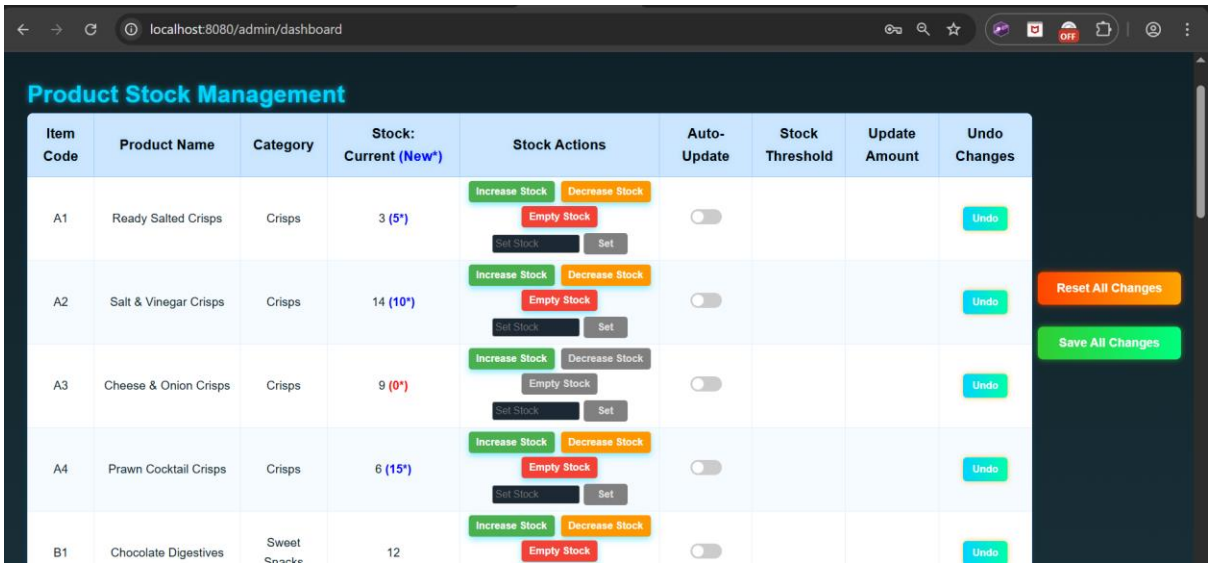
- Verified that only users with the “ADMIN” role could access the admin dashboard.
- Attempting to access the admin page as a regular user or guest correctly redirected to login or returned a forbidden error.
- This was already tested successfully in the backend testing section.

Stock Modification Tests

Admin users can manually edit product stock levels from the dashboard UI.

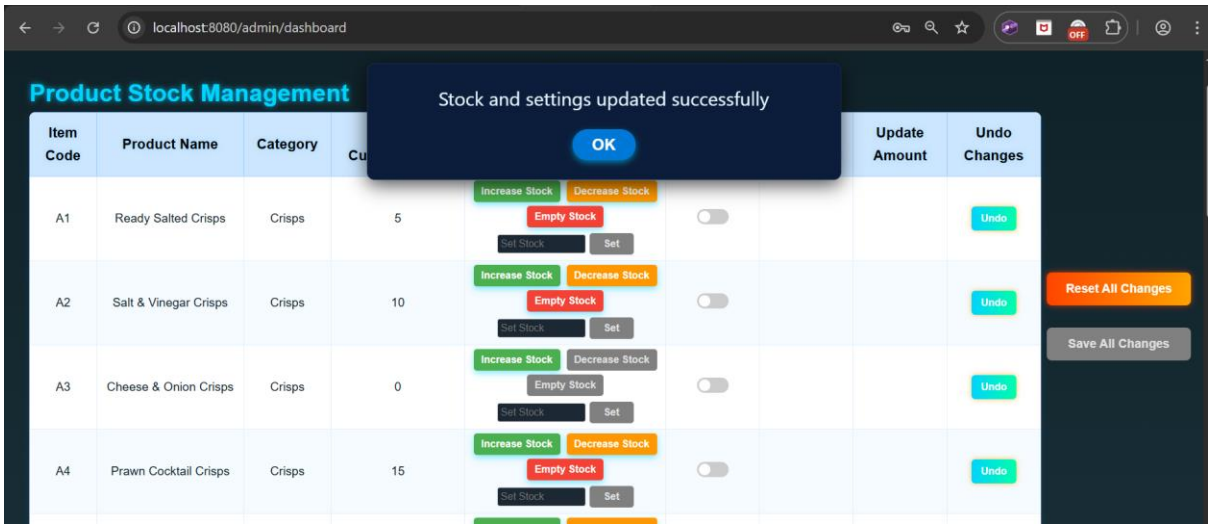
Tests Performed:

