UNIVERSITY OF LEICESTER

School of Computing and Mathematical Sciences

CO7201 Individual Project

Final Report

Stock Control System

Apurva Vivek Kulkarni

Avk7@student.le.ac.uk

Student Id - 239064472

Project Supervisor: Prof Anthony Conway
Principal Marker: Dr Severi Paula

Word Count: 13399
16/05/2025

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 pages. 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: Apurva Kulkarni
Date: 15/05/2025

Acknowledgements

I would like to express my deepest gratitude to my supervisor Prof Anthony Conway, for their valuable guidance and continuous support through my thick and thin times and patience through this research. Their expertise, continuous motivation to push my limits and insight have been fundamental to the success of this work. I am immensely grateful for the countless hours they dedicated to refine my ideas which helped me improving the quality of this thesis.

I also want to extend my sincere gratitude to my Principal marker Dr Severi Paula for taking significant interest in my project and their constructive feedback in my project, challenging questions and insightful suggestions to improve my project.

I am deeply indebted to my family for their unconditional love, support and faith in me. To my parents Ujwala and Vivek Thank you for your constant motivation, faith, understanding during the demanding times and being my backbone in my MSc Journey.

Finally, to my friends Gandhali and Yogesh thank you for your patience, positivity and encouragement which helped me pass through the tough times. Your understanding, when I felt homesick and when I was wrapped in research work means a lot to me.

Lastly, I would say thank you God for protecting me, guiding me and making me capable to complete this project.

Thank you all!

Abstract

This project presents the design, development, and implementation of a Stock Control System specifically tailored for a computer warehouse environment. The aim of the project was to address the limitations of traditional manual inventory management processes and introduce a centralized, real-time system that enhances accuracy, efficiency, and decision-making. By building an integrated platform that connects clients and administrators through secure dashboards, the system ensures a seamless flow of inventory operations, order placement, and stock monitoring.

Key objectives of the project included providing real-time product tracking, generating automated low-stock alerts, enabling role-based access control, offering predictive sales analytics, and ensuring data security through verified user onboarding. The system is developed using a Flask backend and an Angular frontend, leveraging MySQL for database management and SMTP protocols for secure email communication. Features such as stock updates triggered immediately upon order placement, shelf capacity visualization, and future sales prediction using machine learning algorithms are embedded to support efficient warehouse operations and proactive stock replenishment strategies.

A significant achievement of the project is the successful implementation of a Linear Regression model to predict future sales of products based on historical customer purchase data, which enables data-driven decisions in inventory planning, supplier ordering, and discount management to optimize stock levels and reduce overstock or stockouts. The project also implements a comprehensive reporting and analytics dashboard that displays top-selling products, least demanded items, profit trends, and future stock forecasts, empowering administrators with data-driven insights to support inventory and sales planning.

The system underwent thorough testing to validate its performance, reliability, and user experience. Results demonstrate that the Stock Control System effectively reduces the risk of stockouts, improves the speed and accuracy of inventory updates, and enhances operational visibility. Clients benefit from a clean, intuitive interface for order management, while administrators gain access to powerful monitoring and reporting tools that support informed decision-making.

Overall, the developed Stock Control System provides a scalable, secure, and user-centric solution for modern computer warehouse management, bridging the gap between traditional stock handling practices and the needs of today’s fast-paced, technology-driven supply chain environments.

Table of Content

Chapter 1

Introduction

Stock Control System management plays a vital role in ensuring operational efficiency across supply chains. As businesses increasingly rely on technology for their warehousing needs, the shortcomings of traditional inventory systems such as maintaining physical record books or using Excel spreadsheets are becoming increasingly evident. These basic methods are prone to human error, lack real-time visibility, and are inadequate for managing complex operations in fast-paced industries. As a result, they often lead to stock inaccuracies, delays, and financial losses due to overstocking or stockouts. This sensitivity of the performance to the inventory inaccuracy becomes even greater in systems operating in lean environments [1]. In such environments, an intelligent and centralized inventory management system becomes essential for maintaining accuracy and supporting efficient decision-making.

1.1 Background

Modern warehouses dealing with high-value, fast-moving components such as CPUs, GPUs, and storage devices require accurate stock visibility, secure user management, and timely replenishment mechanisms. Traditional stock control methods such as spreadsheets or manual logs not only introduce human error but also lack the ability to support advanced operations such as predictive analytics, system-wide alerts, and role-based access control. With the proliferation of web-based applications, there is a growing opportunity to automate and centralize warehouse activities. This includes real-time tracking of stock levels, secure access based on user roles (e.g., administrators, clients), and the use of analytical dashboards to inform procurement and sales strategies. These improvements are especially critical for small to mid-sized warehouses seeking cost-effective and scalable solutions.

1.2 Motivation

As businesses strive for operational excellence, the need to transition from reactive to proactive inventory management becomes critical. This project is driven by the realization that traditional inventory systems are no longer sufficient for managing modern warehouses. The adoption of technologies to manage inventory and supply chain processes represents a pivotal step towards achieving operational excellence, enhancing efficiency, and gaining a competitive edge in today's dynamic business landscape [2].
Machine learning offers a transformative approach by predicting sales trends and helping warehouse managers plan restocking activities based on data. When integrated with a modern web application framework, this approach can lead to a powerful tool that increases efficiency, reduces human error, and enhances business agility [3].

1.3 Aim and Objectives

1.3.1 Aim

The aim of this project is to design and implement a secure, centralized, and intelligent Stock Control System for managing computer warehouse. The Stock Control System is a web-based application designed to manage and monitor stock levels of computer components in a computer warehouse. It ensures real-time inventory tracking, optimizes the stock assembly process, and enhances operational efficiency by providing accurate reporting and invoicing features. The system helps warehouse managers prevent stock shortages, reduce excess inventory, and improve overall productivity.

A key feature of this system is location tracking, which allows precise identification of where each item is stored or moved during stock assembly. Products are segregated zone-wise and shelf-wise within the warehouse for efficient management and retrieval. The system also provides low stock alerts to the admin both on the dashboard as soon as they log in and via automated email notifications to ensure timely restocking.

To ensure the authenticity of user-provided email addresses, the system implements email verification using SMTP, which sends a One-Time Password (OTP) during registration. This validation prevents fake email usage, especially since email receipts are automatically sent to users upon purchasing products.
It also incorporates role-based access control, where administrative users must be approved by the Head Admin before gaining full access. By supplying context filters during the definition of a role, a security administrator can easily limit the applicability of users' role memberships to particular subsets of the target objects [4].

1.3.2 Objectives

To meet this aim, the project is designed with the following objectives:

1. **Real-Time Inventory Updates**
   Automatically update stock quantities in the system as client orders are placed, ensuring accuracy and visibility.
2. **Automated Low-Stock Alerts**
   Notify administrators upon login when inventory for any item falls below a predefined threshold to enable timely procurement via email.
3. **Role-Based Access Control with Approval Workflow**
   Implement secure authentication to differentiate between administrator and client access. Admin registration requires approval from the Head Admin before account activation
4. **Predictive Sales Forecasting Using Machine Learning**
   Use a Linear Regression model to predict sales over the next 90 days based on historical data, enabling informed restocking decisions.
5. **Interactive Reporting and Analytics**
   Integrate Apex Charts to visually present metrics such as top-selling products, monthly sales, and profit trends through dashboards.
6. **Email-Based Notifications and PDF Invoicing**
   Use SMTP email services to send OTPs for secure user registration and PDF invoices to clients upon order placement.

1.4 Structure of the Report

The report is structured as follows:

- **Chapter 2: Literature Review** – Analyses current trends in inventory management systems, predictive analytics, and related technologies.
- **Chapter 3: System Objectives** – Presents the functional and non-functional requirements refined through research and design planning.
- **Chapter 4: System Architecture and Database Design** – Describes the technical stack, backend–frontend interactions, and MySQL schema design.
- **Chapter 5: Implementation** – Covers the implementation of key modules such as authentication, stock management, alert systems, and invoicing.
- **Chapter 6: Sales Prediction Model** – Explains the supervised learning algorithm (Linear Regression), dataset preparation, model training, and accuracy.
- **Chapter 7: Case Study** – Demonstrates practical use cases of the system in a simulated warehouse scenario with sample data.
- **Chapter 8: Evaluation** – Discusses system performance, user feedback, and the prediction model's accuracy (over 85%).
- **Chapter 9: Conclusion and Future Work** – Summarizes achievements and outlines future enhancements including supplier APIs, dynamic pricing, and IoT-based stock sensors.

Chapter 2

Literature review

A comprehensive literature review is essential to understand the existing developments, methodologies, and technologies relevant to inventory management and stock control systems. This section explores the core research questions driving the project, outlines the strategies employed to source academic materials, critically examines existing inventory management systems, and reviews the key technologies that have enabled modern stock control solutions. By synthesizing insights from recent studies and technological advancements, this literature review focuses on definitions covering warehouse management, small and medium-sized enterprises, enterprise resource planning (ERP) systems, and warehouse technology of the Stock Control System [5].

2.1 Research Questions

The evolution of inventory management systems has been driven by the need for accuracy, real-time tracking, and efficient decision-making in complex supply chain environments. Despite technological advancements, many small to medium-sized computer warehouses continue to face challenges related to manual processes, delayed updates, and lack of predictive capabilities. The primary goal of this project is to develop a centralized, real-time Stock Control System that integrates automation, machine learning, and user-centric design to address these challenges. These technologies are redefining the operational paradigms of inventory management, order fulfillment, predictive maintenance, and resource optimization [6].

To guide the research and development of the system, the following research questions were formulated:

RQ1: How can inventory management in computer warehouses be optimized through real-time stock movement tracking and automated alert systems?
Manual inventory tracking often results in stock discrepancies and delayed procurement decisions, leading to stockouts or overstocking. To address these challenges, this project proposes a comprehensive smart warehouse management solution that combines real-time inventory tracking with advanced predictive analytics and intelligent stock alert mechanisms [7].
RQ2: How can predictive analytics, specifically machine learning models, be utilized to forecast future inventory requirements and support proactive stock replenishment?
Forecasting future demand remains a complex challenge for warehouse managers. By employing supervised machine learning models, such as Linear Regression, this project aims to predict future sales trends and recommend optimal stock levels, thereby supporting informed and timely purchasing decisions.

RQ3: How can role-based authentication and secure communication channels enhance the reliability and trustworthiness of a warehouse management system?
Security is a major concern in web-based inventory management systems, especially when

dealing with sensitive transactional data. This research question investigates how implementing OTP-based email verification during registration and role-based user access control can strengthen system security and improve user confidence in daily operations.

RQ4: What are the critical success factors for designing a scalable, modular Stock Control System that can adapt to future technological enhancements such as IoT integration or automated supplier management?
Scalability and adaptability are key for ensuring the long-term viability of inventory systems. This research question explores architectural strategies that allow for easy integration of future technologies like real-time IoT-based stock sensors and direct supplier API linkages [7].

By systematically addressing these research questions, this project aims to contribute both theoretically and practically to the domain of warehouse inventory management systems, particularly within the specialized context of computer hardware warehouses.

2.2 Search Strategy

To build a comprehensive understanding of the current state of inventory management systems, predictive analytics techniques, and secure web-based solutions, a systematic search strategy was developed. The search process targeted a wide range of academic and technical resources to ensure that the literature review would be grounded in credible and contemporary research.
Databases such as Google Scholar, ResearchGate, and IEEE Xplore were utilized to locate relevant studies, due to their extensive coverage of peer-reviewed journals, conference papers, and technical reports across multiple disciplines. The search was guided by key themes aligned with the project's research questions, including real-time inventory tracking, machine learning applications in forecasting, secure authentication for web systems, and scalable warehouse technologies.
Keywords were combined using Boolean operators to refine search results effectively. Searches combined terms like "inventory management systems" with "real-time stock updates" and "supply chain optimization" to explore operational improvements through digital tracking. Similarly, terms like "predictive analytics," "machine learning," and "sales forecasting" were used to find literature related to data-driven demand prediction models. Secure registration systems linked to OTP verification and web application security were also explored.
The inclusion criteria focused on articles published between 2015 and 2025, written in English, and addressing inventory control, predictive modeling, system security, or warehouse management technologies. Priority was given to studies demonstrating real-world applications relevant to small-to-medium enterprises. After an initial identification of approximately 300 articles, careful screening reduced the pool to around 40 papers critically analyzed throughout this review.
This structured search approach ensures that the Stock Control System design is informed by the latest academic research and industrial practices.

2.3 Review of Existing Inventory Management Systems

Inventory management systems have evolved from traditional manual processes to complex, technology-driven solutions. Documentation from various departments reflected the inheritance of the manual inventory errors through increased expenses and financial loss [8]. The transition to computerized inventory systems began with basic databases and Enterprise Resource Planning (ERP) which include three major giants providing ERP systems namely SAP, Oracle and Microsoft with different approaches to target market. [9].

Existing systems typically offer barcode scanning, supplier management, real-time tracking, and basic reporting. Yet many lack predictive analytics capabilities and proactive inventory control, relying instead on static reorder points. Moreover, small warehouses face challenges integrating these systems into daily operations, leading to poor adoption rates.

In terms of security, most systems offer basic password authentication but lack advanced two-step verification and secure email-based registration, exposing vulnerabilities. Thus, there remains a significant gap for affordable, predictive, and secure inventory systems tailored for specialized industries like computer hardware storage. To protect against this danger, user authentication is an essential part of any security infrastructure, serving as the first line of defence [10].

2.4 Review of Technologies used for Stock Control Systems

The technological landscape for stock control systems has broadened with advancements in software frameworks, database management, cloud computing, and machine learning.
On the backend, we have utilized Flask, a lightweight and extensible web framework written in Python, we can leverage its simplicity and flexibility to develop a scalable and efficient API [11]. For the frontend, Angular JS enable dynamic, responsive interfaces crucial for real-time inventory visibility and management.
The two most extensively used relational databases are MySQL and Oracle. MySQL is more popular with the websites [12]. They support concurrent transactions, real-time updates, and efficient data querying.
Secure communication is bolstered through SMTP email services integrated for user registration verification via OTP and for sending automated receipts, thus enhancing system trustworthiness.
Predictive analytics, particularly using Linear Regression models, allows stock forecasting based on historical sales data, enabling proactive procurement decisions. The findings suggest that the adoption of predictive analytics can significantly enhance operational efficiency, reduce costs, and foster a moreagile approach to product planning [13].
Integrating such machine learning pipelines enhances stock control and minimizes risks associated with overstocking or stockouts.
Data visualization using libraries like ApexCharts and Chart.js enables administrators to make rapid, informed decisions based on real-time sales trends, profit margins, and forecasted demands.

2.5 Summary of Literature Gaps

The review of existing inventory management systems and emerging technologies reveals several critical gaps that this project aims to address.

First, while real-time inventory tracking has been widely implemented, many systems still rely on manual threshold setting for reorder levels, lacking integration with predictive models that dynamically forecast future demand. Incorporating machine learning algorithms, such as Linear Regression, to automate sales forecasting represents a significant opportunity for innovation.

Second, many small-to-medium-sized warehouse operators face barriers to adopting advanced inventory management systems due to high costs, complexity, and scalability issues [5]. There is a need for lightweight, customizable, and affordable solutions that meet specific operational needs without overwhelming users with unnecessary features.

Third, system security in inventory management has often been limited to basic password protection. The inclusion of secure, two-step verification during registration, supported by SMTP-based email services, remains underutilized in this domain. Enhancing security protocols ensures not only the protection of sensitive stock and user data but also builds user trust.

Finally, while several systems offer reporting dashboards, they often lack actionable insights derived from real-time analytics and predictive modeling. Visual dashboards that integrate sales forecasts, low-stock alerts, and profit trend analysis remain relatively rare, especially in solutions designed for specialized warehouses such as computer hardware stockrooms. With the exponential growth of data generated through various warehouse operations such as inventory tracking, order processing, and supply chain logistics organizations face the challenge of transforming raw data into meaningful insights. Data visualization emerges as a vital tool in this process, enabling stakeholders to interpret complex datasets quickly and make informed decisions that drive efficiency and productivity [14].

By addressing these gaps through real-time automation, predictive analytics, enhanced security, and intuitive dashboards this project provides a significant advancement over existing stock control solutions.

Chapter 3

System Requirements and Scope

This chapter defines the technical and functional framework that guided the design and implementation of the Stock Control System. It outlines the system's core requirements both functional and non-functional that ensure the application delivers its intended capabilities in real-time inventory management, role-based operations, and predictive analytics. The chapter also identifies the system constraints that shaped architectural decisions, such as technology choices, deployment limitations, and operational boundaries within the academic context. Finally, the scope and limitations are discussed to provide a realistic view of what the system currently offers and which areas remain open for future development. Together, these elements provide a foundation for evaluating the system's completeness, practicality, and alignment with modern warehouse management needs.

3.1 Functional Requirements

Functional requirements define the core capabilities that the Stock Control System must possess in order to meet its intended operational goals. These requirements have been prioritized based on their criticality to system functionality and overall user experience. The prioritization follows the common practice of classifying requirements into essential (must-have), recommended (should-have), and optional (nice-to-have) categories to support efficient development planning and evaluation [15].

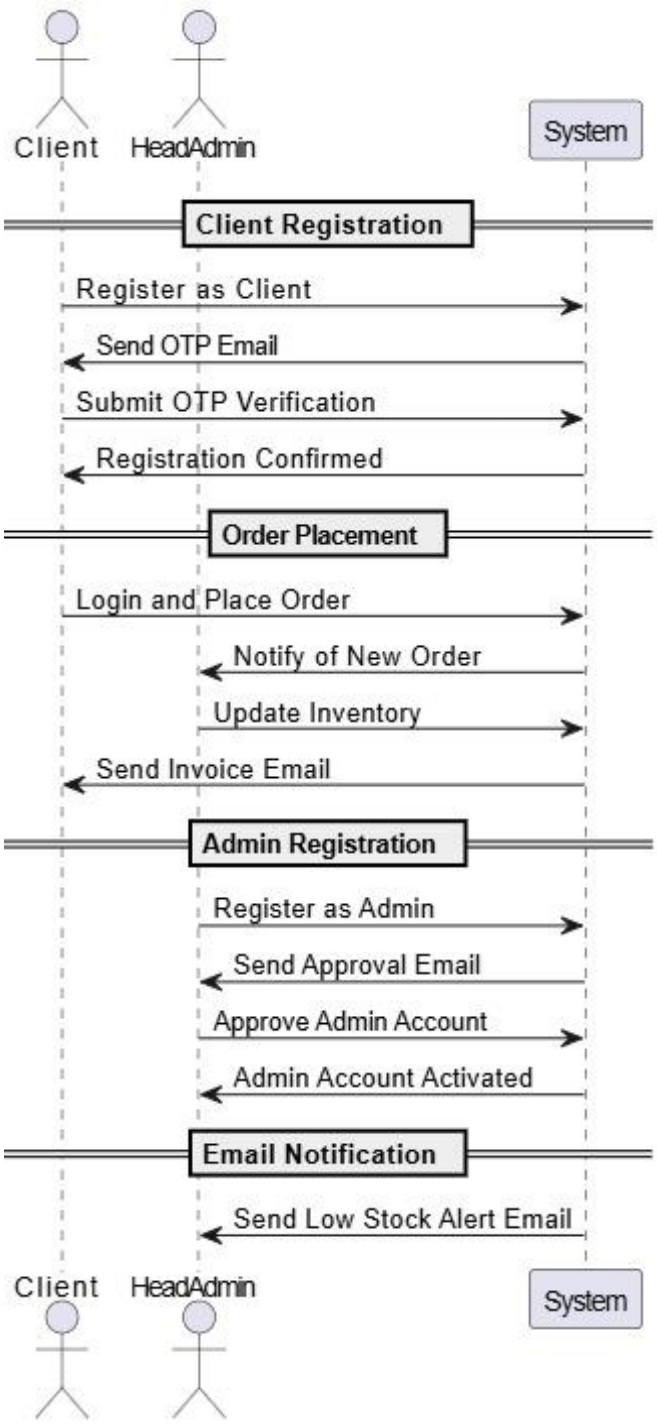
Table 1 - Requirements according to priority

Priority	Requirement Description	Implementation Details
Essential	Centralized database for real-time inventory tracking and record management	▪ MySQL database used with SQLAlchemy ORM in Flask for real-time storage, updates, and retrieval of stock data
Essential	Real-time stock movement and location tracking within the warehouse	▪ Product movements tracked via Angular forms and Flask APIs. ▪ products are mapped to zones and shelves in the backend
Essential	Low-stock alert system with dashboard warnings and automated email notifications	▪ Angular admin dashboard displays real-time alerts; Flask sends emails via SMTP when thresholds are crossed

Essential	Role-based access with two-step admin verification requiring Head Admin approval	▪ Admin registration triggers Flask SMTP email to Head Admin; Head Admin must approve via confirmation link for activation
Essential	Machine learning-based prediction of low-stock products	▪ Linear Regression model built in Python using historical sales data, integrated into Flask backend and visualized using ApexCharts in Angular
Essential	Invoice management for clients and suppliers with PDF/CSV download and email dispatch	▪ Flask generates downloadable PDF/CSV invoices. ▪ SMTP integration sends invoices to users upon order confirmation
Essential	Secure authentication, encryption, and backup for data protection and recovery	▪ Passwords hashed using bcrypt. user sessions managed securely, regular backups implemented at the database level
Recommended	Email alerts in addition to dashboard notifications	▪ Flask sends email notifications using SMTP when critical stock levels are detected
Recommended	Admin recommendation engine based on predicted stock demand	▪ Products with predicted low stock are flagged on admin dashboard using output from the Linear Regression model
Recommended	Graphical warehouse layout displaying shelves and zones	▪ Angular components display zone-wise and shelf-wise categorization. ▪ zones fetched via Flask APIs and visualized accordingly

Recommended	Cloud deployment and ERP/accounting system integration	▪ Proposed for future work would involve deploying Flask on cloud (e.g., AWS/GCP) and integrating with APIs from ERP tools
Optional	Discount feature for products with low forecasted sales	▪ Angular admin interface allows discount flag to be applied to selected products based on forecast results
Optional	Automated ordering system without manual admin input	▪ Proposed feature for future could integrate Flask backend triggers with predefined reorder thresholds and supplier endpoints
Optional	Mobile-friendly and voice-command-enabled stock interface	▪ Angular UI is responsive by default; voice-command feature not implemented yet but considered for future enhancements
Optional	Multi-language support and self-service portals for clients and suppliers	▪ Not currently implemented planned future enhancement to extend system usability and reduce admin dependency

Fig 1 Use Case diagram for Stock Control System



3.2 Non-Functional Requirement

The non-functional requirements are used primarily to drive the operational aspects of the architecture in my project they define the quality attributes that govern how the Stock Control System operates beyond its core functionality [16]. These include performance, security, scalability, usability, reliability, and maintainability all critical for ensuring the system performs efficiently under real-world conditions. The system enforces security through password hashing using bcrypt, secure JWT-based session handling, and strict role-based access control. A two-step admin verification mechanism ensures that new administrator accounts are only activated upon Head Admin approval, reducing the risk of unauthorized access. To ensure system reliability, scheduled MySQL database backups and exception handling mechanisms are implemented in the Flask backend to preserve data and maintain service continuity. Performance is also a priority, with real-time stock updates supported by low-latency APIs and a lightweight Angular frontend that ensures rapid page loads and form submissions. Usability is enhanced through responsive design, clean UI dashboards, and client-side validation using Angular Reactive Forms. While cloud deployment is not yet implemented, the system is architected to support cloud infrastructure for high availability and remote access in future enhancements. These non-functional attributes collectively ensure that the Stock Control System remains secure, reliable, and adaptable to real-world operational demands.

3.3 System Constraints

The development of the Stock Control System is governed by a set of technical, operational, and environmental constraints that have significantly influenced the design, implementation, and deployment of the application. These constraints were identified during the initial planning phase and are shaped by the limitations of available resources, institutional guidelines, and the defined project timeline [15].
Technologically, the system is developed using Flask for the backend, Angular 16+ for the frontend, and MySQL as the relational database. These technologies were chosen for their compatibility with the academic environment, community support, and ease of local deployment [16]. The machine learning functionality for predicting sales is intentionally limited to a basic supervised learning approach specifically, Linear Regression due to the small volume of historical data available and the need for transparent, low-complexity models that are easy to maintain [15].