- Increased stock values via the "Increase Stock" button.
- Decreased stock values via the "Decrease Stock" button.
- Emptied stock via the "Empty Stock" button.
- Manually entered stock values using the input field.
- Pressed "Save All" to submit changes.



(Increased the stock for A1 by 2 using the “Increase Stock” button, decreased the stock for A2 by 4 using the “Decrease Stock” button, emptied the stock for A3 using the “Empty

Stock" button and manually set the stock for A4 to 15 using the "Set Stock" input field and corresponding "Set" button.)

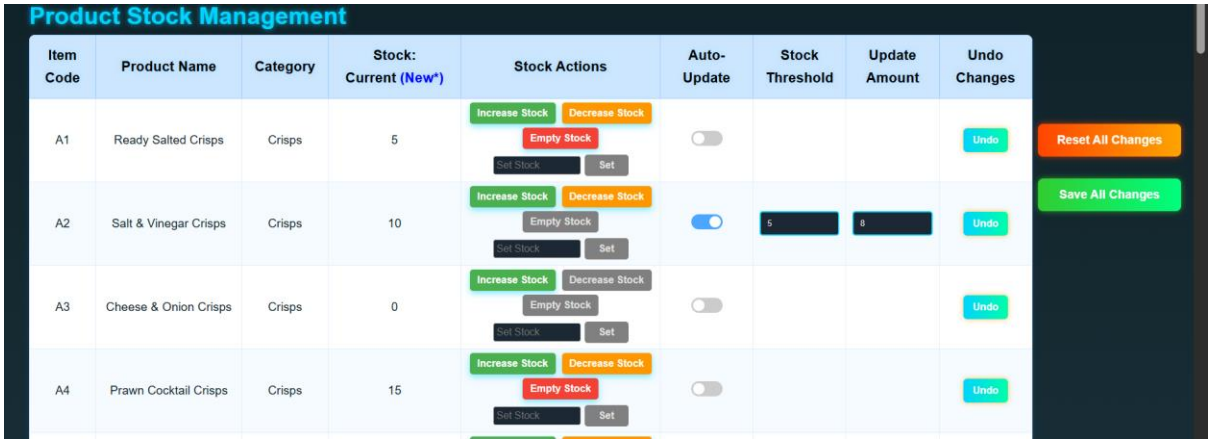


(After pressing the "Save All Changes" button, these new stock levels are applied.)