Deployment is restricted to local environments, as cloud-based hosting and automated CI/CD pipelines fall outside the scope of this academic project. The system relies on Gmail's SMTP service for sending OTPs, alerts, and invoices, which while effective for small-scale applications, limits scalability for enterprise use [17]. Authentication within the system is handled using JWT tokens and bcrypt-based password hashing. Although secure, the absence of OAuth2 or LDAP support limits integration with external identity providers [18].
The application supports only modern browsers specifically Chrome, Firefox, and Microsoft Edge ensuring consistent performance and rendering across current platforms. However, no support is provided for outdated browsers such as Internet Explorer [19]. Accessibility features such as screen reader compatibility, high-contrast modes, and multi-language

interfaces have not been implemented due to time constraints, though these remain areas for future enhancement [15].

3.4 Scope and Limitations

The Stock Control System has been designed as a modular, web-based application tailored to support small to mid-sized computer hardware warehouses. Its core scope includes real-time inventory management, order processing, automated alerts for low stock, predictive sales forecasting, and role-based access control. The system allows administrators to monitor product movement, generate supplier orders, manage invoices, and utilize data-driven dashboards for procurement planning [15][17]. Clients are able to place orders, view invoices, and access personalized dashboards. A key enhancement is the use of a supervised machine learning model (Linear Regression) for short-term demand forecasting, enabling warehouse managers to plan stock replenishment based on predicted trends [20].

Additionally, the application enforces security through authentication mechanisms, OTP-based registration verification, and encrypted password handling using bcrypt. Features such as email notifications, admin approval workflows, and downloadable invoices contribute to a more professional and automated warehouse management experience [21]. The architecture supports modularity and future scalability by separating frontend, backend, and predictive components following a layered approach [17].

However, several limitations exist due to the academic nature of the project. First, the application is not deployed to a cloud platform, restricting its accessibility to local environments only. While email functionality is supported via SMTP, it is currently limited to Gmail and lacks integration with enterprise-grade services [18]. The machine learning model is limited to Linear Regression due to dataset constraints and does not incorporate advanced forecasting factors such as seasonality or external variables [20].
Moreover, the system does not include multi-language support, screen reader compatibility, or other accessibility features, which affects its inclusivity. Integration with third-party ERP systems, real-time IoT-based stock sensors, and automated supplier APIs are also outside the current scope. These limitations, while reasonable within the academic timeline, define key areas for potential enhancement in future development cycles [15].

Chapter 4

System Architecture and Design

This chapter details the structural and design-oriented aspects of the Stock Control System, explaining how each layer of the architecture contributes to the application's functionality, scalability, and modularity. It begins with an overview of the system's three-tier architecture presentation, application logic, and data which collectively support seamless interaction between users, business processes, and persistent storage. The subsequent sections explore the backend design using Flask, database schema modeling via SQLAlchemy, routing mechanisms, and session management strategies. Special attention is given to the integration of email functionality using SMTP, and the machine learning module for product demand forecasting. Each component has been developed with maintainability, security, and extensibility in mind, reflecting a design philosophy that balances real-time responsiveness with architectural clarity. The diagrams, table schemas, and implementation specifics presented in this chapter provide a complete view of how the system's functional goals are translated into technical execution.

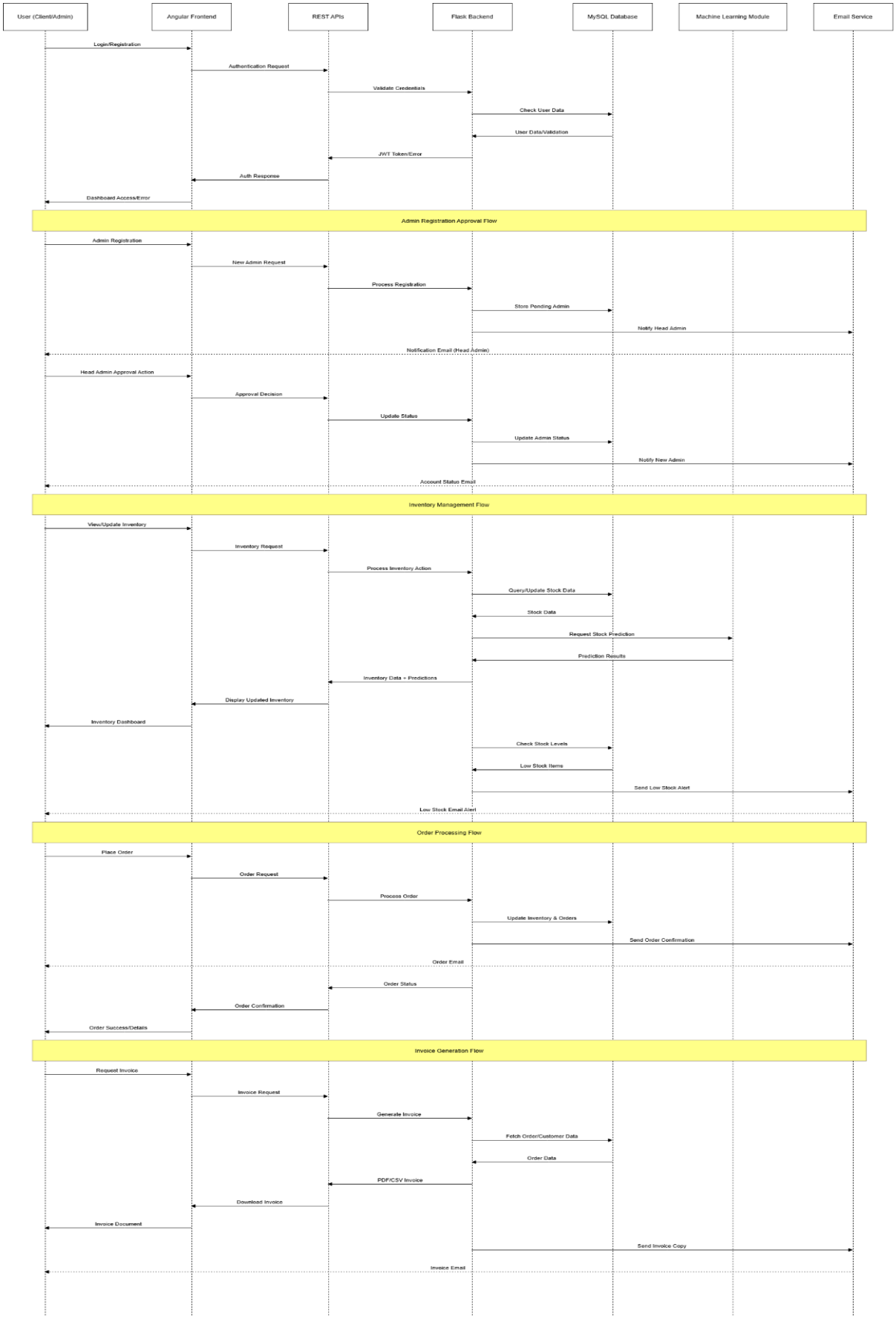
4.1 Overall System Architecture

The Stock Control System follows a three-tier architecture comprising the Presentation Layer, Application Logic Layer, and Data Layer. The frontend is built using Angular 16+, which handles all user interactions and data visualization. It communicates with the Flask backend via HTTP requests over RESTful APIs. The Flask backend manages business logic, session handling, role-based access control, and triggers database operations. It also interfaces with the machine learning module and SMTP email services. The backend communicates with the MySQL database, which acts as the persistent data store for products, users, orders, invoices, and system logs [15].

The design ensures modularity, allowing independent development and testing of components. For example, the machine learning component is kept asynchronous and stateless to prevent interference with live transactional operations. The use of RESTful APIs promotes separation of concerns and scalability, allowing for future integration with mobile apps, supplier APIs, or cloud-hosted services [17].

By adopting this layered structure, the system achieves better maintainability, enhanced performance, and the ability to isolate failures. It also lays the foundation for future enhancements such as real-time IoT stock sensors and dynamic pricing mechanisms powered by predictive analytics [22].

Fig 2 Data Flow Diagram of Stock Control System


User (Client/Admin)
Angular Frontend
REST APIs
Flask Backend
MySQL Database
Machine Learning Module
Email Service
Login/Registration
Authentication Request
Validate Credentials
Check User Data
User Data/Validation
JWT Token/Error
Auth Response
Dashboard Access/Error
Admin Registration Approval Flow
Admin Registration
New Admin Request
Process Registration
Store Pending Admin
Notify Head Admin
Notification Email (Head Admin)
Head Admin Approval Action
Approval Decision
Update Status
Update Admin Status
Notify New Admin
Account Status Email
Inventory Management Flow
View/Update Inventory
Inventory Request
Process Inventory Action
Query/Update Stock Data
Stock Data
Request Stock Prediction
Prediction Results
Inventory Data + Predictions
Display Updated Inventory
Inventory Dashboard
Check Stock Levels
Low Stock Items
Send Low Stock Alert
Low Stock Email Alert
Order Processing Flow
Place Order
Order Request
Process Order
Update Inventory & Orders
Send Order Confirmation
Order Email
Order Status
Order Confirmation
Order Success/Details
Invoice Generation Flow
Request Invoice
Invoice Request
Generate Invoice
Fetch Order/Customer Data
Order Data
PDF/CSV Invoice
Download Invoice
Invoice Document
Send Invoice Copy
Invoice Email

4.2 Backend Architecture Flask

The backend of the Stock Control System is developed using Python Flask, a lightweight and modular web framework suitable for building RESTful APIs. The architecture is designed with modularity, maintainability, and scalability in mind, allowing seamless communication with the Angular frontend and MySQL database.

4.2.1 Project Structure Explained

The backend is organized into the following core components:

Table 2 – Backend core structure

Folder/File	Purpose
app.py	Main entry point. Initializes the Flask app, sets up CORS, database, login manager, and registers all route Blueprints.
config.py	Stores configuration settings like database URI, mail server, and credentials.
models.py	Defines the SQLAlchemy ORM models such as User, Customer, Order, etc.
routes/	Contains modular route files grouped by user role or functionality. Each file is registered as a Flask Blueprint.
utils/email_utils.py	Handles sending OTPs and order receipts through Gmail SMTP.
templates/admin_approval.html	A Jinja2 HTML template used to visually approve pending admin accounts.

The **modular structure** ensures that each responsibility whether it's admin operations, customer actions, or authentication is encapsulated and maintainable.

4.2.2 Application Bootstrapping in app.py

This file does the heavy lifting of initializing and wiring everything together.
Key responsibilities:

- Configures CORS to allow communication with Angular at localhost:4200.
- Initializes the database with SQLAlchemy and sets up Flask-Migrate for schema updates.
- Registers all user-defined Blueprints for different roles.
- Sets up Flask-Login for managing user sessions.

Fig 3 – Shows the definition of Blueprints in both auth_routes.py and customer_routes.py

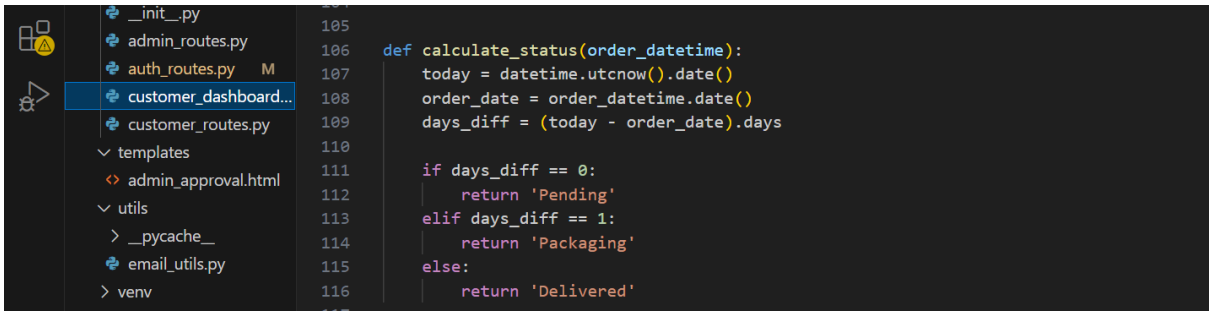
4.2.3 Blueprint-Based Routing System

Each route file under the routes/ folder encapsulates logic for a specific group of users:

- auth_routes.py: Handles login, registration, customer dashboard functions and OTP verification.
- admin_routes.py: Admin functionality like product management segregating warehouse zones and shelves and supplier orders.
- customer_routes.py: General customer profile-related operations.
- customer_dashboard_routes.py: Main business logic product browsing, order placement, tracking, etc.

For instance, customer_dashboard_routes.py consists of the /confirm_order API to handle placing new orders, checking stock, reducing inventory, and sending low-stock alerts.

Fig 4 – Stages of delivery declared in customer_dashboard.py for products purchased



4.2.4 Model Design using SQL Alchemy

The models are defined using SQL Alchemy ORM in models.py, allowing Python classes to represent database tables.

Key design features:

- Relationships between tables like User ↔ Customer, Customer ↔ Order.
- Use of back_populates for bidirectional mapping.
- Encapsulated utility methods for password hashing.

Example: One-to-One Relationship

class User(db.Model, UserMixin)

customer = db.relationship("Customer", back_populates="user", uselist=False, cascade="all, delete-orphan")

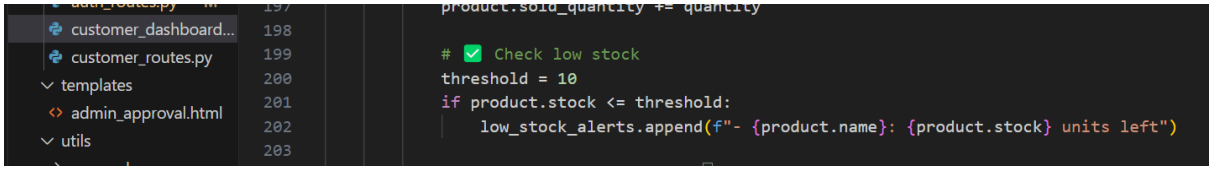
This lets a user and their customer details stay in sync and be easily queried.

4.2.5 Order Workflow and Business Logic

The confirm_order() route demonstrates robust backend logic:

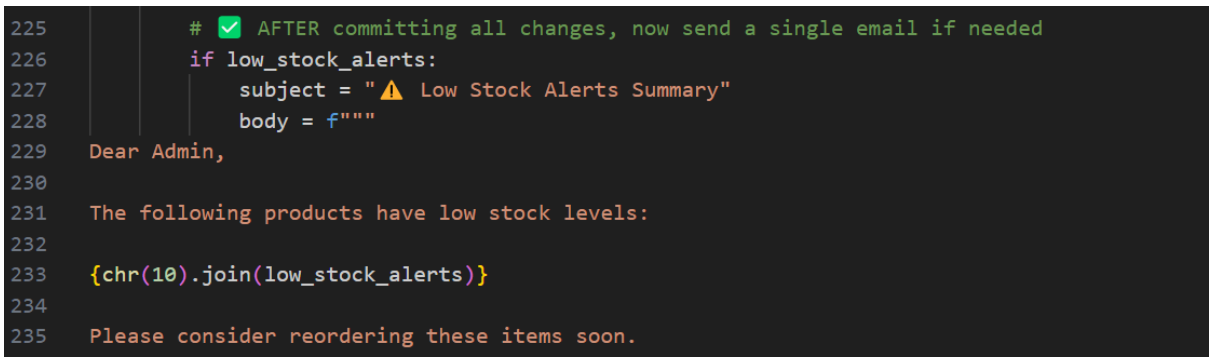
- Creates a new order and its related order details.
- Deducts stock and updates sales.
- Tracks products with low inventory and sends alert emails.
- Returns a success response with the total payable and order summary.

Fig 5 – Low Stock Alert Threshold



After order confirmation, the backend triggers an email alert to admins for restocking.

Fig 6 – Low Stock Alert Email Triggered to Admin



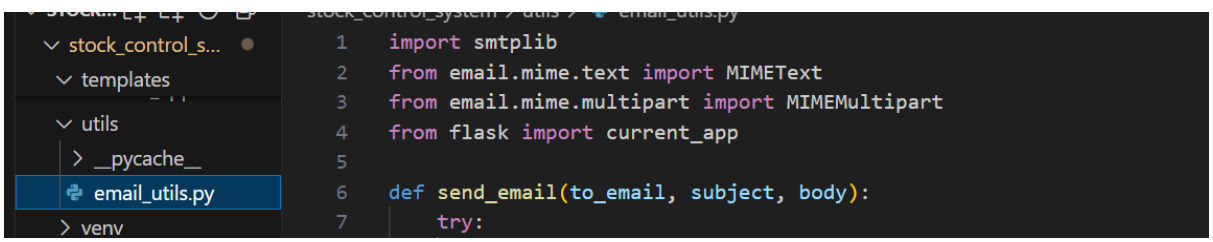
4.2.6 Email Functionality via email_utils.py

The app uses **Gmail SMTP** to send OTPs and order receipts securely. This is centralized in utils/email_utils.py**.**

Highlights:

- Sends plain-text email with smtplib.
- Uses app configuration (MAIL_USERNAME, MAIL_PASSWORD) for secure login.
- Called by registration and order confirmation routes.

Fig 7 – Email definition in email_utils.py



Session and Security

- **Flask-Login** is used for managing user sessions with cookies.
- OTP verification ensures verified user accounts.
- CORS is set to supports_credentials=True to maintain session across frontend/backend.
- Passwords are hashed using **Flask-Bcrypt** before storing in the database.

4.3 Frontend Architecture Angular

The frontend of the Stock Control System is built using Angular 19, a modern and modular JavaScript framework known for its reactive design and two-way data binding. It implements a component-based architecture to support scalability, reusability, and maintainability. Each view or functional segment of the system is represented as a standalone Angular component, making the application highly modular and responsive.

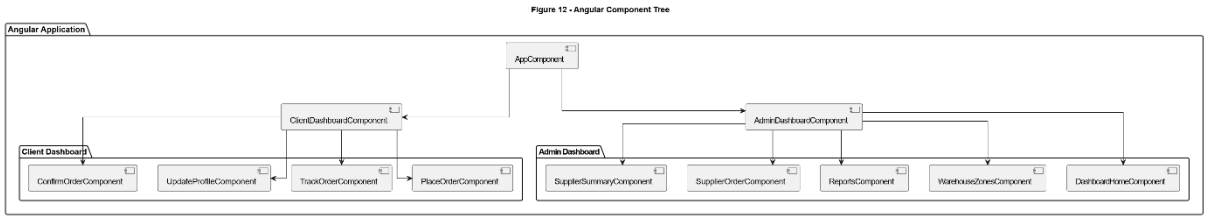
The application is structured with shared modules, services, and router configurations that manage navigation between pages. Angular's built-in support for TypeScript adds type safety and powerful tooling features during development.

Key architectural elements include:

- **Component-based structure:** The system uses components such as ClientDashboardComponent, ConfirmOrderComponent, TrackOrderComponent, and AdminDashboardComponent. Each encapsulates its own logic, view, and styling.
- **Service-based data flow:** Dedicated services (e.g., OrderService, ProductService) handle API calls and business logic, promoting clean separation of concerns.
- **Routing:** Angular's RouterModule manages route-based navigation. Routes are defined centrally and are linked to corresponding components with support for child routes, lazy loading, and guards.
- **Forms and Validation:** Angular's Reactive Forms are used extensively to enforce field-level validation, including OTP, email, phone number formatting, and password matching logic.
- **Bootstrap & Apex Charts:** Bootstrap ensures responsive layout and styling, while ApexCharts is used for dynamic visualizations like sales prediction graphs and low-stock alerts.

This architecture enables seamless user experiences for both client and admin dashboards, while maintaining a clear separation between presentation and logic layers. The component hierarchy is illustrated below

Figure 8 Angular Component Tree



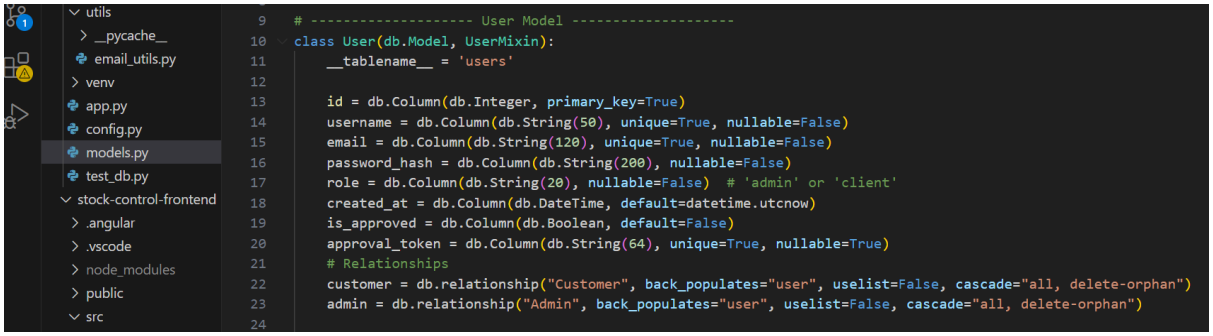
4.4 Database Design MySQL Schema

The database design for the Stock Control System is centered around normalization, data integrity, and clarity of relationships between core entities such as users, customers, orders, and products. The system uses **MySQL** as its relational database engine, and **SQLAlchemy ORM** in Flask to define and interact with the database schema.

All data models are **defined in models.py**, which establishes the tables, columns, data types, constraints, and inter-table relationships. This section describes the key models and their roles in the system.

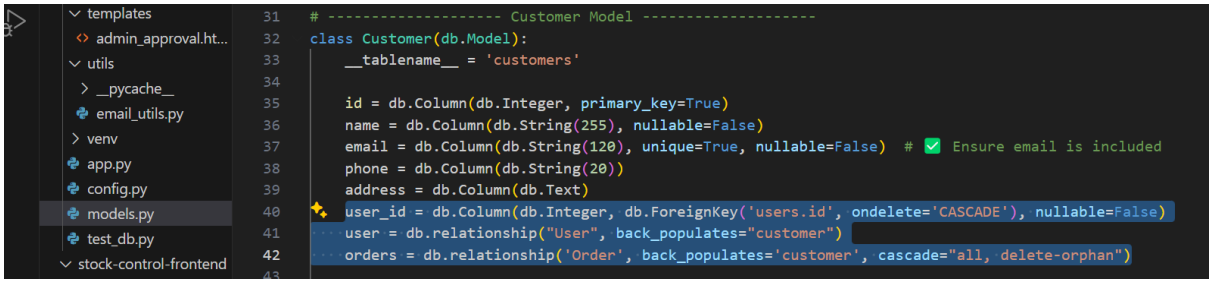
The **User** model is the central entry point for any authenticated actor in the system, including both customers and administrators. It includes core identity attributes such as username, email, hashed password, and role. The design **uses one-to-one relationships** to map each user to either a **Customer** or **Admin** profile, depending on their role. Passwords are securely hashed using Flask-Bcrypt, ensuring no plain text credentials are stored.

Fig 9 – User model definition depicting One to One relationship



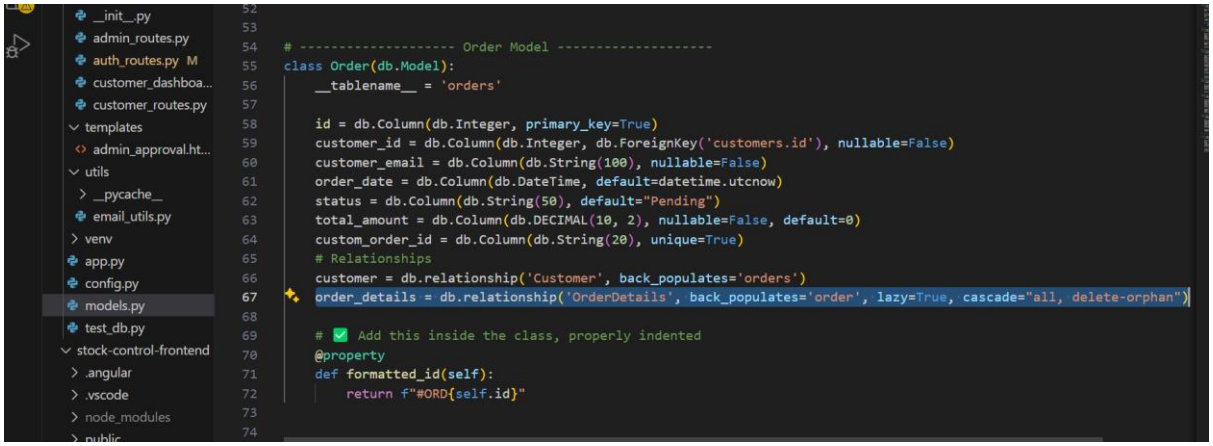
The **Customer** model contains additional personal and contact information, such as phone number and address. It includes a foreign key to the users table, forming a one-to-one relationship with the **User** model. Each customer may place multiple orders, which establishes a **one-to-many** relationship **between Customer and Order**.

Fig 10 – One to Many Relationship between Customer and Order



The **Order** model captures every transaction made by a customer. It includes a foreign key reference to the customers table and stores metadata such as the total amount, order date, status, and a custom-formatted order ID. This model also uses a **back-reference** to the **Customer model**, enabling easy tracking of order history.

Fig 11 – Back-reference declaration in Order table to Customer table

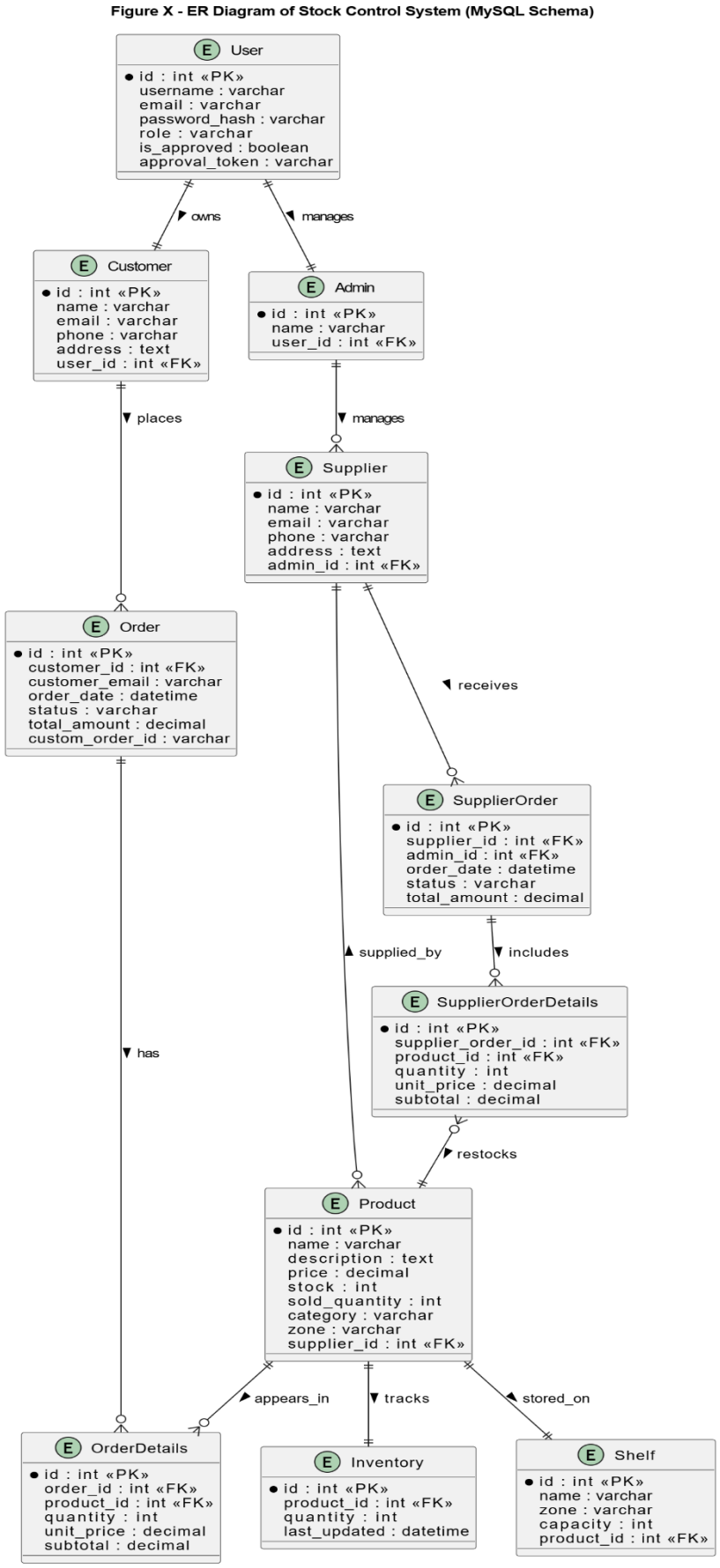


Each Order is linked to one or more products through the **OrderDetails** model, forming a **many-to-one relationship** with both **Order** and **Product**. This design enables detailed tracking of which products were sold in what quantity and at what price within each order.

The system ensures referential integrity by defining foreign keys with cascading delete rules. For instance, deleting a customer will also delete all their orders and related order details, which prevents orphaned records and maintains a clean database state.

The overall schema follows third normal form (3NF) to eliminate redundancy and ensure each table serves a single purpose. Each model includes appropriate data types, uniqueness constraints (e.g., on email and usernames), and default values (e.g., for order status and timestamps). All these practices collectively support the robustness and scalability of the database layer within the system.

Fig 12 - ER Diagram of Stock Control System



The schema has been normalized and structured to follow third normal form (3NF), reducing redundancy while preserving the ability to perform complex transactional operations [26]. A total of ten core entities have been defined: User, Admin, Customer, Product, Order, OrderDetails, Supplier, Inventory, Shelf, and SupplierOrder.

4.4.1 Entity Relationships and Schema Design

Each database model is defined as a Python class using SQLAlchemy, with primary keys, foreign keys, and cascading behaviors enforced to preserve referential integrity [27]. The diagram in Figure 5 (ER Diagram) provides a visual overview of the schema relationships, while the subsequent sections present sample table definitions for better understanding.

1. Admin and Customer

Every user in the system is registered through the User table. This includes both customers and admins, differentiated by the role attribute. A one-to-one relationship exists between User and Customer, and between User and Admin. This design ensures centralized login management while allowing role-specific extensions.

2. Customer, Order, and OrderDetails

Each customer can place multiple orders, creating a one-to-many relationship between Customer and Order. Each order, in turn, contains one or more line items in the OrderDetails table, which links each order to the specific Product purchased, along with quantity and pricing metadata. This structure allows precise sales tracking and invoicing.

3. Product and Inventory Control

The Product table includes product metadata such as name, price, and zone. Each product belongs to a Supplier, has a one-to-one link to the Inventory table for tracking quantity, and is assigned to a Shelf. This relational model allows administrators to map physical storage with digital tracking, facilitating real-time inventory updates [28].

4. Supplier, SupplierOrder, and SupplierOrderDetails

Suppliers are linked to specific admins via a foreign key in the Supplier model. When stock levels fall below a threshold, admins place replenishment orders through the SupplierOrder model. This model captures the supplier, admin, order date, status, and total cost. Detailed items in the order are stored in the SupplierOrderDetails table, maintaining traceability of stock purchases and restocking operations.

5. Shelf and Zone Allocation

Shelves are physical inventory locations defined with name, zone, and a fixed capacity. Each shelf is assigned to a Product, and constraints are enforced in the application logic to prevent overstocking beyond the defined limit. Shelf-level visualization is integrated into the admin dashboard for real-time warehouse layout awareness.

Table 4 - Table Schemas

Field Name	Type	Description
id	INT, PK	Unique product ID
name	VARCHAR(255)	Name of the product
description	TEXT	Product details
price	DECIMAL(10,2)	Unit price
stock	INT	Current stock count
sold_quantity	INT	Number of items sold
supplier_id	INT, FK	Links to Supplier.id
category	VARCHAR(255)	Product category
zone	VARCHAR(255)	Storage zone in warehouse

4.5 Email Integration SMTP

Email integration plays a central role in the system by facilitating secure and automated communication for critical events such as user registration, order confirmations, and inventory alerts. The following subsections explain the design, implementation, and challenges encountered.

SMTP-Based Communication Architecture

The system uses the **Simple Mail Transfer Protocol (SMTP)** to manage all outbound transactional emails. Python's built-in smtplib library, in conjunction with the email.mime package, is used to create and send structured messages [23]. The application currently sends:

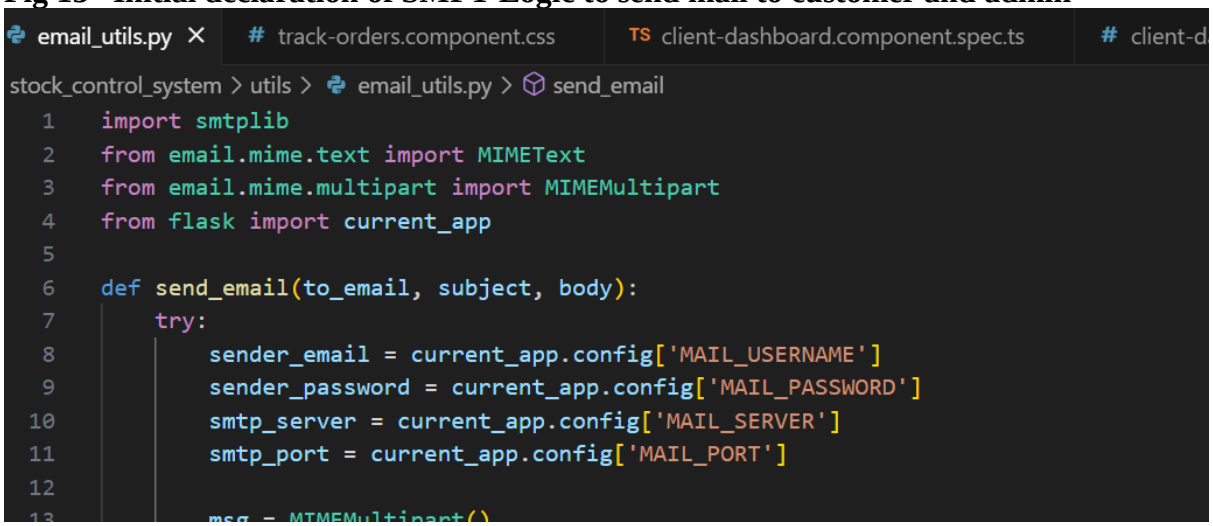
- OTPs during customer registration,
- Order receipts after purchase confirmation,
- Low-stock notifications to administrators.

This setup ensures that essential communication occurs in real time without manual intervention.

Secure Configuration Using Flask current_app.config

To safeguard email credentials and support flexible deployment environments, SMTP settings (e.g., server address, port, sender email, app password) are stored in Flask's current_app.config. This approach keeps sensitive data out of the core logic, improving both **security** and **maintainability** [24].

Fig 13– Initial declaration of SMPT Logic to send mail to customer and admin



TLS encryption is enabled by default when connecting to Gmail's SMTP server, ensuring secure transmission of messages over the network.

Email Composition Using MIME Format

Emails are structured using the **MIMEMultipart** class to allow for multi-part formatting. The current implementation uses **plain-text bodies** for broad compatibility across different email clients and devices. The design also leaves room for extensibility—future versions could support:

- PDF attachments (e.g., invoices),
- HTML-formatted emails for improved styling.

Handling CORS Errors and Temporary OTP Storage in Server Session

During the initial implementation of the OTP verification feature, a CORS (Cross-Origin Resource Sharing) error occurred when sending requests from the Angular frontend (localhost:4200) to the Flask backend. The error blocked communication between the frontend and backend, especially when the OTP was expected to be returned and verified across two separate requests.

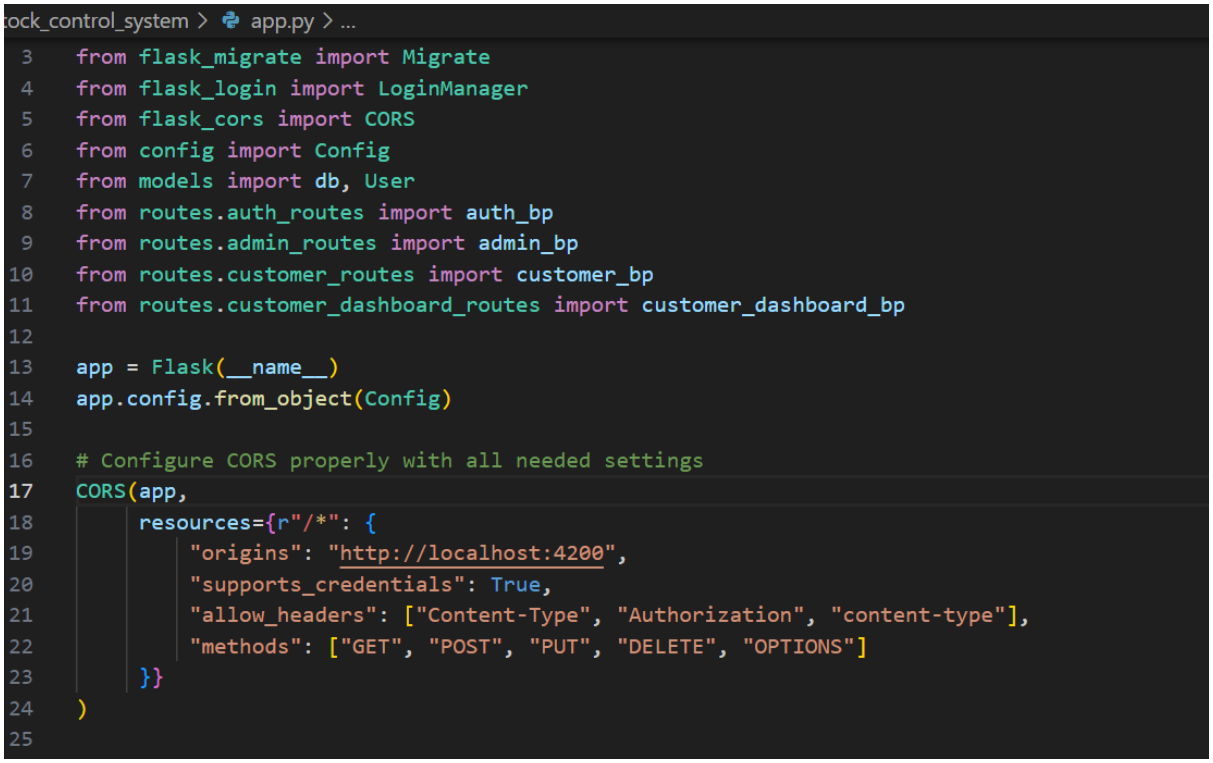
To resolve this, server-side session storage was temporarily used to store the generated OTP. When the /send_otp API was called, the OTP was:

- Generated on the server,
- Sent via email to the user using Flask's SMTP integration,
- And stored in the Flask session object (session['otp']).

This allowed the OTP to persist securely across requests during the registration flow. When the user submitted the OTP via the verification form, it was compared against the server-stored value for validation.

By enabling supports_credentials=True in Flask-CORS configuration, and allowing credentials (cookies) on the Angular side with withCredentials: true, the server was able to maintain session state reliably between frontend and backend.

Fig 14 – CORS configuration in app.py



This approach ensured that OTP validation remained secure and stateless for the user while resolving CORS-related session discontinuity. Although more scalable token-based approaches exist (e.g., sending OTP tokens via JWT), this method was effective and appropriate for the scope of the project.

Table 5: Email Notification Scenarios

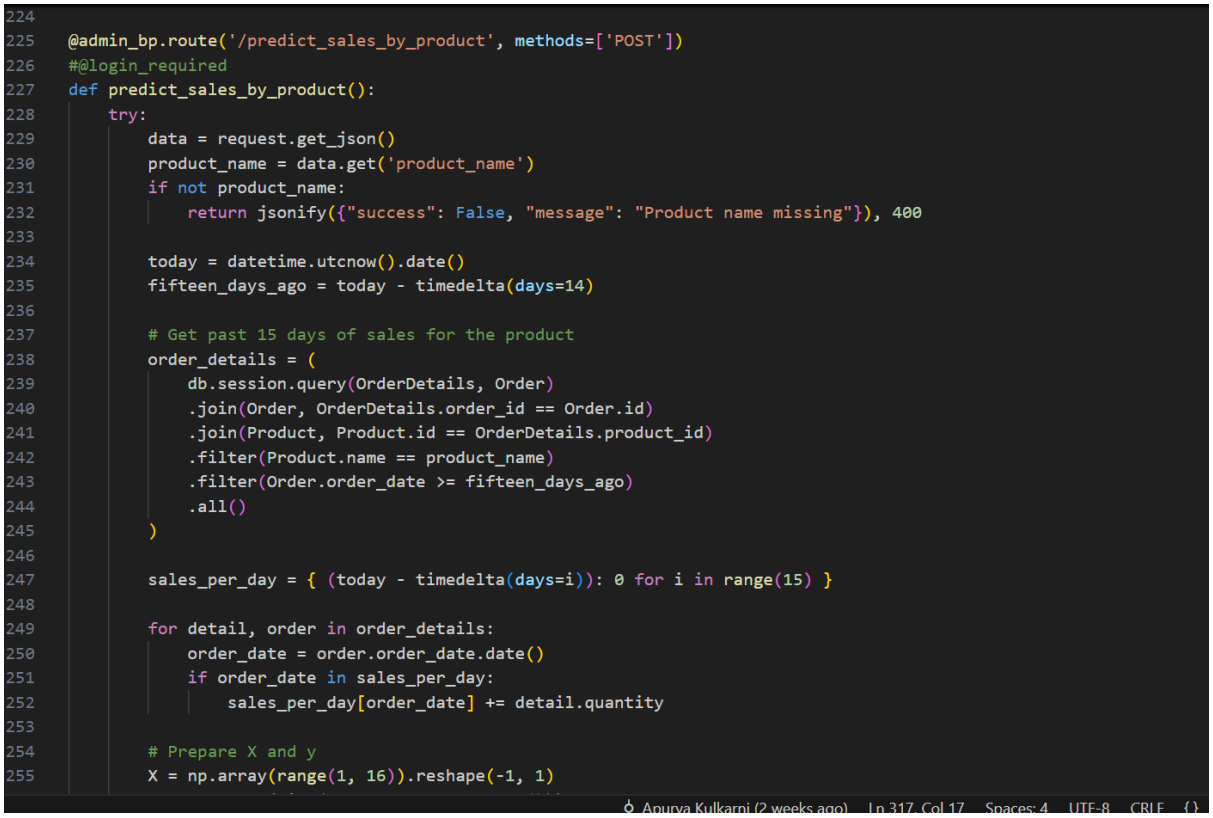
Purpose	Trigger Source	Recipient	Email Content
OTP Verification	User registration (/auth/send_otp)	Customer	6-digit OTP with 5-minute expiry
Order Receipt	Order confirmed (/send_order_receipt)	Customer	List of items, quantity, total amount
Low Stock Alert	Order confirmation (/confirm_order)	Admins	Product name and remaining stock quantity

4.6 Machine Learning Model Linear Regression for Forecasting

To enhance inventory planning and support proactive supplier ordering, a supervised machine learning model was implemented within the Stock Control System. The model predicts future product demand by analyzing the sales patterns of individual items over the past fifteen days and estimating expected sales for the next thirty days. This forecasted data assists administrators in identifying trends, avoiding understocking, and preparing timely supplier orders.

The prediction system is embedded in the Flask backend under the **/admin/predict_sales_by_product** route. Upon receiving a product name from the frontend, the system fetches the corresponding sales history by joining order and product tables in the database. The past fifteen days of order records are aggregated into a daily time series. Each date is converted into a numeric input variable, and the number of units sold on that date becomes the corresponding output. These values are used to train a Linear Regression model built with the Scikit-learn library. The trained model then generates a sales forecast for the following thirty days. The predicted values are rounded, ensured to be non-negative, and returned as a structured JSON response. On the admin dashboard, ApexCharts is used to visually plot these values in an interactive graph.

Fig 15 – Code to predict sales by product using Linear Regression



```
224
225 @admin_bp.route('/predict_sales_by_product', methods=['POST'])
226 #@login_required
227 def predict_sales_by_product():
228     try:
229         data = request.get_json()
230         product_name = data.get('product_name')
231         if not product_name:
232             return jsonify({"success": False, "message": "Product name missing"}), 400
233
234         today = datetime.utcnow().date()
235         fifteen_days_ago = today - timedelta(days=14)
236
237         # Get past 15 days of sales for the product
238         order_details = (
239             db.session.query(OrderDetails, Order)
240             .join(Order, OrderDetails.order_id == Order.id)
241             .join(Product, Product.id == OrderDetails.product_id)
242             .filter(Product.name == product_name)
243             .filter(Order.order_date >= fifteen_days_ago)
244             .all()
245         )
246
247         sales_per_day = { (today - timedelta(days=i)): 0 for i in range(15) }
248
249         for detail, order in order_details:
250             order_date = order.order_date.date()
251             if order_date in sales_per_day:
252                 sales_per_day[order_date] += detail.quantity
253
254         # Prepare X and y
255         X = np.array(range(1, 16)).reshape(-1, 1)
```

```
        # Prepare X and y
        X = np.array(range(1, 16)).reshape(-1, 1)
        y = np.array(list(sales_per_day.values()))

        model = LinearRegression()
        model.fit(X, y)

        # Predict next 90 days
        future_days = np.array(range(16, 46)).reshape(-1, 1)
        predictions = model.predict(future_days)
        predictions = [max(0, int(round(p))) for p in predictions]

        future_sales = [{"day": i + 1, "quantity": qty} for i, qty in enumerate(predictions)]

        return jsonify({
            "success": True,
            "product": product_name,
            "predicted_sales": future_sales
        }), 200

    except Exception as e:
        return jsonify({"success": False, "error": str(e)}), 500
```

Linear Regression was selected for its simplicity, transparency, and suitability for small datasets. The project deals with short time windows and limited historical data, making advanced models like Random Forest or Neural Networks unnecessarily complex and resource-intensive. Linear Regression offers a lightweight, interpretable solution that integrates easily with the Flask architecture and runs on-demand without requiring continuous background processing [20]. Because the model operates asynchronously and does not affect transactional performance, it fits well into the system's modular and scalable design [25].

Table 6 – Comparison between Linear Regression and other Algorithms

Criteria	Linear Regression	Complex Models (e.g., Random Forest, ANN)
Data Requirement	Performs well with small datasets	Require large volumes of training data
Interpretability	High – easy to explain and visualize	Low – often considered a black box
Training Time	Fast and lightweight	Slower, more resource-intensive
Deployment Complexity	Easy to integrate into Flask API	Requires tuning, libraries, and more dependencies
Suitability for Scope	Ideal for academic or lightweight systems	Overkill for short-term, low-volume forecasting

This implementation delivers practical forecasting capability within the constraints of a local, resource-conscious system. Although it does not incorporate external variables like seasonal trends or pricing changes, it effectively addresses the need for short-term demand prediction and provides a foundation for more advanced analytics in future iterations [25].

Understanding the Best-Fit Line using Linear Regression Intuition

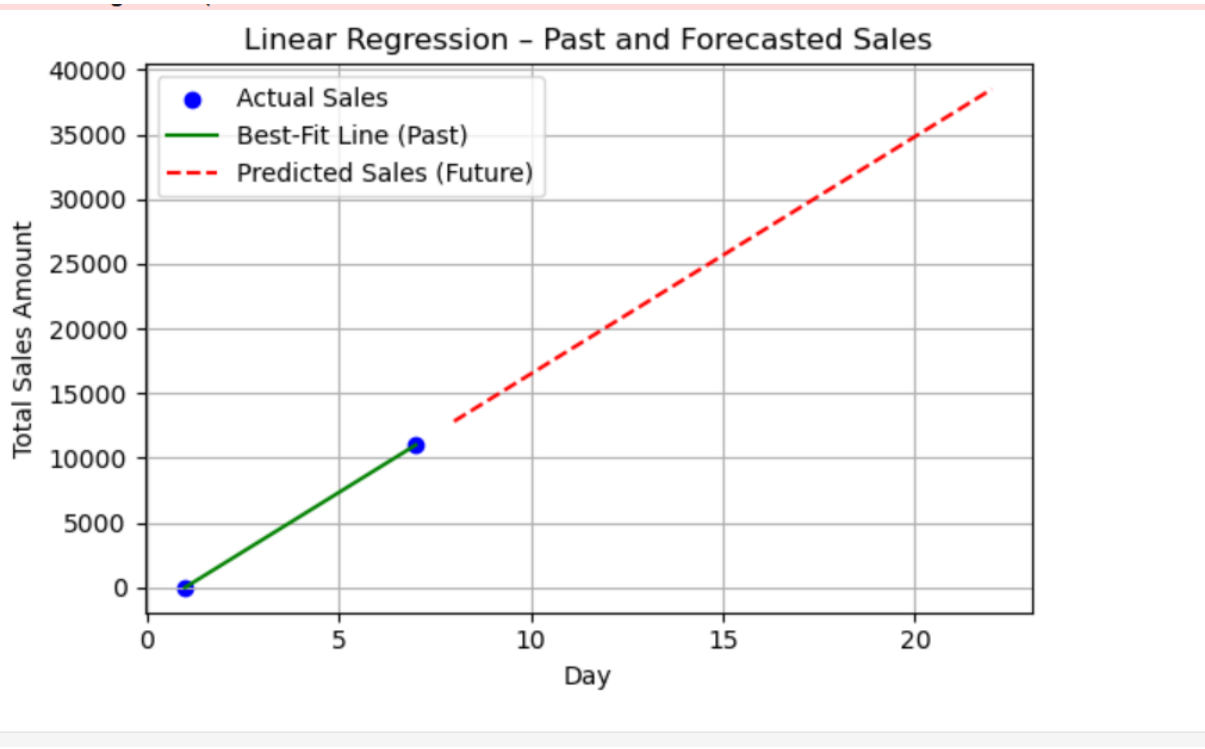
To enhance inventory planning and support proactive supplier ordering, a supervised machine learning model was implemented within the Stock Control System. The model forecasts future product demand by analyzing sales patterns of individual items over the past fifteen days and estimating their likely sales over the next thirty days. These predictions enable administrators to identify declining trends early and generate timely supplier orders to avoid understocking.