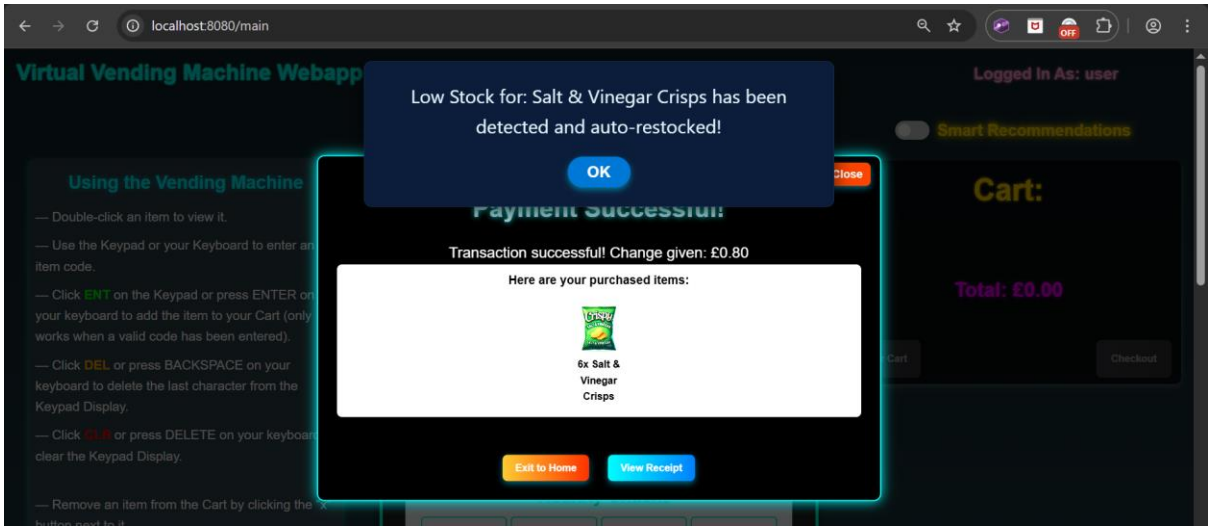
Auto-Restock Toggle and Thresholds

Auto-restock configuration was tested by:

- Toggling auto-restock ON for selected products.
- Setting a threshold and update amount.
- Saving changes and confirming database update (MySQL Workbench inspection).
- Performing a frontend purchase that dropped stock below threshold to trigger auto-restock.



(For item A2 turned on auto-update, set threshold at 5 and update amount at 8.)



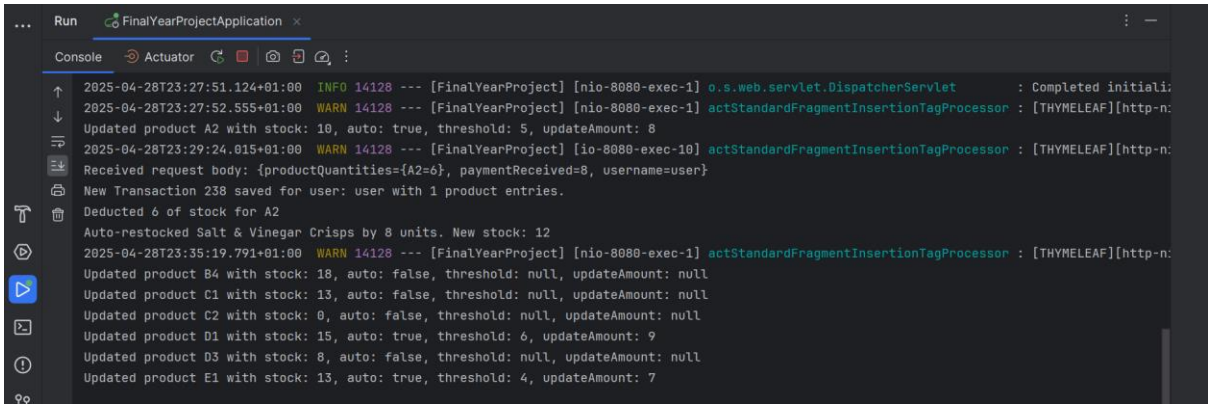
(Now after buying 6 of item A2 so it goes below the threshold, we get an auto-restock alert.)

```
2025-04-28T23:27:45.895+01:00  INFO 14128 --- [FinalYearProject] [           main] o.s.d.j.r.query.QueryEnhancerF
2025-04-28T23:27:47.004+01:00  WARN 14128 --- [FinalYearProject] [           main] JpaBaseConfiguration$JpaWebCon
2025-04-28T23:27:47.093+01:00  INFO 14128 --- [FinalYearProject] [           main] r$InitializeUserDetailsManager
2025-04-28T23:27:47.779+01:00  INFO 14128 --- [FinalYearProject] [           main] o.s.b.w.embedded.tomcat.Tomcat
2025-04-28T23:27:47.790+01:00  INFO 14128 --- [FinalYearProject] [           main] c.e.f.FinalYearProjectApplicat
2025-04-28T23:27:51.120+01:00  INFO 14128 --- [FinalYearProject] [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost]
2025-04-28T23:27:51.121+01:00  INFO 14128 --- [FinalYearProject] [nio-8080-exec-1] o.s.web.servlet.DispatcherServ
2025-04-28T23:27:51.124+01:00  INFO 14128 --- [FinalYearProject] [nio-8080-exec-1] o.s.web.servlet.DispatcherServ
2025-04-28T23:27:52.555+01:00  WARN 14128 --- [FinalYearProject] [nio-8080-exec-1] actStandardFragmentInsertionTa
Updated product A2 with stock: 10, auto: true, threshold: 5, updateAmount: 8
2025-04-28T23:29:24.015+01:00  WARN 14128 --- [FinalYearProject] [io-8080-exec-10] actStandardFragmentInsertionTa
Received request body: {productQuantities={A2=6}, paymentReceived=8, username=user}
New Transaction 238 saved for user: user with 1 product entries.
Deducted 6 of stock for A2
Auto-restocked Salt & Vinegar Crisps by 8 units. New stock: 12
```