Linear regression attempts to draw a straight line the best-fit line that minimizes the total squared difference between actual sales values and the predicted values on that line. In other words, it answers the question:

Given how many units we sold on each day recently, what is the likely number we'll sell on future days if this trend continues?

This is useful in warehouse settings where decisions need to be made based on short-term trends.

Fig 16 – Best Fit Line Representation of my Order sales data



4.7 Technical Challenges and Solutions

1. Setting Up One-to-One Relationships with SQLAlchemy

Challenge:
Implementing precise one-to-one relationships between User–Customer and User–Admin models using SQLAlchemy proved tricky. Initial attempts caused duplicate entries or circular foreign key conflicts.
Solution:
The problem was resolved by using uselist=False in the db.relationship() declaration and applying cascade="all, delete-orphan" to ensure proper deletion of child records. The relationships were also clearly named (customer, admin) and back-populated using back_populates for bidirectional linking.

2. Email OTP Functionality via Flask Mail

Challenge:
During user registration, integrating a secure and functional OTP system with Gmail SMTP often led to failures due to port misconfiguration, timeout errors, and Gmail's less secure app restrictions.

Solution:
The Flask-Mail configuration was carefully adjusted with correct port (587), TLS enabled, and Gmail account settings updated to allow SMTP access. The OTP logic was tested with invalid/expired tokens to ensure secure gating of password creation.

3. CORS and Cookie Handling in Angular–Flask Communication

Challenge:
Cross-origin requests from Angular frontend to Flask backend often failed when session cookies were not sent correctly, especially on protected routes like /place_order.

Solution:
CORS was configured using Flask-CORS with supports_credentials=True, and all Angular HttpClient calls used { withCredentials: true }. Backend routes were also adjusted to explicitly allow credentials. This allowed session-based login to persist across frontend calls.

4. Pytest Configuration with Flask App Factory

Challenge:
Unit testing using pytest raised ModuleNotFoundError: No module named 'app' due to misconfiguration in conftest.py and improper project structure.

Solution:
The PYTHONPATH was correctly set, and the Flask app import in conftest.py was adjusted to use relative imports. In-memory SQLite was configured (sqlite:///:memory:) to avoid corrupting live data, and pytest-flask was installed to support client fixtures.

5. Sales Forecasting Integration using Linear Regression

Challenge:
Integrating a machine learning model for 90-day sales prediction introduced compatibility issues between Python libraries and the Flask API response structure.

Solution:
The model was trained using scikit-learn, and predictions were returned as structured JSON. In Angular, ApexCharts was used to render prediction results dynamically, with data mapped to a time-series line graph.

Chapter 5

Implementation and Development Process

The development of the Stock Control System followed a modular, component-based architecture, prioritizing maintainability, scalability, and clarity in both backend and frontend implementations [15]. The system was built incrementally using Agile-inspired iterative cycles, allowing individual features to be tested and refined as they were integrated [29]. This chapter documents the step-by-step implementation process of the core functionalities, including backend services developed with Flask, the Angular-based frontend interface, the integration of machine learning for sales forecasting, and secure email-based features for system alerts and communication.

Best practices were followed throughout the codebase, including RESTful API design in Flask [27], the introduction of Standalone Components represents a significant evolution in Angular's architecture, aimed at simplifying development, improving performance, and enhancing scalability. [30] SQLAlchemy ORM was used to map backend models to MySQL database tables [27], and structured exception handling was implemented to maintain system stability. Integration between layers was facilitated through clear API contracts and shared data schemas, ensuring consistent and predictable behavior during runtime. Testing and bug resolution were incorporated as part of the development cycle, with feedback loops used to identify and address functional and UI issues early in the pipeline [29].

5.1 Backend Implementation of Flask Services

The backend of the Stock Control System was developed using the Flask microframework due to its flexibility, simplicity, and extensive community support [27]. The application adopts a modular structure using Blueprints (auth_bp, admin_bp, customer_bp, customer_dashboard_bp) to separate concerns and improve maintainability. All configurations are stored centrally in config.py, which defines database URI, mail server settings, and security tokens [31].

The main application logic is initialized in app.py, where CORS settings are configured to support cross-origin communication with the Angular frontend. SQLAlchemy is used for ORM-based interaction with the MySQL database, and Flask-Migrate handles schema changes through versioned migrations [32]. The app is started using:

Fig 17 - CORS Configuration

```
# Configure CORS properly with all needed settings
CORS(app,
     resources={r"/*": {
         "origins": "http://localhost:4200",
         "supports_credentials": True,
         "allow_headers": ["Content-Type", "Authorization", "content-type"],
         "methods": ["GET", "POST", "PUT", "DELETE", "OPTIONS"]
     }}
)

# Initialize database
db.init_app(app)
migrate = Migrate(app, db)
```

Secure user authentication is achieved using Flask-Login and password hashing via bcrypt [27]. Passwords are hashed and verified as follows:

Fig 18 – Encryption of password using bcrypt

```
    def set_password(self, password):
        self.password_hash = bcrypt.generate_password_hash(password).decode('utf-8')

    def check_password(self, password):
        return bcrypt.check_password_hash(self.password_hash, password)
```

5.2 Frontend Implementation of Angular Components

The frontend of the Stock Control System was developed using Angular 16+ as a standalone component-based architecture, ensuring clean routing, high reusability, and responsiveness. The **client-dashboard.component.ts** and **confirm-order.component.ts** serve as the primary interfaces for customer operations like browsing products, managing the cart, and confirming purchases. Angular's reactive component model and seamless integration with Bootstrap CSS enhanced both the usability and maintainability of the interface [33].

The **Client Dashboard** (**client-dashboard.component.ts**) loads all available products from the Flask backend using the **/customer_dashboard/products** API. Users can browse, filter (by stock status, price, or sales volume), and add products to a virtual cart. The state of each item's quantity is tracked using the **orderQuantities** object and validated dynamically with stock constraints.

Fig 19 – Filtering products on client dashboard code logic

```
getStockLabel(stock: number): string {
  if (stock === 0) return 'Out of Stock';
  if (stock <= 10) return `Low Stock: ${stock} left`;
  return `In Stock: ${stock} available`;
}

validateQuantity(productId: number, stock: number): void {
  const quantity = this.orderQuantities[productId];
  this.orderErrors[productId] = '';

  if (isNaN(quantity) || quantity <= 0) {
    this.orderErrors[productId] = 'Please enter a valid quantity';
    this.orderQuantities[productId] = 1;
    return;
  }

  if (quantity > stock) {
    this.orderErrors[productId] = `Only ${stock} items available`;
    this.orderQuantities[productId] = stock;
    return;
  }
```

The above method ensures the cart respects stock limitations by performing client-side validation before placing an order [34]. Toasts and dropdowns are handled using Angular event bindings and Bootstrap utilities for responsive design.

Once items are added, the **Confirm Order** component (confirm-order.component.ts) is loaded using Router.navigate() and navigation state. This component calls /customer_dashboard/**confirm_order** to finalize the order and receives back the order ID, items, and total cost, which are then rendered on screen.

Fig 20 – Confirm order API

```
  this.http.post<any>('http://127.0.0.1:5000/customer_dashboard/confirm_order', {
    email: this.email,
    items: this.selectedItems.map(item => ({
      product_id: item.id,
      quantity: item.quantity
    }))
  }).subscribe(response => {
    if (response.success) {
      this.totalPrice = response.total_price;
      this.orderId = response.order_id;  // TEMP ID
      this.selectedItems = response.items.map((item: any) => ({
        id: item.id,
        name: item.name,
        price: item.price,
        quantity: item.quantity,
        subtotal: item.subtotal,   // ✅ USE backend subtotal directly
        image: item.image
      }));
    }
  });
}
```

This API integration ensures consistent data syncing with the backend [27]. The user can then trigger the payNow() function which calls /customer_dashboard/send_order_receipt, sending an email confirmation using backend SMTP utilities. Each product image is resolved dynamically using the filename pattern from static assets (assets/productName.png), enabling clean visual representation [35].

Routing is configured in app.routes.ts, using Angular's RouterModule.forRoot() structure. Admin and customer dashboards are separated via route guards and layouts. The dashboard pages are designed as standalone components, which reduces coupling and improves lazy loading capability [33].

Fig 21 - Angular Component Tree

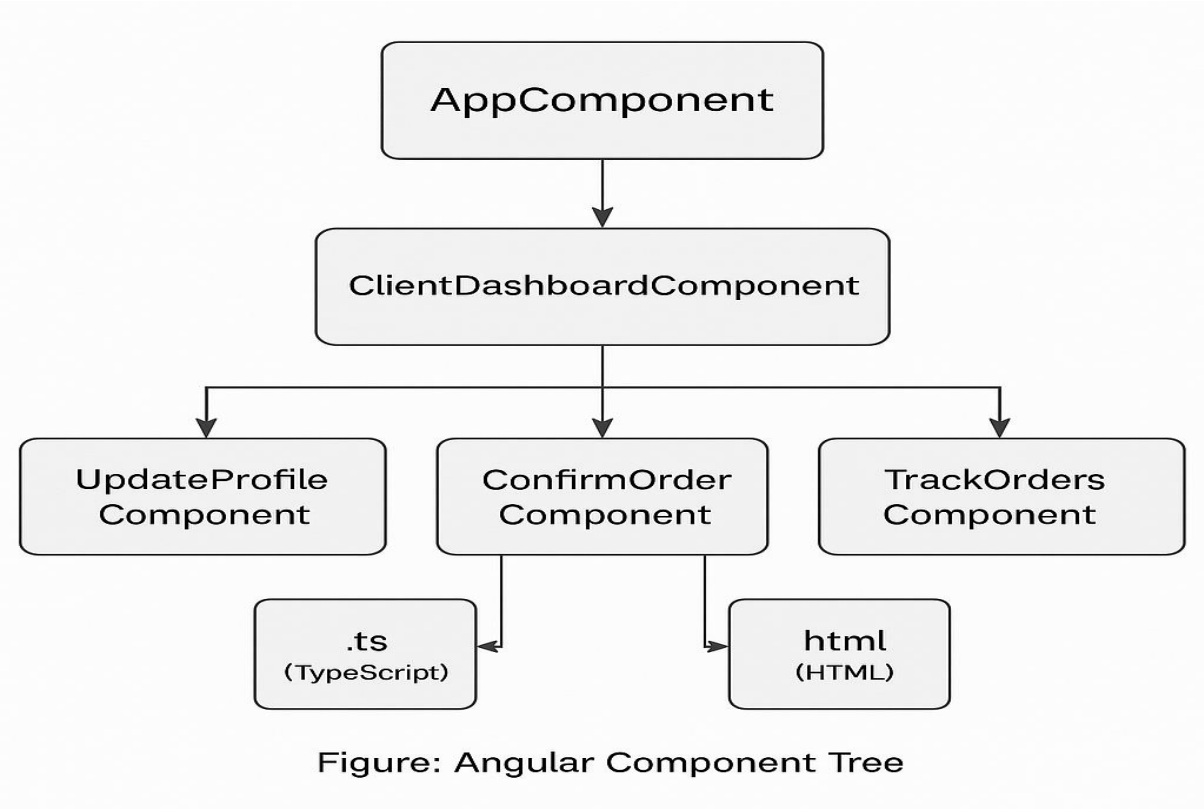


Figure: Angular Component Tree

5.3 Integration of Machine Learning Model

The Stock Control System incorporates a machine learning-based sales prediction feature that aids administrators in making proactive stock replenishment decisions. This integration is built using a Linear Regression model from scikit-learn in the Flask backend and visualized via ApexCharts in the Angular admin dashboard [36].

The integration begins at the Flask API **/predict_sales_by_product**, which is triggered when an admin selects a product. The endpoint fetches the last 15 days of real order quantities from the database, trains a regression model, and returns 30 future day predictions:

Fig 22 – API logic of Predict sales by products

```python
@admin_bp.route('/predict_sales_by_product', methods=['POST'])
def predict_sales_by_product():
    try:
        data = request.get_json()
        product_name = data.get('product_name')

        if not product_name:
            return jsonify({"success": False, "message": "Product name missing"}), 400

        # Step 1: Get product ID
        product = Product.query.filter(Product.name.ilike(product_name)).first()
        if not product:
            return jsonify({"success": False, "message": "Product not found"}), 404

        product_id = product.id

        today = datetime.utcnow().date()
        thirty_days_ago = today - timedelta(days=29)
```

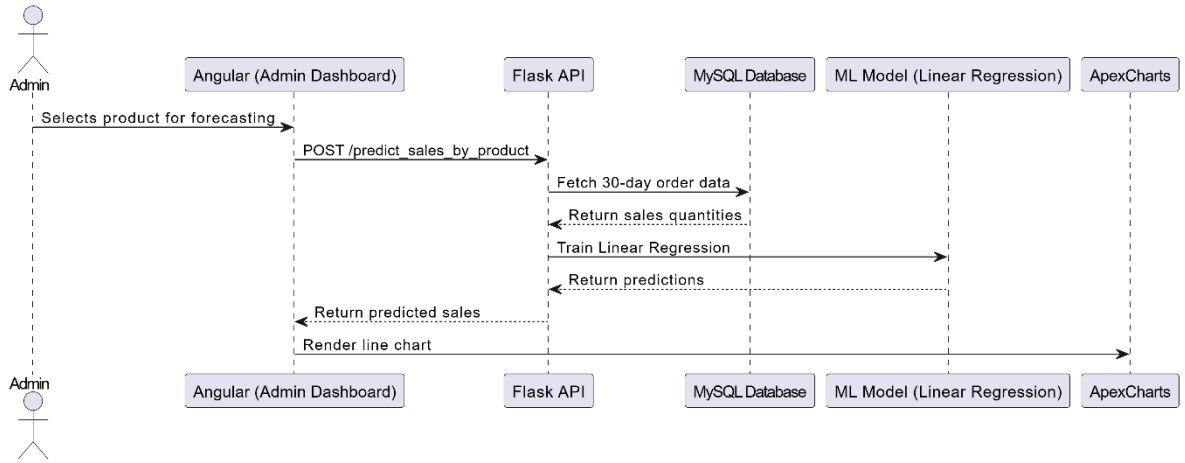
This ensures lightweight predictive capabilities without requiring cloud-based ML or GPU support [37].

On the frontend, dashboard-home.component.ts in the Angular admin panel makes a POST request with the selected product name, retrieves the prediction list, and maps it into chart-compatible series data.

Fig 23 - dashboard-home.component.ts

```typescript
fetchPredictionByProduct(productName: string) {
  if (!productName) return;

  const url = 'http://127.0.0.1:5000/admin/predict_sales_by_product';

  this.http.post<any>(url, { product_name: productName }, { withCredentials: true }).subscribe({
    next: (response) => {
      console.log("✅ Backend Response:", response); // debug output

      // 🔥 Fix: Change predictions → predicted_sales
      if (response.success && response.predicted_sales?.length) {
        this.futureSalesByProductSeries = [{
          name: productName,
          data: response.predicted_sales.map((p: any) => p.quantity) // quantity only
        }];
```

Fig 24 - Integration Workflow Diagram



This integration supports predictive inventory management, allowing the system to anticipate low stock trends and reduce order delays. By using linear regression and focusing on a minimal yet useful data model, the implementation balances performance and simplicity for academic scope [36][38].

5.4 Email Functionality Implementation having OTP + email Receipts

To enhance user communication and transaction integrity, the Stock Control System includes robust email features powered by Flask's SMTP integration. Two key functionalities are supported: (1) sending OTPs during registration and (2) delivering order confirmation receipts post-purchase. These features were implemented using the Gmail SMTP service, configured securely through environment-based credentials in config.py.

OTP Verification During Registration
Upon entering their email on the registration form, users receive a 6-digit OTP generated using Python's random.randint() method. The backend route /customer_dashboard/send_otp handles this, composing and dispatching the OTP via the **send_email()** utility. The send_email() utility internally uses Python's built-in smtplib library [41] to establish a secure connection with Gmail's SMTP server and deliver the email.

Email delivery uses TLS-encrypted SMTP over Gmail [39]. Here is the real code from **customer_dashboard_routes.py:**

Fig 25 – send_otp API

```python
@customer_dashboard_bp.route('/send_otp', methods=['POST'])
def send_otp():
    data = request.get_json()
    email = data.get('email')

    if not email:
        return jsonify({'success': False, 'message': 'Email is required'}), 400

    otp = str(random.randint(100000, 999999))

    # Send OTP via Email
    subject = "🔐 Your OTP for Registration"
    body = f"""
Hi there,

Your One-Time Password (OTP) for completing your registration is: {otp}

This OTP is valid for 5 minutes.
```

If successful, the OTP is returned in the response for frontend validation (not persisted in the database). This protects users from unauthorized registrations while avoiding unnecessary storage of sensitive tokens.

Order Receipt Email Post Confirmation

Once a customer places an order on the Confirm Order page **confirm-order.component.ts,** a POST request is made to /**customer_dashboard/send_order_receipt** with order ID, items, and email. The backend prepares a readable summary, e.g.,

Fig 26 – send_order_receipt email to customer API

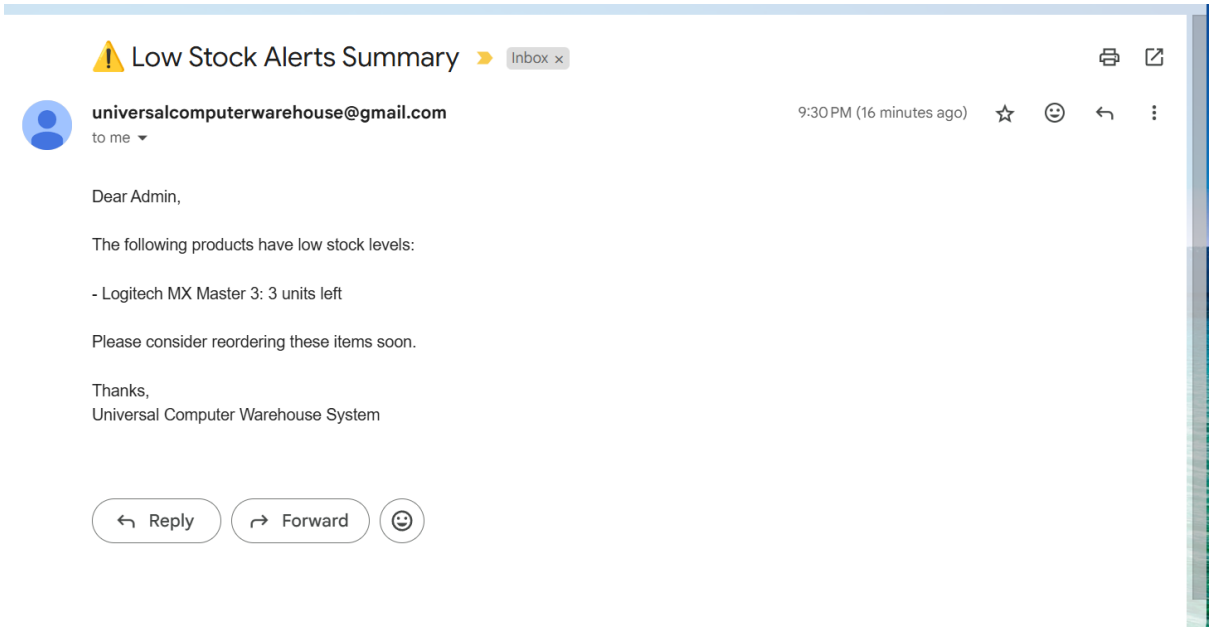
```python
@customer_dashboard_bp.route('/send_order_receipt', methods=['POST'])
def send_order_receipt():
    try:
        data = request.json
        email = data.get('email')
        items = data.get('items', [])
        total_price = data.get('total_price')
        order_id = data.get('order_id')
        if not email or not items or total_price is None:
            return jsonify({"success": False, "message": "Incomplete data"}), 400

        # Prepare email content
        item_lines = [f"- {item['name']} (Qty: {item['quantity']})" for item in items]
        item_text = "\n".join(item_lines)
        subject = "🧾 Order Confirmation - Thank You for Your Purchase!"
        body = f"""
Hi there,

Thank you for your order! 🎉
```

The same send_email() utility dispatches this receipt. For admin users, if low stock is detected during order placement, a separate alert email is sent using this model:

Fig 27 – Low Stock email triggered to admin function

```
        # ✅ AFTER committing all changes, now send a single email if needed
        if low_stock_alerts:
            subject = "⚠️ Low Stock Alerts Summary"
            body = f"""
Dear Admin,

The following products have low stock levels:

{chr(10).join(low_stock_alerts)}

Please consider reordering these items soon.

Thanks,
Universal Computer Warehouse System
"""
            admin_users = User.query.filter_by(role="admin").all()
            admin_emails = [admin.email for admin in admin_users]

```

Fig 28 - LOW STOCK email alert to Admin



Security & Configurations

SMTP credentials are abstracted through environment variables in config.py.

Fig 29 – SMTP setup of warehouse email the one which sends otp and receipt to clients

```
# ✅ Add these for email config
MAIL_SERVER = 'smtp.gmail.com'
MAIL_PORT = 587
MAIL_USE_TLS = True
MAIL_USERNAME = 'universalcomputerwarehouse@gmail.com'
```

Plain text passwords are avoided in production via .env and os.getenv() (recommended for deployment). The send_email() function uses smtplib and TLS encryption to ensure email delivery is secure.

Frontend Integration

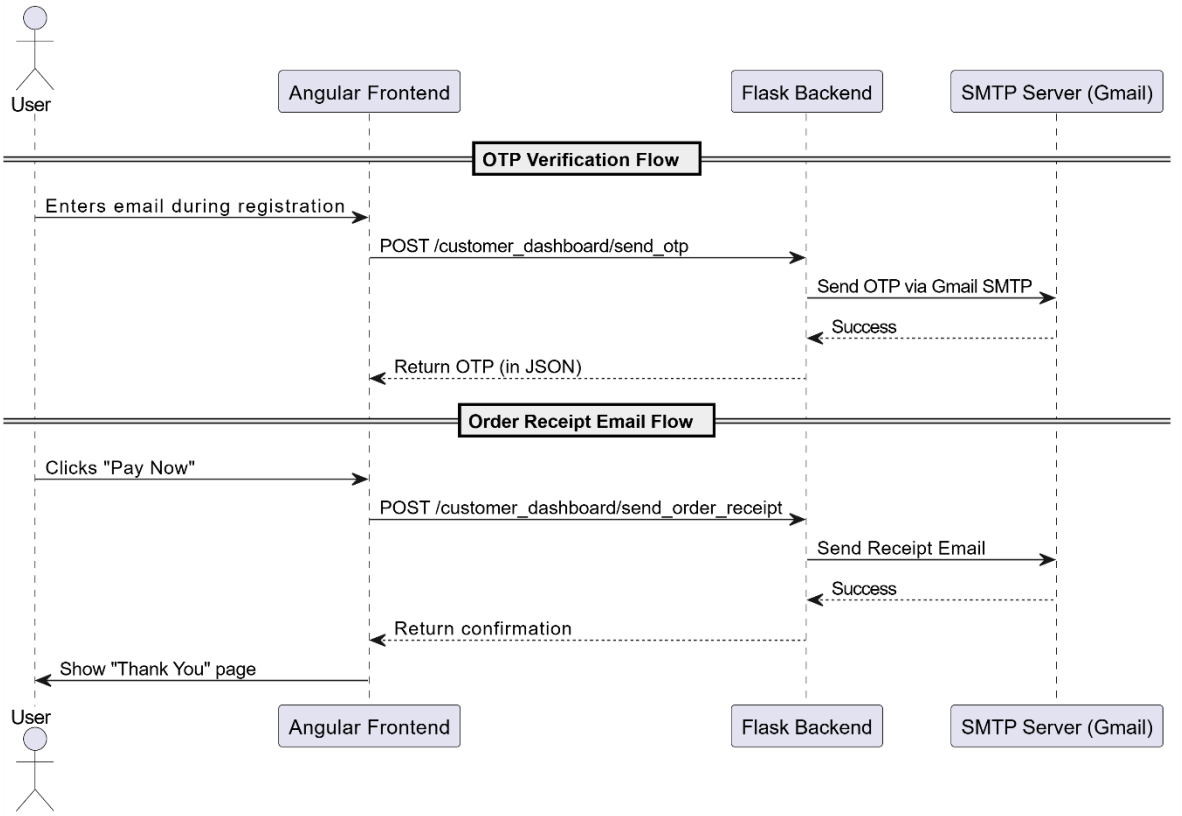
On the Angular frontend uses HttpClient.post() [40] the OTP flow is handled by a POST request to **/send_otp**, and on receipt, the OTP is validated client-side before registration proceeds. For receipts, **payNow()** in confirm-order.component.ts triggers the following:

Fig 30- send_order_receipt triggering email to customer logic

```
62
63   payNow(): void {
64     this.http.post<any>('http://127.0.0.1:5000/customer_dashboard/send_order_receipt', {
65       email: this.email,
66       order_id: this.orderId,
67       total_price: this.totalPrice,
68       items: this.selectedItems.map(item => ({
69         name: item.name,
70         quantity: item.quantity
71       }))
```

If the response is successful, the app navigates to the thank-you page.

Fig 31- Sequence Diagram Code with Email and OTP + Receipt



This diagram clearly splits the **OTP** and **Order Confirmation** sequences, includes **Angular**, **Flask**, **SMTP**, and the **User**, and follows correct PlantUML sequence syntax.

5.5 Testing Strategies and Bug Fixes

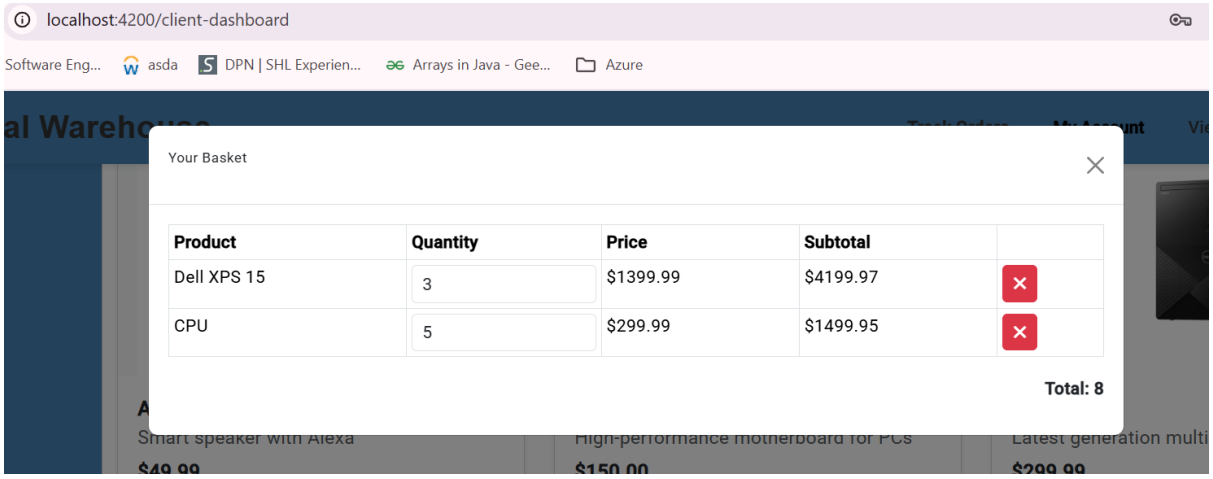
Robust testing was integral to ensuring a seamless user experience and stable performance of the Stock Control System. The frontend and backend were tested separately to validate logic flows, user actions, error messages, and system behavior under edge conditions.

Frontend Validation and UI Testing

On the Angular client side, interface-level validations were implemented and tested across all interactive pages. Filters such as "Low Stock" or "Price Range" on the Client Dashboard were checked using sample product data to verify correct sorting and dynamic updates. Quantity selection fields incorporated ngModel-based validation to ensure that users could not exceed the available stock. Error messages were shown beside each invalid input field using Angular's state tracking and dynamic bindings [43].

In the Angular frontend, logic errors were identified during UI rendering and interaction testing. Filters on the client dashboard (e.g., low stock, price range) were tested using mock data and user input. Form validation was implemented using ngModel and checked with incorrect quantity inputs to trigger error messages, ensuring a smooth customer experience [43]. The "View Basket" modal and confirmation flow were verified to ensure state persistence during routing.

Fig 32 - View Basket modal



Email Functionality Testing (OTP and Receipts)

The backend email system was tested by simulating customer actions such as registration and order completion. Flask's built-in smtplib module was used to trigger OTPs and order receipts to a designated test email account (universalcomputerwarehouse@gmail.com). Email failures caused by invalid credentials or SMTP errors were logged server-side, while corresponding alerts were triggered on the frontend to notify the user [39][41].

Fig 33 - EMAIL triggered to the customer after they click Pay

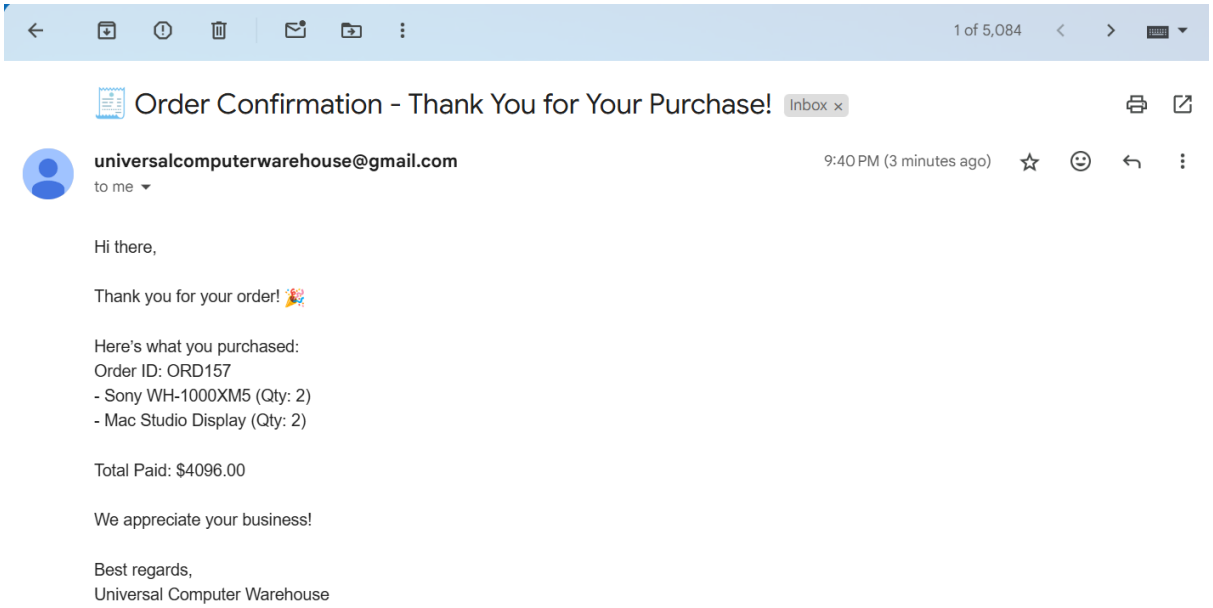
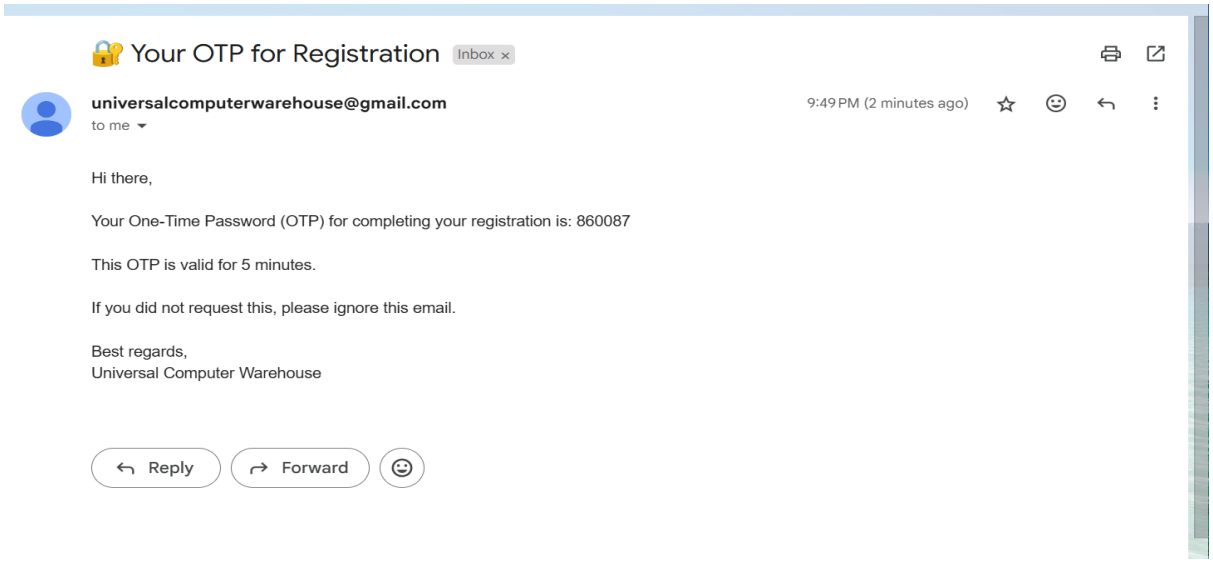


Fig 34 - OTP sent to customer for validation expiring after 5 mins



Key Bugs & Fixes

- **Bug**: Order status was not updating correctly in the track_orders component.
  Solution: Implemented dynamic status calculation based on the difference between the order date and the current date (Pending, Packaging, Delivered).
- **Bug:** Shelf capacity was being exceeded when placing supplier orders.
  Solution: Added logic to prevent over-ordering by showing **toast warnings**, tracking

cumulative quantity, and disabling the **'Add'** button once the shelf reaches full capacity.

- **Bug:** Invalid or excessive product quantities were being added to the cart.
  Solution: Input validation was implemented using Angular's state tracking and custom error messages next to quantity fields.
- **Bug:** Email receipt occasionally failed without clear feedback.
  Solution: Added error handling and fallback alerts to inform users if email sending failed post-payment.

Chapter 6

User Interface and Functionalities

The Stock Control System was designed with a user-centric approach, prioritising clarity, responsiveness, and seamless functionality across roles. Built using Angular for the frontend and integrated with a Flask backend, the interface adapts dynamically based on user type Admin or Client. Each role is granted access to dedicated dashboards tailored to their responsibilities, whether it's order placement and tracking for clients or inventory oversight and analytics for admins. This chapter details the structure, navigation, and behaviour of these interfaces, explaining how user actions are connected to real-time backend operations via RESTful APIs.

6.1 User Roles Overview

The Stock Control System distinguishes between two primary user roles: **Clients (Customers)** and **Admins**, each with their own distinct interface and privileges. This separation not only enhances usability but also enforces security boundaries through role-based access control [44].

Client Role

Clients access the system through a simplified interface focused on **product browsing, placing orders**, **tracking order status**, and **updating their profile**. Upon logging in, they are greeted with a dashboard displaying featured products, filters (e.g., low stock, best sellers), and a cart-based order mechanism refer to **Figure : Client Dashboard View** [45]. After selecting quantities, users proceed to a confirmation page and are sent a receipt via email post-payment.

Fig 35 - Client Dashboard View

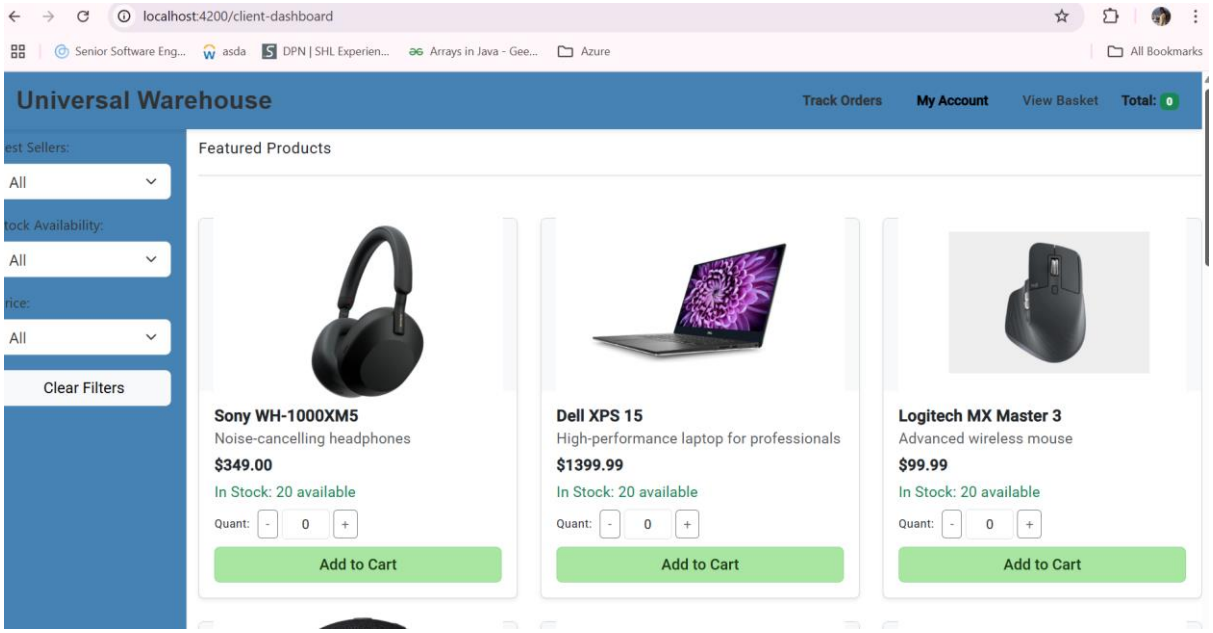
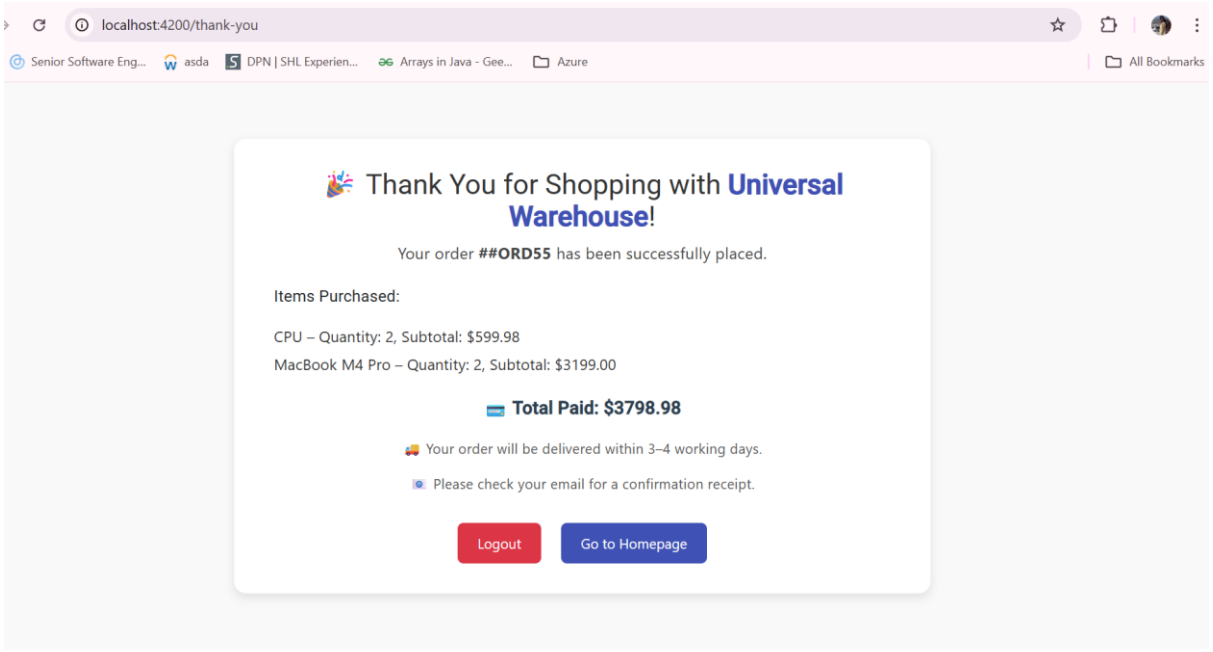


Fig 36 - Thank You page



Admin Role

Admins, on the other hand, gain access to a **multi-tabbed dashboard** which includes sections for **inventory control**, **warehouse zone management**, **supplier ordering**, **report generation**, and **sales forecasting**. This role is restricted to verified users and includes approval workflows during registration [46]. Admins can generate predictive insights using

machine learning graphs, manage stock replenishments, monitor low-stock alerts, and apply discounts as needed.

Both roles are authenticated using JWT tokens, and permissions are enforced in both the frontend routes (Angular) and backend routes (Flask) [47]. The UI components across roles are built with **modular Angular components**, and page routing is dynamically rendered based on user type.

Fig 37 - Admin Dashboard Overview

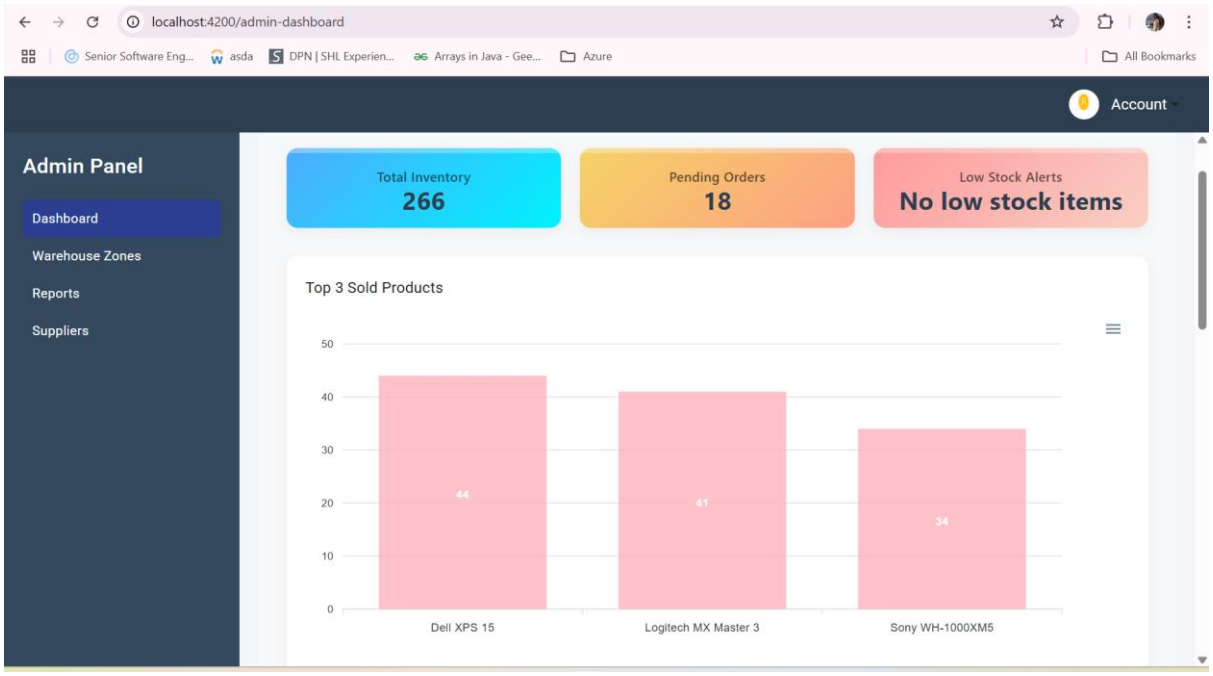
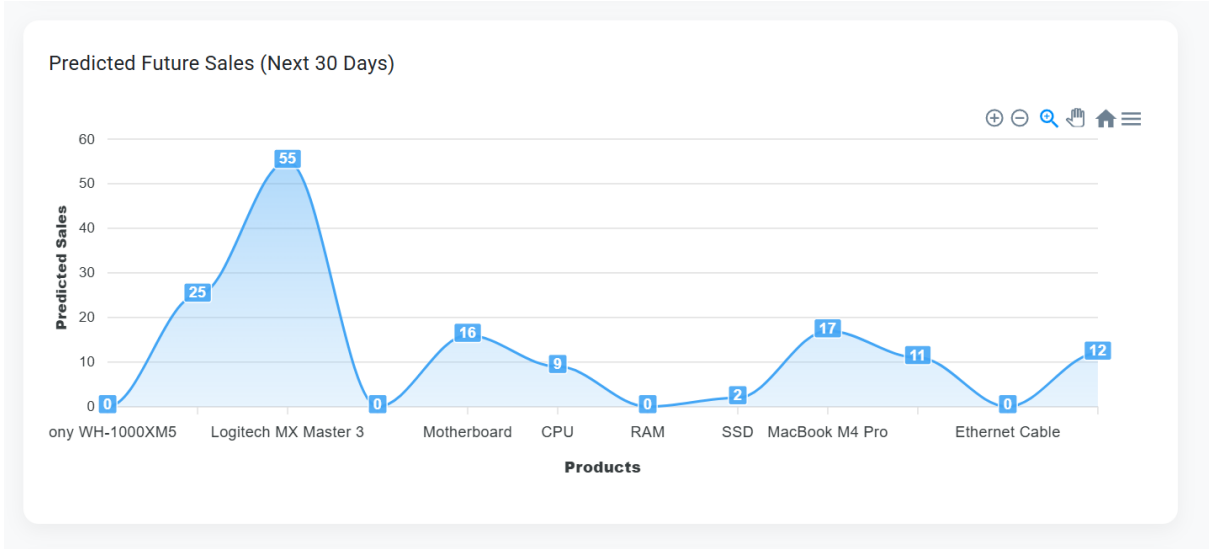


Fig 38 - Prediction graph of future sales



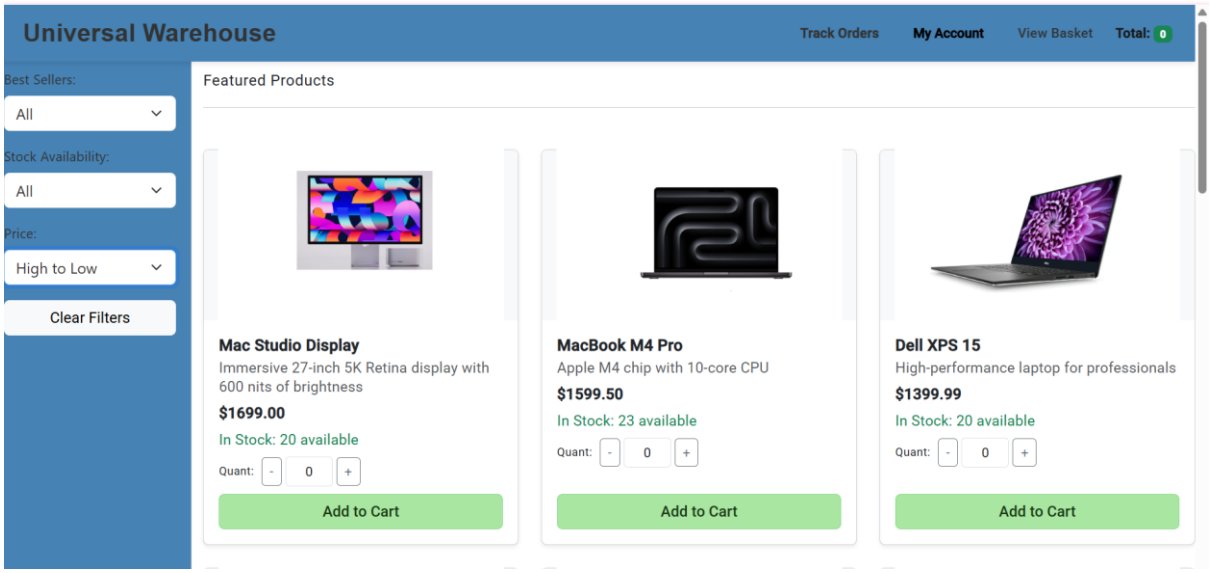
6.2 Client Dashboard Functionalities

The **Client Dashboard** acts as the central interface for end-users to explore available products, filter based on stock levels or popularity, place orders, and track purchases. Built entirely with **Angular standalone components**, it integrates seamlessly with the Flask backend using RESTful API calls [45].