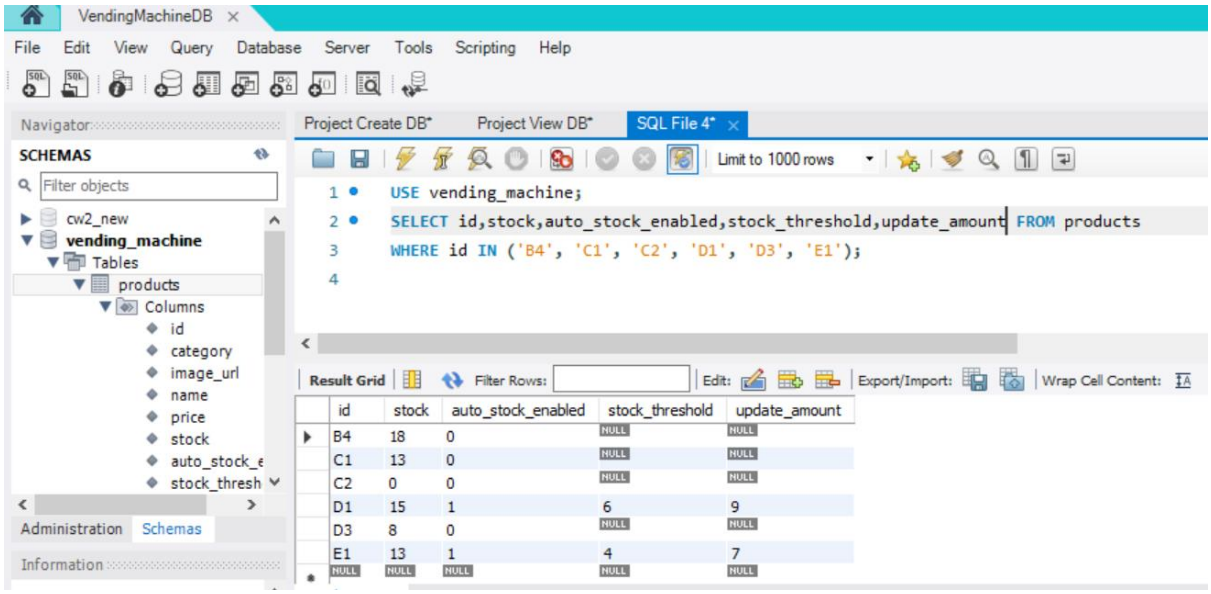
(The system logs show it has been auto restocked by the update amount which was set at 8.)