Product Display and Filtering

Clients are presented with a list of products fetched via GET /customer_dashboard/products. Each product card displays the name, description, stock status, price, and a quantity input. The filtering system enables users to refine product listings based on **stock availability** (e.g., in stock, low stock), **price order**, and **bestseller rankings**, which is computed using the sold_quantity attribute stored in the backend database [48].

Fig 39 - Product Listing with Filters

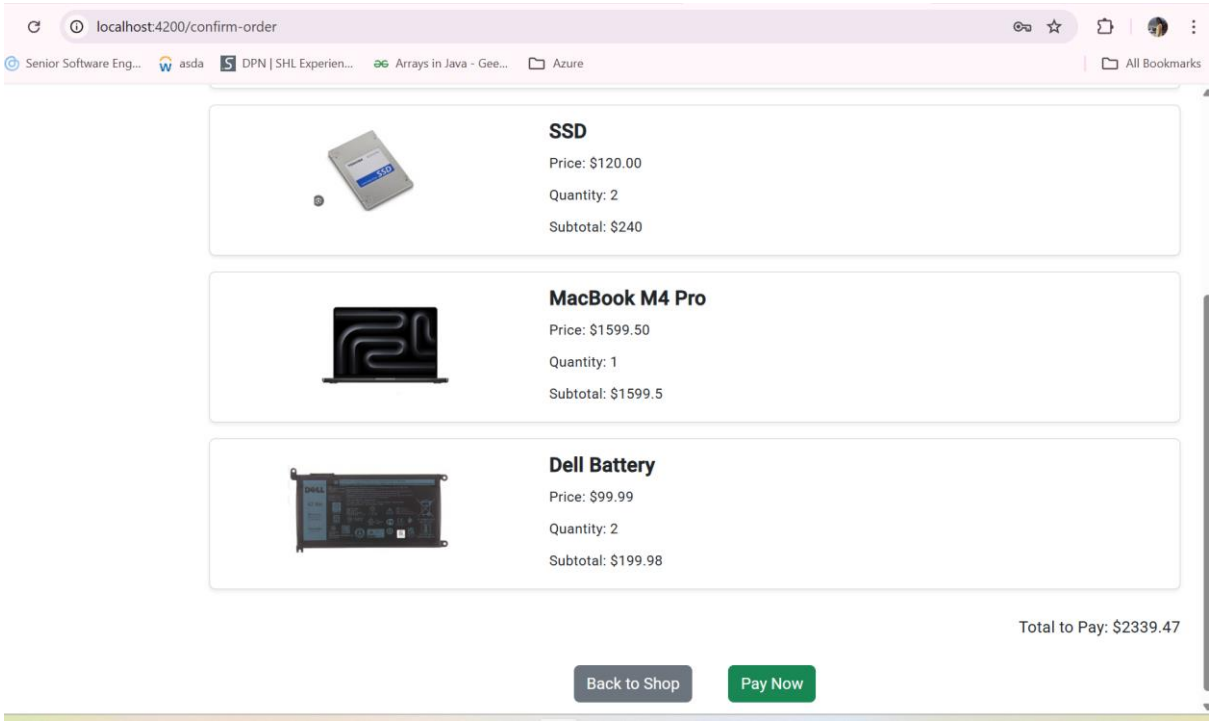


Cart Management and Order Placement

Quantity input fields include validations to ensure users cannot select more than the available stock. A client can click "Add to Cart" and view selected items using the "View Basket" feature. Upon confirmation, items are passed to the Confirm Order page (/confirm-order) along with the user's email stored in session storage.

This workflow involves a POST /customer_dashboard/confirm_order request which validates stock, calculates totals, and persists the order in the database. The order ID, total price, and item list are returned and displayed in the confirmation screen.

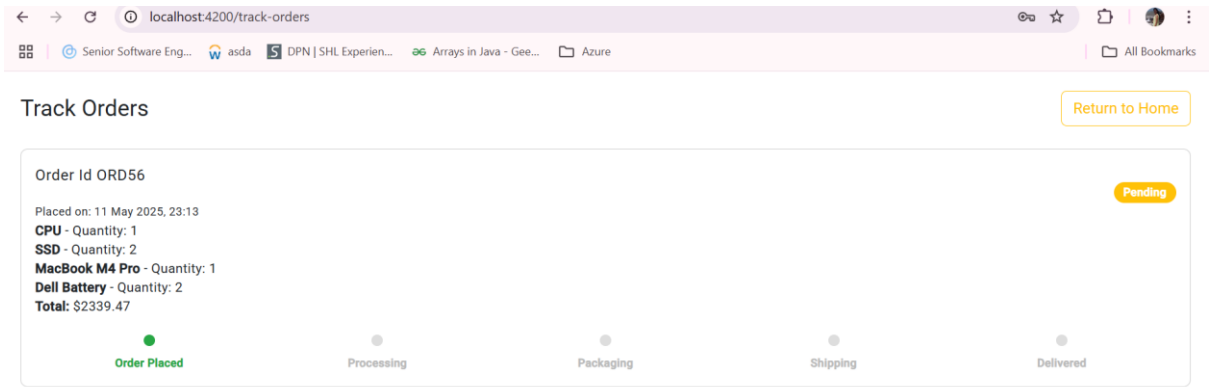
Fig 40 - Insert Confirm Order Page



Order Tracking

After placing an order, clients can monitor its progress using the Track Orders section, which displays a timeline of order statuses including Pending, Packaging, Delivered status. This is powered by a dynamic calculation of date differences on the Flask backend, ensuring status transitions are time-based rather than manual [49].

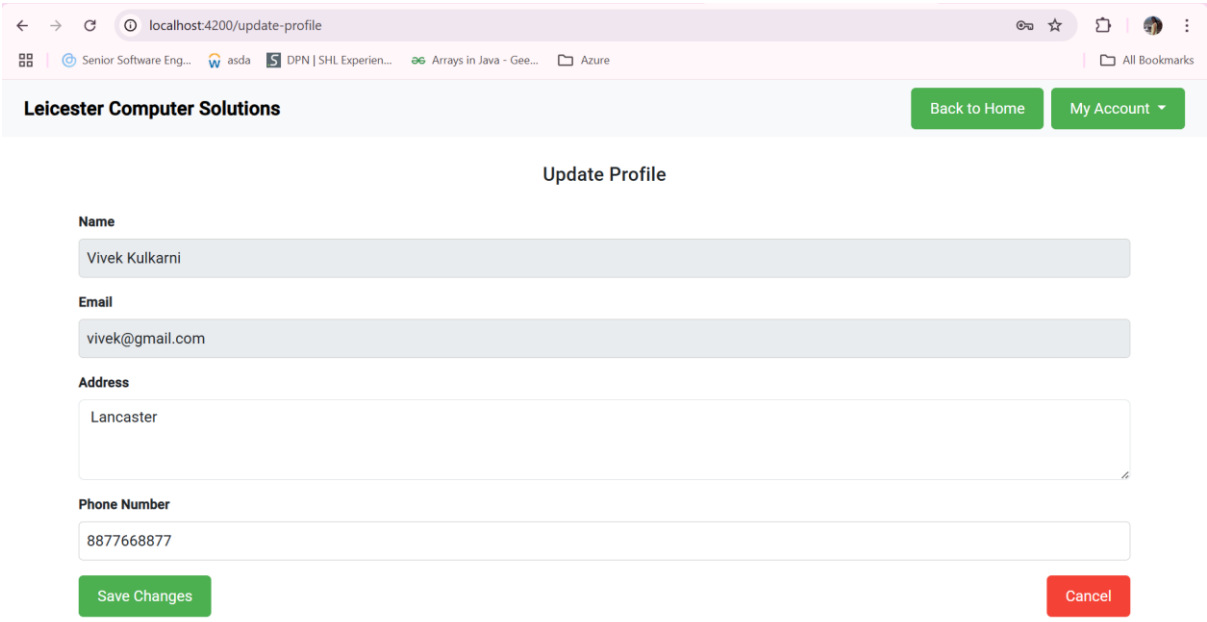
Fig 41 – Track Order Page



Profile Management

Clients may update their **phone number and address** from the Update Profile section (/update-profile), which triggers a POST /customer_dashboard/update_account API. Email is used as a secure identifier, and updates are reflected immediately due to SQLAlchemy's session commit logic.

Fig 42 – Update Profile



The client interface is designed to be **mobile-responsive**, with **Bootstrap**-based layout, intuitive buttons, toast notifications for actions like adding to cart, and visually appealing product cards [35].

6.3 Admin Dashboard Functionalities

The **Admin Dashboard** is the control center of the Stock Control System, enabling administrators to manage products, monitor inventory, track profits, place supplier orders, and view detailed reports. It is structured using Angular child routes under /admin-dashboard, and integrates with the Flask backend using authenticated APIs [50].

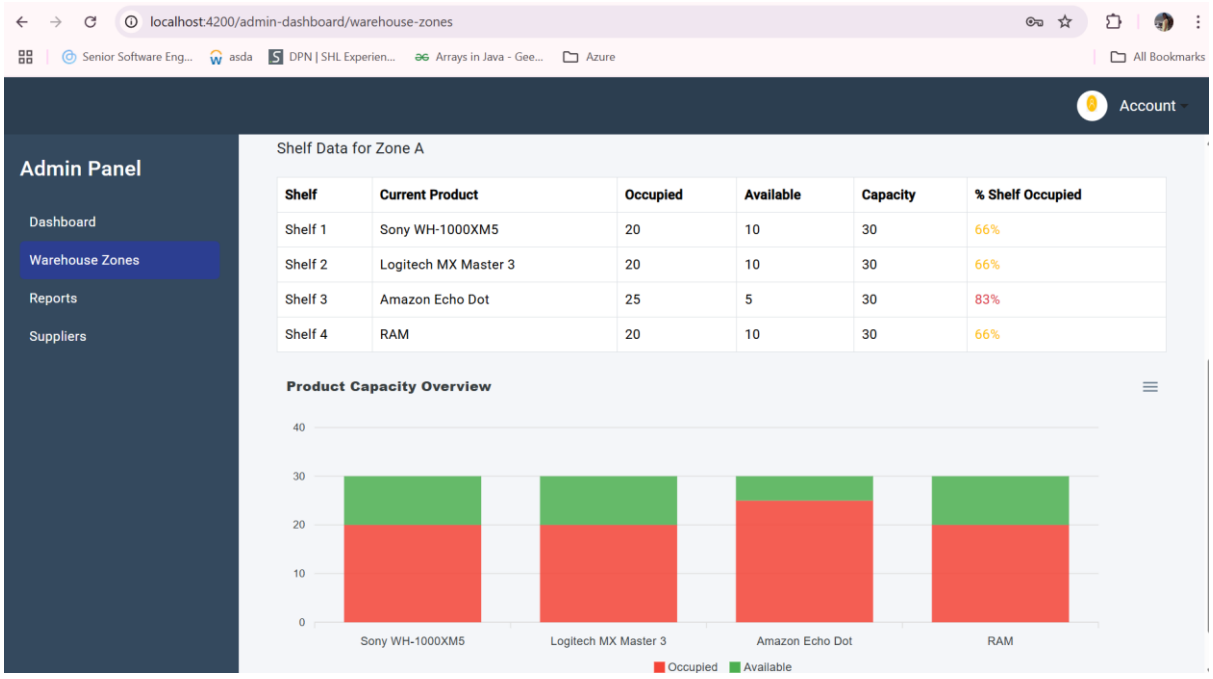
Inventory Overview and KPIs
The landing section of the admin dashboard which is DashboardHomeComponent presents high-level **Key Performance Indicators (KPIs)** such as total inventory, pending orders, low stock alerts, and top/least sold products. It also visualizes **daily profit trends over the last 10 days** using a GET /admin/overview API and Angular ApexCharts for dynamic charting [38].

Warehouse Zone and Shelf Management
Each warehouse is divided into zones (A to D) with specific storage characteristics named like Zone A for accessories, Zone B for high-value items. Admins can navigate to each zone

to view products, their shelf assignments, and current stock levels using the /admin/items_by_zone/<zone> endpoint.

A separate tab within this section displays shelf capacity usage, fetched from GET /admin/get_shelves_by_zone/<zone>, visualized using bar charts. Shelf status is color-coded to indicate fullness, allowing proactive space management [32].

Fig 43 - Zone and Shelf View



Supplier Orders and Validation

Admins can place restock orders through the **Supplier Order** page. On selecting a supplier, available products are fetched and quantities can be specified. A real-time **validation logic** ensures that shelf capacity is not exceeded (POST /admin/validate_shelf_capacity), preventing warehouse overstocking.

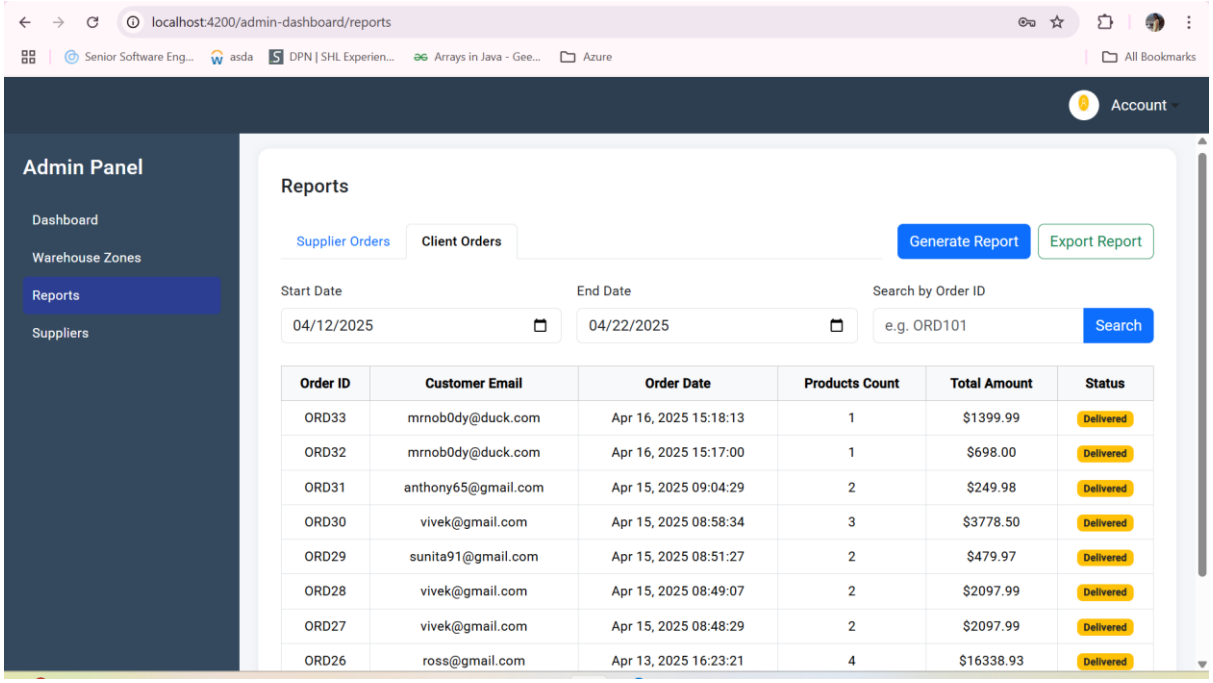
Once validated, the admin submits all selected items to POST /admin/submit_supplier_orders, which stores the order in SupplierOrder and SupplierOrderDetails tables [51].

Client and Supplier Order Reports

Comprehensive reports are available under the **Reports** tab. Admins can filter **client orders** by date range, status, and search by Order ID using GET /admin/get_client_order_reports. Similarly, **supplier orders** can be filtered by supplier name or date using GET /admin/get_supplier_order_reports.

Results include details like number of items, total cost, order status, and date, aiding administrative decision-making.

Fig 44 - Reports with export and search options:



6.4 Real-Time Stock Update Flow

The Stock Control System ensures accurate, up-to-date inventory management by dynamically reflecting stock changes across client and admin interfaces. This section outlines how product stock, shelf capacity, and order statuses are synchronized between the frontend and backend in real-time.

1. Stock Decrement and Low Stock Alerts on Client Orders
When a client confirms an order via POST /customer_dashboard/confirm_order, the backend immediately decrements the product's stock and increments the sold_quantity. If the new stock falls below a threshold (≤10), a low-stock email alert is triggered to all admins using the configured SMTP email utility email_utils.py [48].

Fig 45 - Customer_dashboard_routes.py:

```
192
193         subtotal = float(product.price) * quantity
194         total_price += subtotal
195
196         product.stock -= quantity
197         product.sold_quantity += quantity
198
199         # ✅ Check low stock
200         threshold = 10
201         if product.stock <= threshold:
202             low_stock_alerts.append(f"- {product.name}: {product.stock} units left")
203
```

This ensures the database reflects the real-time state of inventory and provides proactive alerts to maintain availability.

2. Shelf Capacity Validation Before Admin Orders

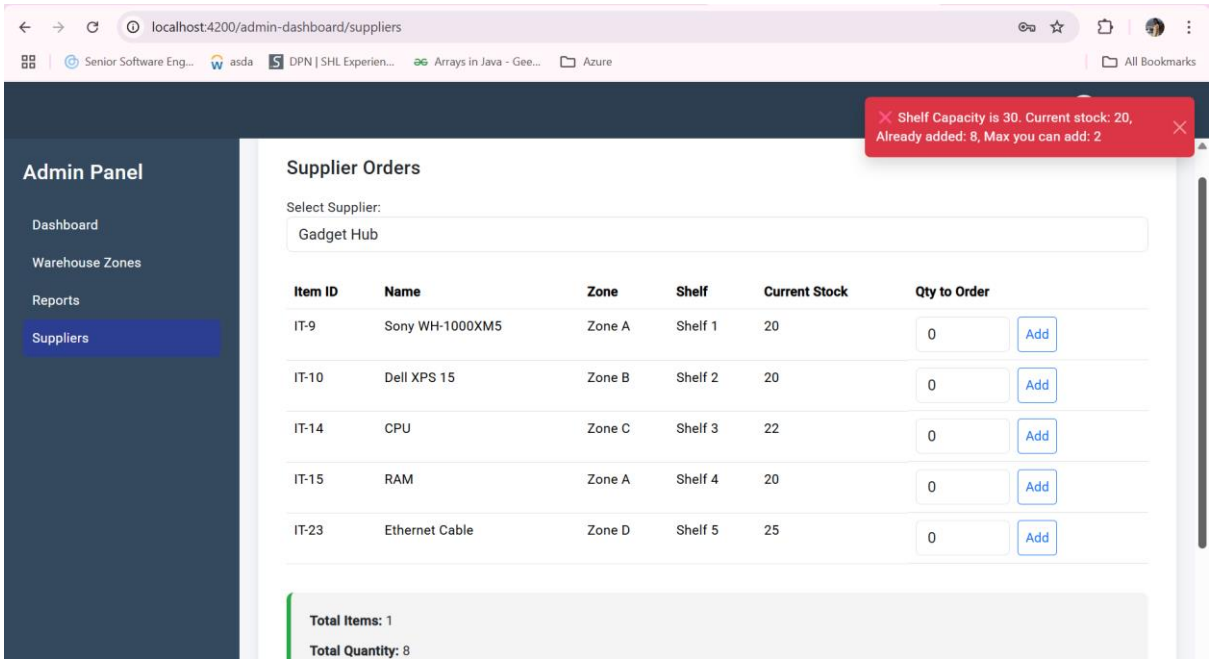
To prevent overfilling shelves, admins must validate item quantities before confirming supplier orders. A POST request to /admin/validate_shelf_capacity compares the requested quantity with current stock and shelf capacity (default 30 units) [40].

Fig 46 - Validation logic in admin_routes.py:

```
221
222     current_stock = product.stock
223     shelf_capacity = 30
224     available_space = shelf_capacity - current_stock
225
226     if requested_quantity > available_space:
227         return jsonify({
228             "success": False,
229             "message": f"Cannot add {requested_quantity} items. Shelf capacity is {shelf_capacity} and already ha
230         }), 400
231
```

On the frontend, a toast warning is shown, and the "Add" button is disabled for that product, ensuring real-time constraint enforcement [42].

Fig 47 - Screenshot of a toast warning



3. Real-Time Shelf Utilization Charts in Admin Dashboard

Once supplier orders are confirmed, stock levels for each shelf are updated, and shelf data is retrieved via GET /admin/get_shelves_by_zone/<zone>. Bar charts in the Angular admin dashboard then visualize:

- Occupied units
- Remaining space
- Usage percentage (e.g., “66% full”)

This helps admins assess physical warehouse conditions immediately after restocking [38].

4. Automatic Order Status Update

The /admin/trigger_update_order_status route is scheduled (or manually triggered) to iterate over all orders and update their status based on order age:

- **Day 0** → Pending
- **Day 1** → Packaging
- **≥ Day 2** → Delivered

Fig 48 - Backend logic in admin_routes.py:

```
488         for order in orders:
489             order_date = order.order_date.date()
490             days_diff = (today - order_date).days
491 
492             if days_diff == 0:
493                 new_status = 'Pending'
494             elif days_diff == 1:
495                 new_status = 'Packaging'
496             else:
497                 new_status = 'Delivered'
498 
499             if order.status != new_status:
500                 order.status = new_status
501                 updated_count += 1
502 
503         db.session.commit()
504 
505         return jsonify({
```

This logic ensures consistent tracking without manual intervention, avoiding stale statuses [50].

5. Session-Based Navigation State (Client)

To preserve user data like selected items and email during checkout, Angular uses history.state and sessionStorage. This enables smooth transitions across pages (/client-dashboard → /confirm-order → /thank-you) without data loss.

Fig 49 - client-dashboard.component.ts:

```
68    // ✅ Pass email + items
69    this.router.navigate(['/confirm-order'], {
70      state: {
71        items: selectedItems,
72        email: customerEmail
73      }
74    });
75  }
```

This approach allows real-time navigation with persistent context while reducing the need for repeated API calls [6].

6.5 Order Placement and Tracking

The order placement and tracking flow in the Stock Control System was designed to mimic real-world shopping workflows, enabling clients to select items, place orders, receive receipts, and track delivery status all in a smooth, intuitive manner. These operations span multiple Angular components and Flask API routes and represent the core transactional engine of the platform.

Order Placement Workflow

On the Client Dashboard, customers can increment/decrement product quantities and add them to a virtual cart. Clicking "Confirm Order" triggers a navigation event to the confirmation screen (/confirm-order), where the selected items, total price, and order ID are displayed.

Order Confirmation and Backend Integration

On the **Confirm Order** page (confirm-order.component.ts), a POST request is made to /customer_dashboard/confirm_order, where Flask:

- Creates a new order in the database
- Updates product stock and sold quantity
- Generates a unique order ID (e.g., ORD36)
- Calculates total cost

If stock is low (≤10 units), a separate low-stock alert email is sent to all admins automatically.

Email Receipt Functionality

Once the customer confirms their order and clicks **"Pay Now"**, another POST request is made to /customer_dashboard/send_order_receipt, which sends a structured order receipt using Gmail SMTP (configured via email_utils.py).

Order Tracking Mechanism

After placing an order, the client can access **Track Orders** via /track-orders, which fetches their order history using the GET /customer_dashboard/track_orders API. Statuses like **Pending**, **Packaging**, and **Delivered** are calculated based on the number of days since the order was placed:

Chapter 7

Testing and Evaluation

To ensure the robustness, reliability, and correctness of the Stock Control System, a comprehensive testing and evaluation strategy was employed. This chapter details the methodologies used to verify both individual components and full-system workflows across the backend the Flask framework, frontend which is Angular 19, and their integration via RESTful APIs. Testing efforts covered unit-level API validation, exception handling, client-server interactions, and dynamic features such as order tracking, real-time stock checks, and email notifications. Each testing phase aimed not only to identify bugs and logic inconsistencies but also to validate performance, usability, and correctness of system outputs under practical usage conditions.

7.1 Unit Testing with Pytest Methodology

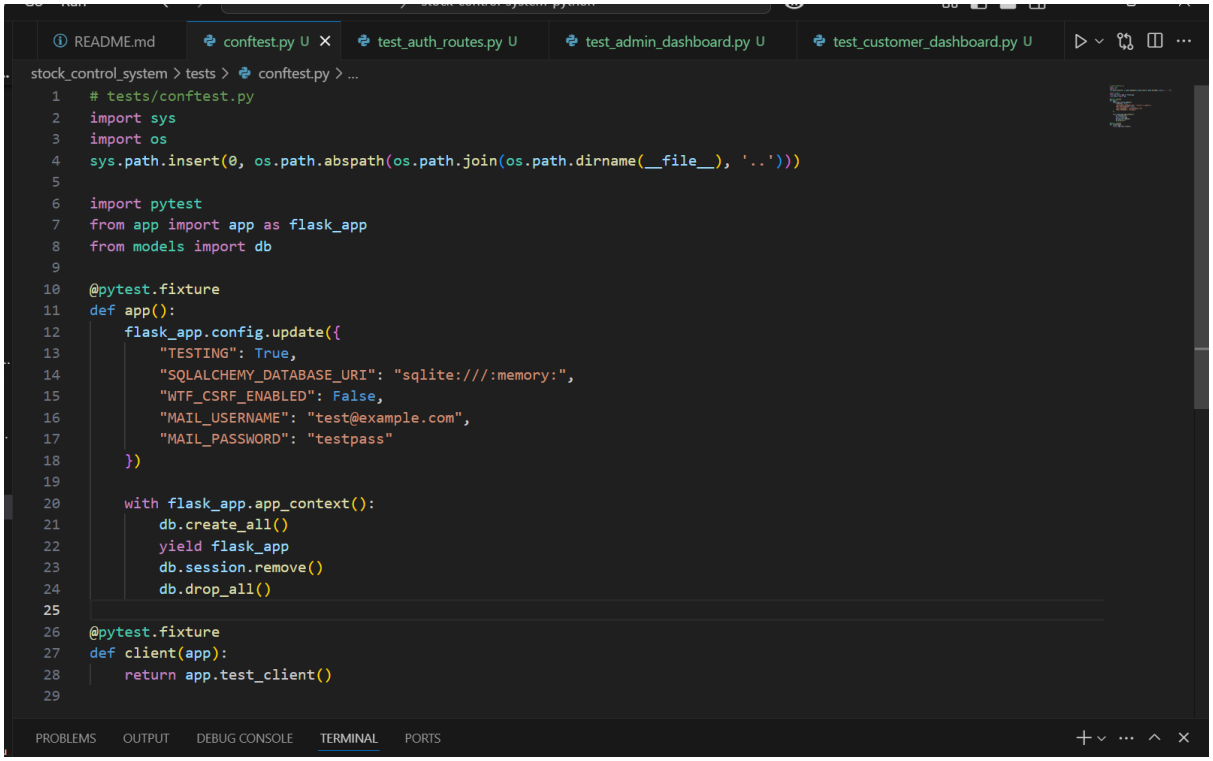
To enhance the reliability, maintainability, and test coverage of the backend services in the Stock Control System, unit testing was implemented using the **Pytest** framework in combination with **pytest-flask** and **pytest-mock**. These tools enabled automated, repeatable, and isolated validation of individual API endpoints built using the Flask framework.

Test Environment Setup

A dedicated tests/ folder was created in the backend project directory. This folder contained:

- conftest.py for shared configurations and fixtures
- Individual test files such as test_auth_routes.py, test_admin_dashboard.py, and test_customer_dashboard.py targeting specific modules
- A lightweight **in-memory SQLite database** (`sqlite:///:memory:`) was used for test isolation, ensuring test data did not interfere with the production MySQL instance. Flask's `test_client()` provided a testable interface to simulate HTTP requests internally without needing to run the full server.
- The structure ensured fast execution and a clean environment for every test run. A sample `conftest.py` setup is shown below:

Fig 50 Confest.py code

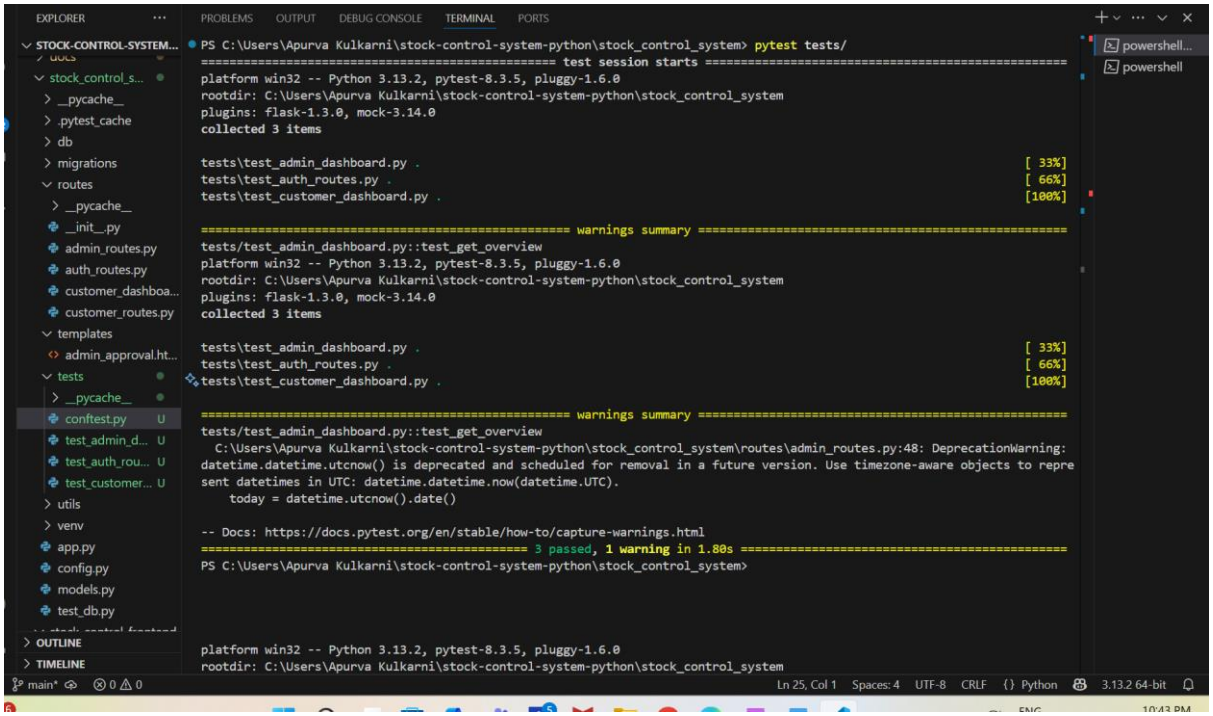


Coverage and Assertions

Test cases were written for key routes using different scenarios:

- **Valid input tests** for expected 200 OK responses
- **Invalid input tests** for correct handling of malformed or unauthorized requests
- **Assertions** to verify JSON structure, status codes, and business logic responses
- All unit tests were executed using the command `pytest tests/` from the backend root directory.
  The results confirmed that all selected endpoints passed successfully, validating expected outputs and response behaviors.
- **Figure** below shows the terminal output from a successful `pytest` run, indicating **three tests passed** and one non-blocking deprecation warning related to date-time handling.

Fig 51 Result of Pytest cases



7.2 Testing Methodology

The overall testing methodology followed a **black box testing approach**, where the focus was on validating inputs and outputs without inspecting internal code logic [56]. Functional testing was applied to ensure that each module behaved according to its specifications. Backend APIs were tested through **integration testing** using Postman, verifying that the communication between frontend and backend modules worked seamlessly. For end-user workflows, **manual end-to-end testing** was employed to simulate real interactions, validate form behaviour, and confirm multi-step processes such as registration, login, and order placement. These techniques collectively ensured that both individual components and the system as a whole were functioning correctly and robustly.

Backend Testing Approach

Postman was used extensively to simulate and test HTTP requests to backend API endpoints [52]. Key functionalities such as user registration, login, product retrieval, order placement, and sales forecasting were validated with a range of test cases:

- Valid Input Tests – to verify expected 200 OK responses and correct payloads.
- Invalid Input Tests – to trigger 400/401 errors and verify input validation.
- Authentication Tests – ensuring protected endpoints required valid JWT tokens.

Frontend Validation

Frontend testing was performed manually using browser-based inspection tools. Angular forms were evaluated for proper field-level validation based on Angular's reactive form mechanisms [2]. Angular forms were evaluated for proper field-level validation. Key checks included:

- Email and phone number format enforcement on registration.
- Toast notifications when input values exceeded allowed thresholds (e.g., shelf capacity on supplier order).
- Password match checks and OTP verification gating.
- Real-time error feedback and button state toggling (e.g., disabling "Add" on full shelves).

In addition, dynamic components such as product cards, charts, badges, and cart calculations were tested across user journeys to confirm correct rendering and data binding.

Developer Tools and Logging

Browser DevTools were used to monitor network requests, inspect JSON responses, and trace component state changes. Angular's error logs highlighted frontend binding issues. On the backend, Flask console logs and custom logging were used to identify unhandled exceptions, route mismatches, and misconfigured payloads [37].

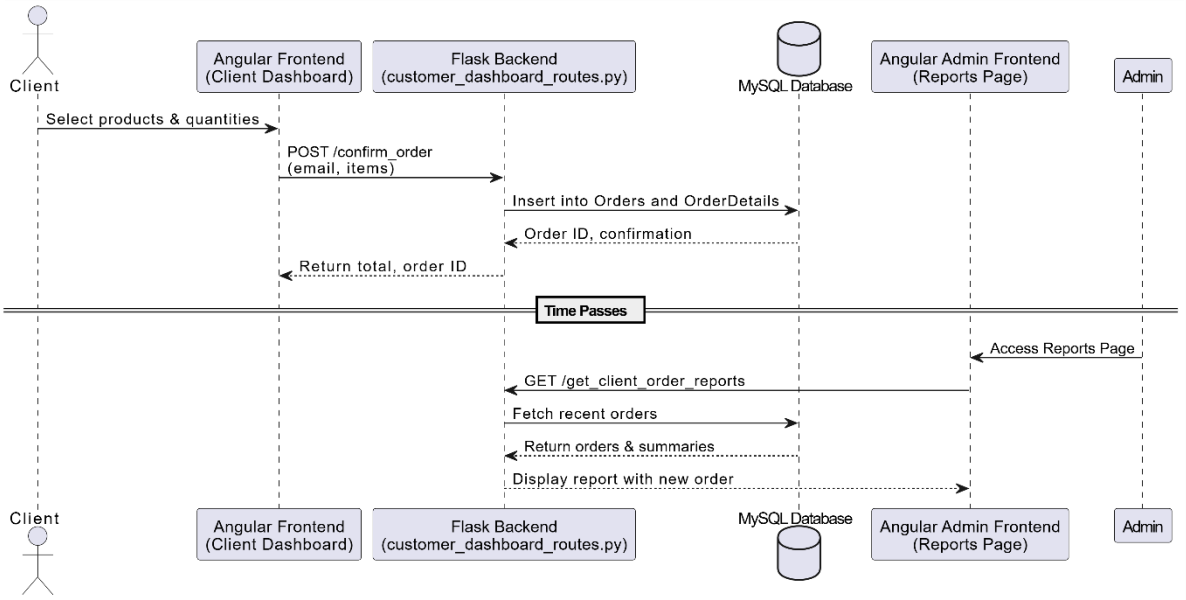
Error Handling and Edge Cases

In addition to successful flows, intentional errors were tested:

- Missing session emails during checkout
- Submitting zero or negative product quantities
- Reaching full shelf capacity

These edge cases were manually triggered to ensure the user interface displayed warnings and the backend safely rejected bad inputs, returning appropriate status messages [45][42]

Fig 52 - System Testing Workflow - Sequence Diagram



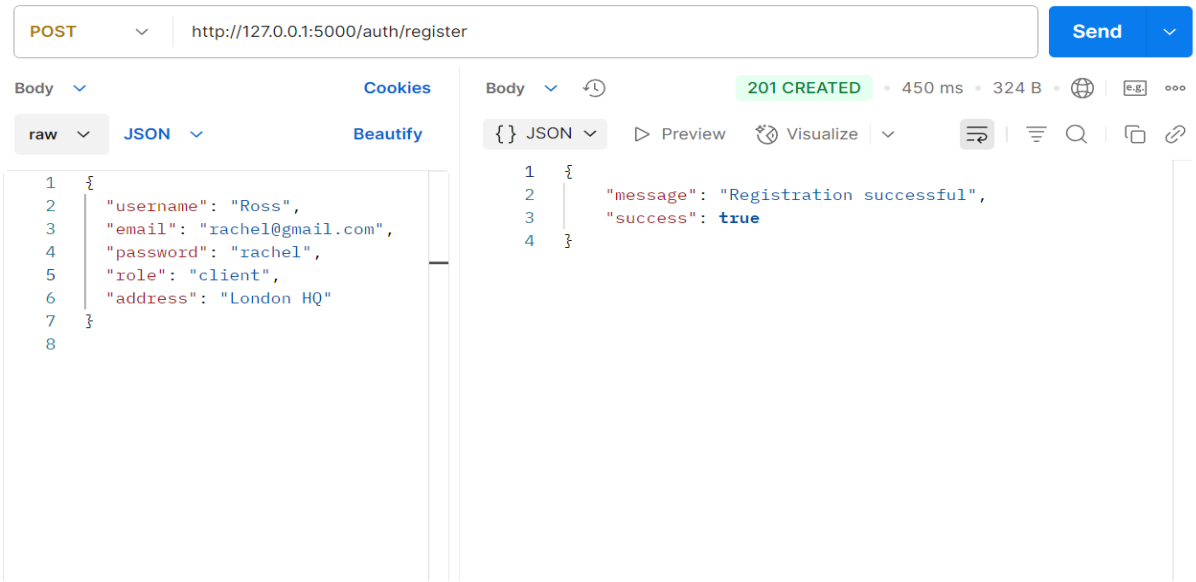
7.3 Backend API Testing

Backend API testing was a critical part of the system validation process to ensure that all Flask endpoints handled data exchange, authentication, and error responses reliably. Postman was used for manually invoking each RESTful route and examining the request-response cycle, response time, headers, and returned JSON data [52].

User Authentication APIs

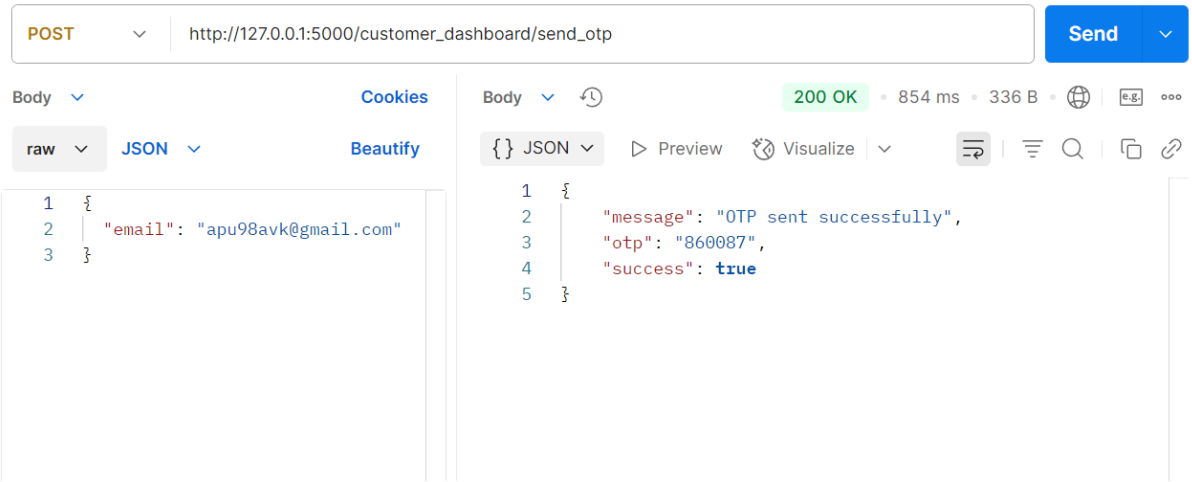
POST /register: Tested with complete and incomplete input data. Verified correct status codes (201 on success, 400 on missing fields) and proper email OTP delivery.

Fig 53 - Valid registration response



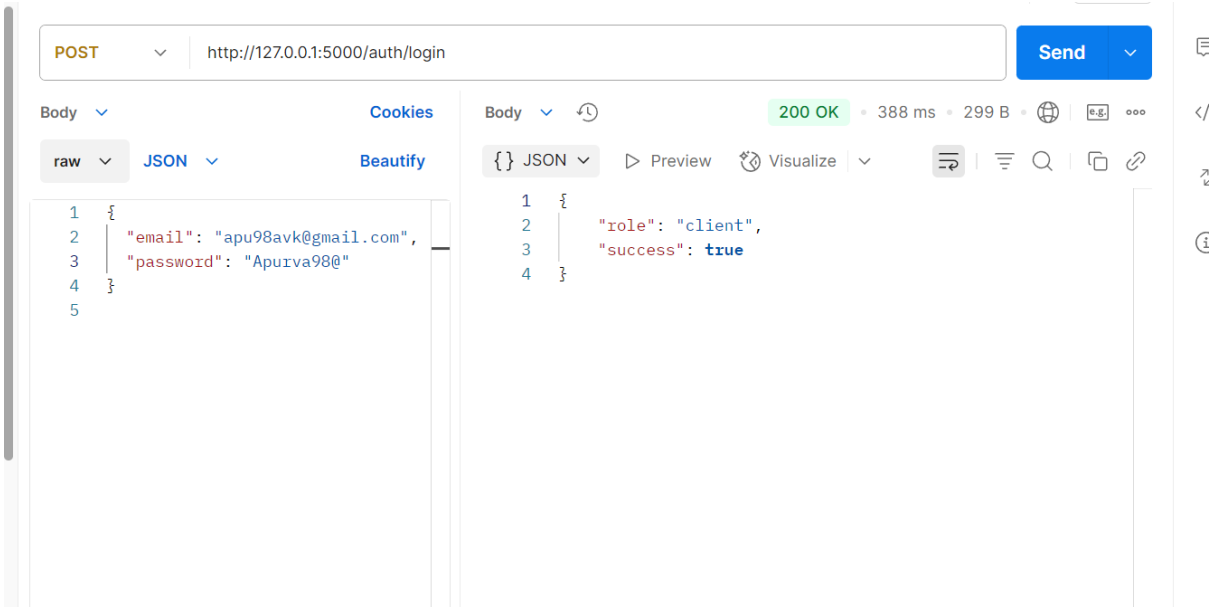
POST /verify_otp: Tested with correct and incorrect OTPs. Expected 200 OK for success and 401 Unauthorized for failure.

Fig 54 - OTP validation API



POST /login: Tested valid and invalid credentials. JWT Tokens possess the advantage of being able to be verified without the presence of an authorization server but are not capable of being revoked. [54].

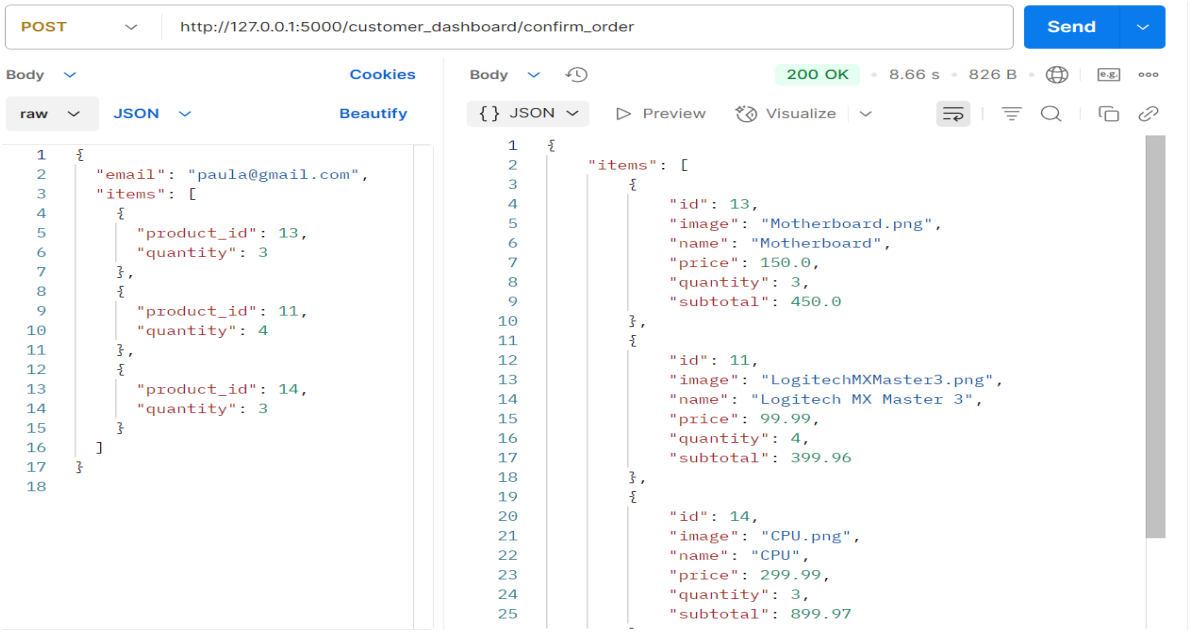
Fig 55 - JWT login response



Order Management APIs

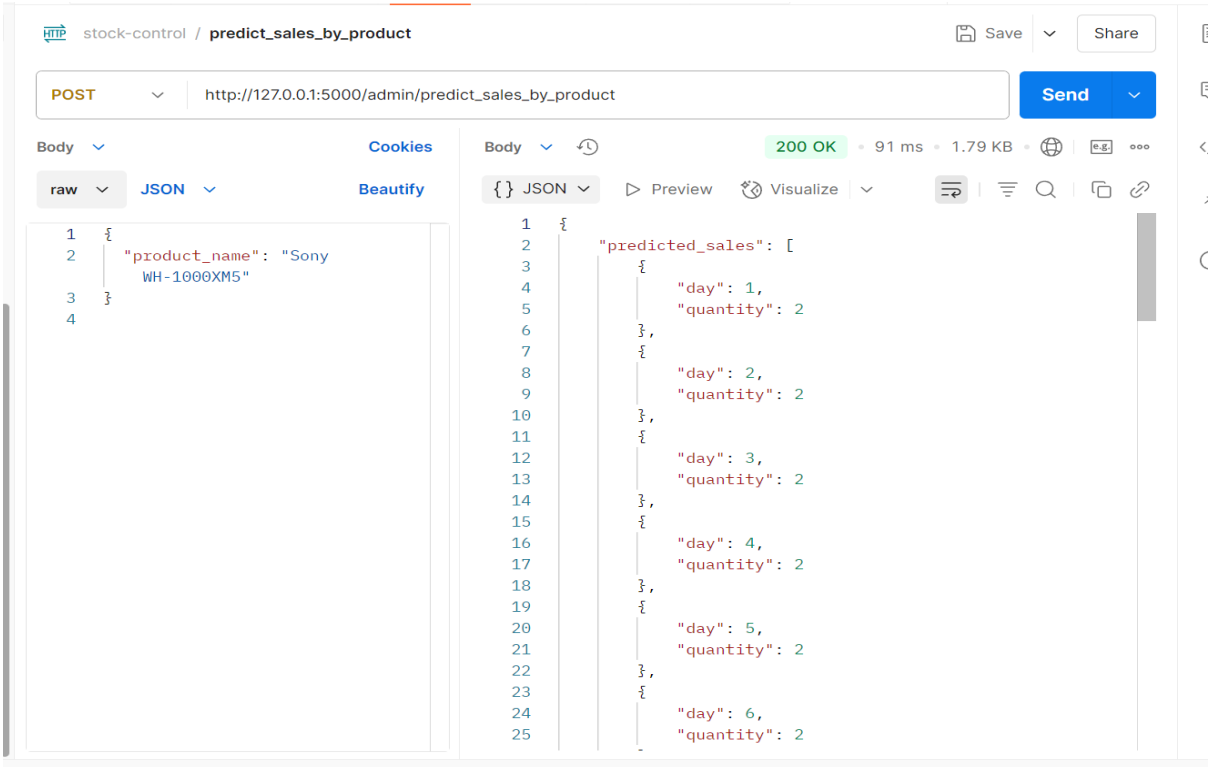
POST /confirm_order: Verified final confirmation triggers order ID generation (e.g., ORD38) and returns full order summary. Checked database record creation.
Screenshot: Confirm order response

Fig 56. Successful response of request of /confirm_order API on POSTMAN



POST /predict_sales_by_product: Sent product names to retrieve 90-day sales predictions using scikit-learns linear regression model [36]. Ensured proper JSON format with keys day and quantity.

Fig 57 Predict sales by product API testing in postman



Admin Supplier Order APIs

- **GET /admin/get_suppliers**: Verified supplier list is returned only with valid session cookies.
- **POST /admin/submit_supplier_orders**: Checked successful creation of supplier order (SO-101) and storage of product-level order details.

All responses were validated for:

- Status codes (200, 201, 400, 401, 500)
- JSON payload correctness
- Authentication enforcement using JWT [54]

Flask console logs and error handlers assisted in tracing exceptions during test failures [55]. API test results were documented with screenshots and are included as visual evidence for functional completeness.