Exploratory Testing

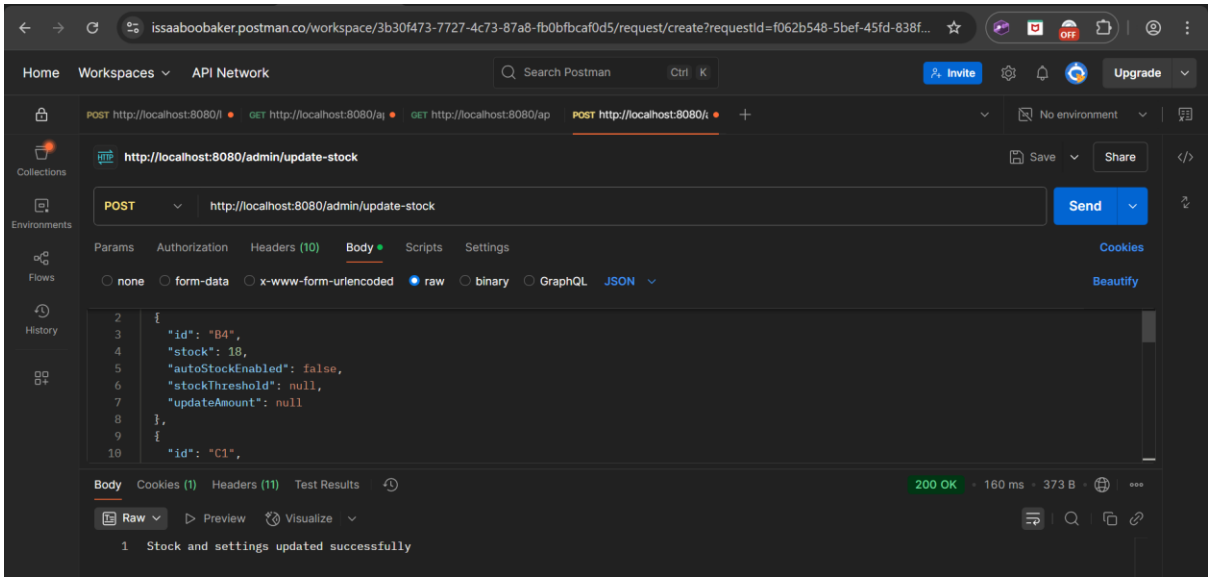
- Rapid edits to stock fields across many products, followed by clicking "Save All Changes".
- Toggled auto-restock on/off rapidly between multiple products before saving.



(System log after making multiple changes to different products and saving all at once.)



(Evidence of the database table ‘products’, showing the system log was correct and the changes made from the admin page were correctly stored in the database.)



(Also tested the admin dashboard update stock settings API (‘/admin/update-stock’) using Postman with the exact same stock changes from above: successfully sends the changes and returns the success message.)

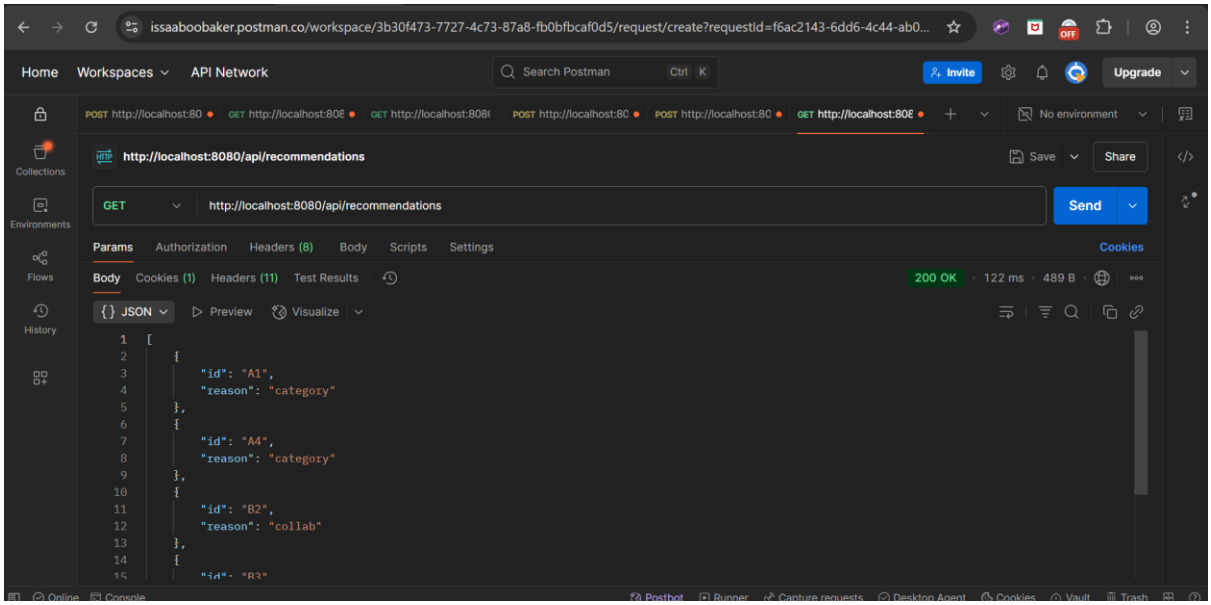
Smart Recommendations Testing

Testing of the Smart Recommendations feature focused primarily on validating the backend scoring algorithm and verifying correct frontend integration. The goal was to ensure that recommendations adapted dynamically to user purchase history and that weighting factors (category match, price similarity, collaborative filtering) behaved as intended.

Backend Recommendation

The backend algorithm was tested manually using Postman:

- Sent authenticated GET requests to the ‘/api/recommendations’ endpoint as users with varying transaction histories (e.g. new users, active users, single-category buyers).
- Confirmed that:
    - For new users (no transaction history) recommendations were based solely on collaborative filtering (popular items), since we have no data for category or spending.
    - For experienced users, recommendations increasingly weighted favourite categories and spending patterns.
    - Final product scores were a combination of all three factors with category match dominant for most users.

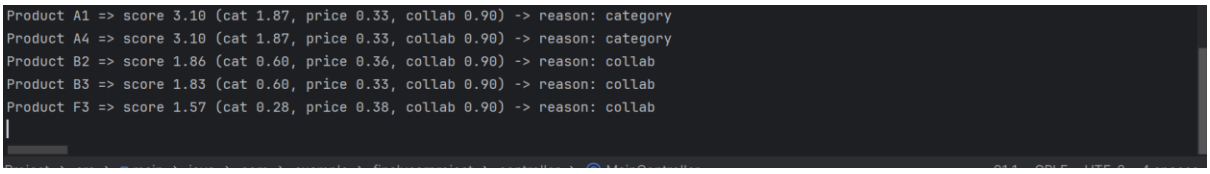


(Tested the recommendations API using Postman. It successfully returns the JSON response containing the IDs of the recommended products, each with its primary reason, to be used in tooltips.)

Log Inspection

The backend ‘RecommendationService’ generates debug logs showing the contribution of each factor’s score for the recommended products.

- Verified through system logs that:
    - Category boosts were higher when the user consistently bought from certain categories.
    - Price proximity was accurately calculated and boosted items within ±20% of average spend value.
    - Collaborative boosts adjusted inversely based on the number of the user’s previous purchases.



(The entire breakdown of the scoring from the algorithm was shown in the system logs.)

Edge Case Handling

The system was tested for edge cases:

- Users with no purchases received purely collaborative recommendations (expected as no category or spending data to use).
- Users with 1–2 purchases showed a mix between collaborative and initial category/price influence.
- Users with many purchases received more category-based and price-range-matched suggestions.

```
Product F1 => score 3.00 (cat 0.00, price 0.00, collab 3.00) -> reason: collab
Product E1 => score 3.00 (cat 0.00, price 0.00, collab 3.00) -> reason: collab
Product F2 => score 3.00 (cat 0.00, price 0.00, collab 3.00) -> reason: collab
Product D1 => score 3.00 (cat 0.00, price 0.00, collab 3.00) -> reason: collab
Product E2 => score 3.00 (cat 0.00, price 0.00, collab 3.00) -> reason: collab
```

(System logs of recommendations for a new user with no transaction history shows that there is no score for categories or price, and a high score for collaborative filtering, which is used as the reason for all recommendations.)