7.4 Frontend Validation Testing

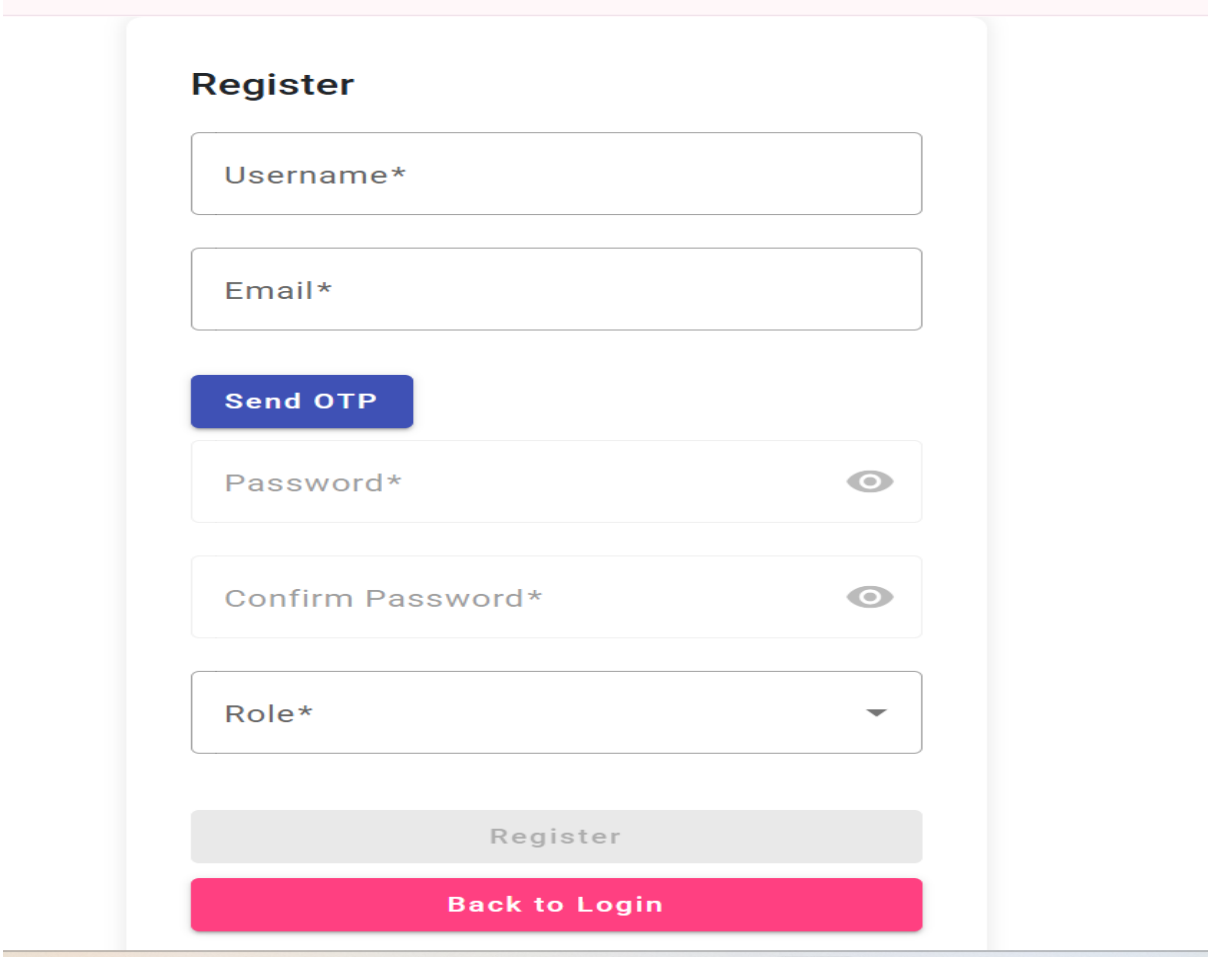
Frontend validation was essential to ensure that the Angular-based user interface responded appropriately to user inputs and enforced data integrity before backend submission. All major forms and components were manually tested in the browser using Chrome Developer Tools, Angular's reactive form mechanisms, and Bootstrap's built-in validation utilities.

Registration Form

The registration form included comprehensive input validations to ensure accurate and secure user data collection. Angular's reactive form controls were used to validate email format, and mobile numbers were restricted to exactly 10 digits. A country code dropdown preceded the phone number field to ensure regionally accurate input. If the entered number exceeded the limit or was left blank, a Bootstrap-styled alert was triggered to inform the user.

Additionally, OTP verification was integrated as a gatekeeping mechanism. Users could not proceed to enter their password until a valid 6-digit OTP was received and confirmed. This dynamic gating of password fields reinforced both security and correctness. Password and confirm password fields were also validated for equality, and submission was only allowed when all conditions were met.

Fig 58 Register page all the other input fields are disabled until OTP verification

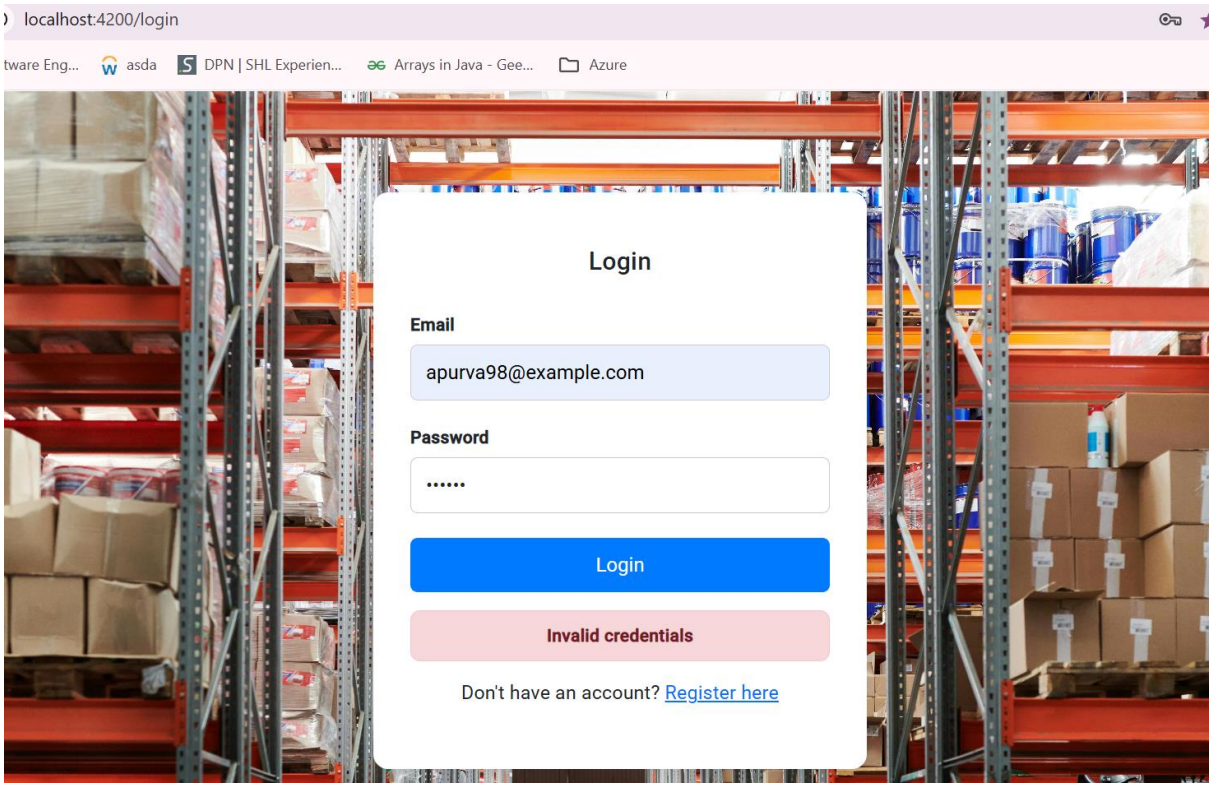


Login Page

On the login page, form submission was blocked if either the email or password field was empty. Angular's form state tracking enabled real-time validation feedback. In the case of invalid credentials, a non-intrusive toast notification was displayed, informing the user without reloading the page. This helped preserve entered data and enhanced the user experience.

The toast-based feedback for invalid login attempts is illustrated in Figure below.

Fig 59 Authentication Error on Login Page



Throughout all frontend testing, Chrome DevTools were used to inspect real-time form behaviour and monitor network interactions. Angular's built-in validation utilities and Bootstrap's responsive form classes were instrumental in enforcing and visualizing input constraints.

Chapter 8

Results and Future Scope

This chapter outlines the key outcomes of the implemented Stock Control System and highlights how the system performs in response to real user interactions and functional requirements. It also presents proposed enhancements based on the original planning table under recommended and optional categories. These future improvements aim to further improve the system's scalability, usability, and operational intelligence.

8.1 System Outcomes

Customer-Side Functionality

- **Product Browsing and Selection:**
  Customers can view a dynamic list of available products through a user-friendly Angular dashboard. Product details such as name, price, and description are fetched from the Flask backend, while images are loaded from a static assets folder. Quantity selectors allow customers to specify the number of units for each item.
- **Real-Time Price Calculation:**
  As customers adjust quantities, the total price updates instantly using Angular's two-way data binding. This improves accuracy and helps users make informed purchase decisions before placing an order.
- **Order Summary and Confirmation:**
  The Confirm Order page presents a breakdown of selected items along with the total cost. When the "Pay Now" button is clicked, the order data is sent to the backend and stored in the MySQL database. A unique order ID (e.g., ORD38) is generated for tracking purposes.
- **Email Receipt Integration:**
  After order confirmation, the customer receives an automated email containing the receipt. This is handled through Flask's SMTP email service, and the message includes product names, quantities, total price, and the generated order ID, providing professional and traceable confirmation.

Admin-Side Functionality

- **Dashboard with Product Insights:**
  Admin users access a dedicated dashboard displaying top-selling and least-selling products through charts built using ApexCharts. This helps in evaluating product performance and inventory trends over time.
- **Sales Forecasting Module:**
  A machine learning model using linear regression using scikit-learn predicts daily product sales for the next 90 days. The results are presented as line charts, allowing admins to make proactive decisions about stock replenishment.
- **Supplier Order Management:**
  Admins can select products that are low in stock, assign shelf zones, and specify quantities for ordering. Once submitted, these orders are saved in the database and

displayed in a structured order summary page. Validation mechanisms prevent over-ordering by triggering toast warnings if shelf capacity is exceeded.

- **Validation and Workflow Reliability:**
  All admin workflows from viewing stock to placing supplier orders are equipped with frontend validations, toast alerts, and success summaries. This ensures data accuracy and reduces operational errors.

All implemented modules were tested for functionality, integration, and real-world usage through API testing using Postman, UI validation, and end-to-end black-box testing. Together, these outcomes represent a fully operational and extensible inventory control system tailored to both customers and warehouse administrators.

8.2 Future Scope

If more time and resources were available, the Stock Control System could be further enhanced through the following developments, each aimed at improving system accessibility, automation, and business integration:

1. Cloud Deployment of the Flask Backend:
Hosting the backend on platforms like AWS, Google Cloud Platform (GCP), or Heroku would enable real-time access for distributed users. Cloud deployment would make the application scalable, accessible from any location, and suitable for multi-user environments. It would also support future integration with CI/CD pipelines and DevOps practices.

2.Automated Supplier Orders Based on Low Stock and Sales Prediction:
The system can be enhanced to automatically generate and dispatch supplier orders without requiring manual admin intervention. By integrating low stock thresholds and machine learning-based sales forecasting, the system would proactively identify replenishment needs and trigger orders to suppliers, ensuring optimal inventory levels and reducing the risk of stockouts.

3. Discount Feature Based on Sales Forecasting:
The admin panel could be expanded to allow discount tagging for products expected to underperform in future sales. This feature would rely on the machine learning model's predictions and enable admins to assign dynamic discounts (e.g., 10%, 20%, 30%) to such items. These offers could be highlighted on the customer dashboard using badges or special labels, helping reduce overstock.

4. Chatbot Integration for User Assistance:
A conversational chatbot could be embedded within the system to assist both customers and admins. For customers, the chatbot could provide order tracking, product inquiries, or navigation help. For admins, it could support dashboard interactions or quick data lookups. This feature could be developed using tools like Dialogflow, Rasa, or a custom NLP model and integrated into the Angular frontend.

Each of these enhancements offers practical value and aligns with the evolving demands of modern warehouse management systems. Though not part of the current implementation, they form a strong foundation for future development phases.

8.3 Project Uniqueness

The Stock Control System developed in this project offers several features that distinguish it from existing inventory management solutions. These unique characteristics are outlined below:

1. Integrated Sales Forecasting Using Machine Learning:

The system uses a linear regression model to forecast product sales over the next 90 days. This predictive feature enables admins to make proactive, data-driven stock replenishment decisions. In contrast, most traditional systems rely on fixed thresholds or manual estimates for reordering.

2. Automated Low-Stock Alerts for Admins via Email:
The system continuously monitors product inventory levels and sends automated email alerts to admin users when any item falls below the defined stock threshold. This functionality ensures that restocking decisions can be made promptly and reduces the risk of supply shortages. It simulates a real-world warehouse notification system and strengthens inventory responsiveness.

3. Shelf-Level Supplier Order Management with Real-Time Validation:
Admin users can place supplier orders while considering zone and shelf-level constraints. The system validates quantity input against shelf capacity, triggering toast alerts when limits are exceeded. Such spatial awareness is typically absent in generic inventory systems.

3. End-to-End Email-Integrated Order Workflow:
Customers receive automated order receipts via email immediately after confirming their purchase. This workflow, managed through Flask's SMTP integration, provides professional transactional visibility and enhances trust an element commonly seen in commercial e-commerce systems.

4. Shelf-Level Supplier Order Management with Real-Time Validation:
Admins can place supplier orders by selecting low-stock products while specifying shelf and zone details. The system performs shelf capacity validation in real-time and uses toast alerts to notify the admin if the entered quantity exceeds the allowed space. This level of precision is rarely found in commercial systems, which often ignore physical storage limitations.

5. Modern Frontend Using Angular 19:
The application uses Angular 19 the latest version of the Angular framework which offers significant improvements over previous versions. Angular 19 supports standalone components (reducing reliance on NgModules), improved performance, smaller bundle sizes, and better developer experience with enhanced type checking and build speeds. These upgrades make the frontend more modular, faster, and easier to maintain compared to older Angular versions.

6. Lightweight, Full-Stack Open-Source Architecture:
The system is built using Flask for backend logic, Angular 19 for a modern frontend, and MySQL for relational data management. This open-source technology stack offers flexibility,

modularity, and cost-effectiveness, making the system suitable for real-world deployment without licensing overhead.

These features collectively position the project as more than a basic academic system. It is a compact, scalable, and intelligent inventory solution that blends practical usability with predictive intelligence bridging the gap between traditional stock control methods and modern AI-driven inventory management platforms.

Chapter 9

Conclusion

This project successfully achieved its core objectives of streamlining inventory control and integrating intelligent forecasting into day-to-day warehouse operations. By enabling both customer and admin roles to interact with the system through intuitive interfaces, the application supports complete end-to-end workflows from order placement to restocking while maintaining accuracy, responsiveness, and traceability. The integration of email alerts, machine learning predictions, and shelf-level supplier validations introduces a level of automation and intelligence often absent in basic inventory systems.

The use of Angular 19 provided a forward-compatible frontend framework with performance improvements and modern component architecture, while Flask enabled lightweight yet powerful backend routing and data processing. The system architecture remains modular and extensible, allowing for seamless future additions such as discount logic, chatbot assistance, or ERP integration.

Overall, the Stock Control System stands as a robust and scalable solution designed to address the operational complexities of small to medium-scale warehouses and retail environments. Unlike conventional inventory tools that offer static functionalities, this system integrates a forward-looking approach by embedding predictive analytics, dynamic user interactions, and automation-driven workflows into its core architecture. The ability to forecast future sales, validate shelf-level constraints, and automate communications through email reflects a deliberate alignment with real-world business challenges such as overstocking, stockouts, and manual inefficiencies.

Beyond its technical execution, the project embodies strategic thinking in software design balancing simplicity with extensibility. It demonstrates how an open-source, modular technology stack can be harnessed to deliver enterprise-level features without the overhead of commercial platforms. Features like admin-level decision support, customer engagement through receipts, and role-specific workflows reflect a clear understanding of stakeholder requirements. Additionally, the architecture is purposefully designed to support future enhancements such as ERP integrations, automated discounting, and AI-based customer assistance, making it adaptable to evolving business and technological trends.

In essence, this project is not only a demonstration of full-stack development capabilities but also a reflection of applied problem-solving, domain understanding, and scalable system design. It bridges the gap between academic prototypes and production-ready warehouse solutions, providing a strong foundation for future innovation and deployment.

References

[1] Kang, Y., & Gershwin, S. B. (2005). Information inaccuracy in inventory systems: stock loss and stockout. *IIE transactions*, *37*(9), 843-859.

[2] Vaka, D. K. (2024). Integrating inventory management and distribution: A holistic supply chain strategy. *the International Journal of Managing Value and Supply Chains*, *15*(2), 13-23.

[3] Pal, S. (2023). Revolutionizing warehousing: Unleashing the power of machine learning in multi-product demand forecasting. *International Journal for Research in Applied Science and Engineering Technology (IJRASET)*, *11*, 615-619.

[4] Kumar, A., Karnik, N., & Chafle, G. (2002). Context sensitivity in role-based access control. *ACM SIGOPS Operating Systems Review*, *36*(3), 53-66.

[5] Shwe, T. (2023). Optimizing warehouse management for small and medium-sized enterprises: a case study of connected Finland Oy.

[6] Prabu, V. P. Improving Warehouse Automation Using Artificial Intelligence and Robotics.

[7] Kumar, S. S. (2025). Smart Inventory and Logistics Management System for MSME'S Remote Locations.

[8] Wynn, S. (2021). The financial impact of manual inventory record errors.

[9] Shahzad, U. (2023). A comparative analysis of ERP system providers.

[10] Papaspirou, V., Papathanasaki, M., Maglaras, L., Kantzavelou, I., Douligeris, C., Ferrag, M. A., & Janicke, H. (2023). A novel authentication method that combines honeytokens and Google authenticator. *Information*, *14*(7), 386.

[11] Ukpongson, M. P. (2023). REST API for a store management system using Flask.

[12] Jatana, N., Puri, S., Ahuja, M., Kathuria, I., & Gosain, D. (2012). A survey and comparison of relational and non-relational database. *International Journal of Engineering Research & Technology*, *1*(6), 1-5.

[13] Alabi, M. (2023). Predictive Analytics for Product Planning and Forecasting.

[14] Adams, M. Visualizing Warehouse Data: Dashboards and Reporting in the Cloud.

[15] Sommerville, I. (2016). *Software Engineering* (10th ed.). Pearson Education

[16] Paradkar, S. (2017). *Mastering non-functional requirements*. Packt Publishing Ltd.

[17] Bass, L., Clements, P., & Kazman, R. (2012). *Software Architecture in Practice*. Addison-Wesley.

[18] Google Developers. (2023). *Sending Emails with Gmail SMTP*. https://developers.google.com/workspace/gmail/api/guides/sending (accessed 8 May 2025)

[19] Angular Docs. (2024). *What's New in Angular 16*. https://blog.angular.dev/angular-v16-is-here-4d7a28ec680d (accessed 8 May 2025)

[20] Géron, A. (2022). *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems.* " O'Reilly Media, Inc.".

[21] Microsoft Docs. (2023). *JWT Authentication in ASP.NET Core*. https://learn.microsoft.com/en-us/aspnet/core/security/authentication/configure-jwt-bearer-authentication?view=aspnetcore-9.0 (accessed 09 May 2025)

[22] Tanenbaum, A. S., & Van Steen, M. (2007). *Distributed Systems: Principles and Paradigms* (2nd ed.). Pearson. https://vowi.fsinf.at/images/b/bc/TU_Wien-Verteilte_Systeme_VO_%28G%C3%B6schka%29_-_Tannenbaum-distributed_systems_principles_and_paradigms_2nd_edition.pdf (accessed 09 May 2025)

[23] Python Software Foundation. (2023). smtplib — SMTP protocol client.

[24] Grinberg, M. (2018). *Flask Web Development: Developing Web Applications with Python*. 2nd ed. O'Reilly Media. https://www.oreilly.com/library/view/flask-web-development/9781491991725/ (accessed 09 May 2025)

[25] James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). *An Introduction to Statistical Learning*. Springer. https://static1.squarespace.com/static/5ff2adbe3fe4fe33db902812/t/6009dd9fa7bc363aa822d2c7/1611259312432/ISLR+Seventh+Printing.pdf (accessed 09 May 2025)

[26] Date, C. (2007). *Date on Database: Writings 2000-2006*. Apress.

[27] Grinberg, M. (2018). *Flask Web Development* (2nd ed.). O'Reilly Media. https://www.oreilly.com/library/view/flask-web-development/9781491991725/ (accessed 09 May 2025)

[28] Fowler, M. (2012). *Patterns of enterprise application architecture*. Addison-Wesley.

[29] Pressman, R. S. (2005). *Software engineering: a practitioner's approach*. Palgrave macmillan.

[30] Kodali, N. (2022). Angular's Standalone Components: A Shift Towards Modular Design. *Turkish Journal of Computer and Mathematics Education (TURCOMAT) ISSN*, *3048*, 4855.

[31] Ronacher, A. (2010). *The Flask Framework Documentation*. https://flask.palletsprojects.com/en/stable/ (accessed 09 May 2025)

[32] SQLAlchemy. (n.d.). *SQLAlchemy ORM Documentation*. https://docs.sqlalchemy.org/en/20/ (accessed 11 May 2025)

[33] Angular Documentation. (2023). *Component Architecture and Routing*. Retrieved from https://angular.dev/ (accessed 11 May 2025)

[34] Flanagan, D. (2011). *JavaScript: the definitive guide*. " O'Reilly Media, Inc.".

[35] Bootstrap Docs. (2023). *Modals, Toasts, and Responsive Utilities*. Retrieved from https://getbootstrap.com (accessed 11 May 2025)

[36] Scikit-learn: Linear Regression https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html (accessed 11 May 2025)

[37] Flask Official Documentation https://flask.palletsprojects.com/en/stable/ (accessed 12 May 2025)

[38] ApexCharts Angular Integration https://apexcharts.com/docs/angular-charts/ (accessed 12 May 2025)

[39] Gmail SMTP Settings https://support.google.com/mail/answer/7126229 (accessed 12 May 2025)

[40] Angular HTTPClient: https://v17.angular.io/guide/understanding-communicating-with-http (accessed 12 May 2025)

[41] Python smtplib https://docs.python.org/3/library/smtplib.html (accessed 12 May 2025)

[42] Flask Documentation. “Error Handling in Flask.” https://flask.palletsprojects.com/en/stable/errorhandling/ (accessed 12 May 2025)

[43] Angular Documentation. “Forms and Input Validation.” https://v17.angular.io/guide/reactive-forms (accessed 12 May 2025)

[44] Angular Documentation. “Route Guards.” https://angular.io/guide/router#milestone-5-route-guards (accessed 12 May 2025)

[45] Angular Documentation. “Component Interaction.” https://angular.io/guide/component-interaction (accessed 12 May 2025)

[46] Flask-Login Docs. “Managing User Sessions in Flask.” https://flask-login.readthedocs.io/en/latest/ (accessed 12 May 2025)

[47] JWT.io. “Introduction to JSON Web Tokens.” https://jwt.io/introduction (accessed 12 May 2025)

[48] Flask-SQLAlchemy Docs.
https://flask-sqlalchemy.palletsprojects.com/en/stable/ (accessed 12 May 2025)

[49] Python datetime Module.
https://docs.python.org/3/library/datetime.html (accessed 12 May 2025)

[50] Angular Routing Guide.
https://angular.io/guide/router (accessed 12 May 2025)

[51] Flask Request Handling
https://flask.palletsprojects.com/en/stable/quickstart/#http-methods (accessed 12 May 2025)

[52] Postman. "Postman API Platform.
https://www.postman.com/ (accessed 12 May 2025)

[53] Angular Docs. "Form Validation.
https://v17.angular.io/guide/form-validation (accessed 12 May 2025)

[54] Bucko, A., Vishi, K., Krasniqi, B., & Rexha, B. (2023). Enhancing jwt authentication and authorization in web applications based on user behavior history. *Computers*, *12*(4), 78.

[55] Aggarwal, S. (2023). *Flask Framework Cookbook: Enhance your Flask skills with advanced techniques and build dynamic, responsive web applications*. Packt Publishing Ltd.

[56] Nidhra, S., & Dondeti, J. (2012). Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, *2*(2), 29-50.