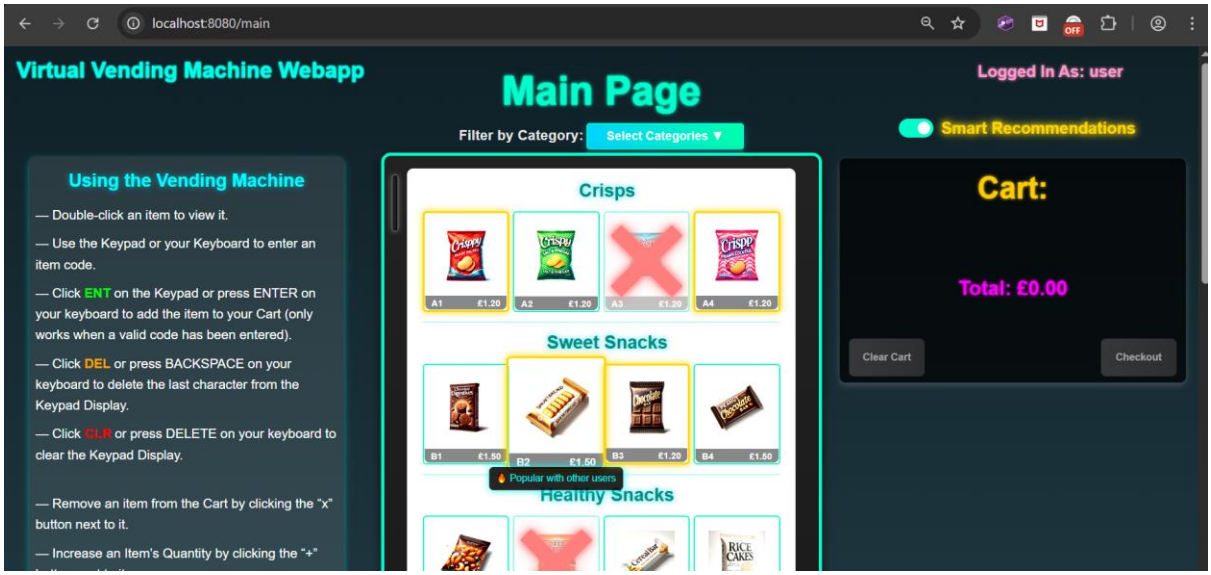
```
Product A1 => score 3.22 (cat 1.30, price 1.02, collab 0.90) -> reason: category
Product A2 => score 3.22 (cat 1.30, price 1.02, collab 0.90) -> reason: category
Product A3 => score 3.22 (cat 1.30, price 1.02, collab 0.90) -> reason: category
Product B1 => score 2.28 (cat 0.08, price 1.30, collab 0.90) -> reason: price
Product B2 => score 2.28 (cat 0.08, price 1.30, collab 0.90) -> reason: price
```

(In contrast, a user with transactions history, has their recommendations based on category and price. The score for collaborative has dropped because it is adaptive based on how many transactions a user has, and in this case, it has become weaker to allow the other factors have priority because they are more personalised.)

Frontend Display and Interaction

The frontend display of smart recommendations on the main page was tested for correct behaviour:

- Recommended products were visually highlighted with a neon gold glow.
- Hovering over a highlighted product displayed a tooltip indicating the reason for recommendation ("Based on your favourite category", "Popular with others" or "Matches your spending habits").
- Turning off smart recommendations via the toggle button immediately removed all highlights and tooltips without requiring a page reload.



(When toggle switch for Smart Recommendations was turned on, the recommended products were successfully highlighted with a glowing gold border. The dominant factor reason appeared on a tooltip, which was different between items.)

Summary of Smart Recommendations Testing

The Smart Recommendations feature was fully validated across backend and frontend layers. Backend recommendation scores dynamically changed based on user behaviour. Frontend visual effects correctly responded based on backend data, therefore enhancing user experience without impacting system stability.

Appendix E - Planning and Timescales

The time plan and Gantt Chart for the full project development process has been taken from my Interim Report and been updated to match the stage we are currently at. Evidence of my progress across the time plan can be found in my GitLab Commits and Project Log.

Semester 1

Stage	Task	Start Date	End Date	Status
1. Research & Planning				
	Research existing vending machine simulations to identify features and user expectations.	22/10/2024	07/11/2024	Completed
	Define project aims and objectives	08/11/2024	10/11/2024	Completed
	Define system requirements	11/11/2024	13/11/2024	Completed
	Develop a detailed project time plan (this)	14/11/2024	16/11/2024	Completed
Checkpoint 1		17/11/2024	19/11/2024	
2. System Design				
	Design the system architecture using Spring MVC Framework.	20/11/2024	24/11/2024	Completed
	Plan the database layout to manage product stock	20/11/2024	26/11/2024	Completed
	Create wireframes for the user interface.	20/11/2024	30/11/2024	Completed
Checkpoint 2		01/12/2024	03/12/2024	
3. Backend Development				
3.1 Set Up Backend	Create a Spring Boot project using Gradle.	21/11/2024	03/12/2024	Completed

	Add dependencies for Spring MVC and other required libraries.	21/11/2024	05/12/2024	Completed

Semester 2

Stage	Task	Start Date	End Date	Status
3.2 Manage Products (Stock)	Create a Product class with attributes: name, price, and stock level.	06/12/2024	08/12/2024	Completed
	Set up a service for product operations	09/12/2024	11/12/2024	Completed
	Implement the database connection to store product information.	12/12/2024	14/12/2024	Completed
	Add a method to update stock levels when they become low.	15/12/2024	17/12/2024	Completed
3.3 Handle Transactions	Create a Transaction class to store details like purchase items and total price.	18/12/2024	20/12/2024	Completed
	Code the logic for selecting multiple products and calculating the total cost.	21/12/2024	23/12/2024	Completed
	Add a method to check if the user's payment matches the total price (or if it exceeds it).	24/12/2024	26/12/2024	Completed
	Code the logic to dispense change if required.	27/12/2024	29/12/2024	Completed
3.4 Generate Receipts	Create a method to generate a receipt with the purchased items, prices, total and change (if there is).	30/12/2024	31/12/2024	Completed
	Format the receipt for printing.	01/01/2025	02/01/2025	Completed
3.5 Testing & Debugging	Perform unit tests for product management features.	03/01/2025	05/01/2025	Completed

	Perform unit tests for transaction logic (e.g. payment, change dispensing).	06/01/2025	08/01/2025	Completed
	Test backend code to ensure it works as expected.	09/01/2025	11/01/2025	Completed
Checkpoint 3		11/01/2025	12/01/2025	
4. Frontend Development				
4.1 Webpage Layout	Design the basic webpage layout in JSP/HTML.	13/01/2025	16/01/2025	Completed
	Create individual components for product display (e.g. product name, price, and stock).	17/01/2025	20/01/2025	Completed
4.2 Cart Functionality	Implement a cart system and add JavaScript to dynamically show selected items.	21/01/2025	25/01/2025	Completed
4.3 Styling	Add styling using CSS to ensure the layout looks appealing.	26/01/2025	29/01/2025	Completed
4.4 Dynamic Features	Add more JavaScript for interactivity.	30/01/2025	04/02/2025	Completed
	Prepare for interview	03/02/2025	17/02/2025	Completed
4.5 Frontend Testing	Test all frontend components for functionality.	05/02/2025	08/02/2025	Completed
Checkpoint 4		09/02/2025	10/02/2025	
5. Integration & Testing				
	Ensure connection between backend and frontend.	11/02/2025	13/02/2025	Completed
	Conduct integration testing for complete functionality.	14/02/2025	16/02/2025	Completed
	Fix any identified bugs or issues.	17/02/2025	19/02/2025	Completed
Checkpoint 5		20/02/2025	22/02/2025	
6. Finalisation				

	Complete project documentation: user manual and project log.	23/02/2025	01/03/2025	Completed
	Revisit unfinished tasks or add additional features/ improvements.	02/03/2025	29/03/2025	Completed
	Final checks and testing.	30/03/2025	31/03/2025	Completed
	Dissertation.	01/04/2025	01/05/2025	Completed
	Prepare for submission.	01/04/2025	01/05/2025	Completed
	Viva.	01/04/2025	19/05/2025	In Progress

Gantt Chart

As most details have been shown in the tables, the Gantt chart has been simplified to show the main stages of development and their time frames.