|  |
| --- |
| CO3204 Software Engineering Project Dissertation |
| **Web-Based Inventory Management System for Small Business** |
| School of Computing and Mathematical Sciences, University of Leicester |

|  |
| --- |
| Choudhury, Arif  04-2025 |

Table of Contents

[**Declaration** iii](#_Toc196855565)

[**Abstract** 1](#_Toc196855566)

[**1.** **Introduction** 1](#_Toc196855567)

[**1.1 Aims** 2](#_Toc196855568)

[**1.2 Objectives** 3](#_Toc196855569)

[**2.** **Survey of Literature/ Information Sources** 3](#_Toc196855570)

[**3.** **Design and Implementation** 5](#_Toc196855571)

[**3.1 System Architecture** 5](#_Toc196855572)

[3.1.1 Interactions Between Components 6](#_Toc196855573)

[**3.2 Frontend Design** 8](#_Toc196855574)

[3.2.1 Technology Justification 8](#_Toc196855575)

[3.2.2 Structure of React Components 9](#_Toc196855576)

[3.2.3 Design Principles 16](#_Toc196855577)

[**3.3 Backend Implementation** 16](#_Toc196855578)

[3.3.1 Technology Justification 17](#_Toc196855579)

[3.3.2 RESTful API Route Structure 17](#_Toc196855580)

[3.3.3 JWT Authentication and Role-Based Access Control 18](#_Toc196855581)

[3.3.4 Security Practices 20](#_Toc196855582)

[3.3.5 Core Functionalities 21](#_Toc196855583)

[**3.4 Database and Authentication** 23](#_Toc196855584)

[3.4.1 Technology Justification 23](#_Toc196855585)

[3.4.2 Database Structure 24](#_Toc196855586)

[3.4.3 User Data Structure and Role Management 24](#_Toc196855587)

[3.4.4 Inventory and Task Data Structure and Implementation 26](#_Toc196855588)

[3.4.5 Firebase Authentication Integration 27](#_Toc196855589)

[**4.** **Results and Discussion** 28](#_Toc196855590)

[**4.1 Demonstrating Core Functionalities** 29](#_Toc196855591)

[**4.2 Discussion of System** 42](#_Toc196855592)

[**4.3 Evaluation against project goals** 44](#_Toc196855593)

[**4.4 Challenges and limitations** 46](#_Toc196855594)

[**5.** **Critical Appraisal** 47](#_Toc196855595)

[**5.1 Critical Analysis of this Project** 47](#_Toc196855596)

[**5.2 Social, Commercial and Sustainability Context** 47](#_Toc196855597)

[**5.3 Personal Development** 48](#_Toc196855598)

[**Conclusion** 48](#_Toc196855599)

[**References** 49](#_Toc196855600)

# **Declaratio**A close-up of a text Description automatically generated**n**

Arif Choudhury



## **Abstract**

This project describes a cloud-based inventory management system for small businesses with no organised IT infrastructure. The project provides a solution for some of the challenges like inventory visibility, operational inefficiencies, and limited technical expertise, offering a streamlined and accessible solution. The application consists of a single page application frontend build with React.js, backend with Flask, and user authentication and database services through Firebase. The cloud-based application allows businesses to manage their inventory in real-time with minimal setup.

This system has a number of features such as: secure login with email link verification or Google OAuth, Role-Based Access Control (RBAC), real time stock updating with Firestore listeners, easy to use inventory management interfaces, CSV bulk upload for large amounts of inventory, visualisation, and ability to generate reports in csv or pdf format for insights whenever businesses would like.

The applications are implemented to be scalable, usable, and secure. It is built using modular code, ensuring that the business can easily expand their inventory cloud-based system as they grow. The cloud-first design means that the application will be accessible on a variety of devices and will simplify future application growth for the business.

While the current implementation excludes sales transaction tracking, it lays the groundwork for integration with external point-of-sale systems. The application provides a low investment option and an extensible inventory management system for small businesses interested in digitising their inventory management.

## **Introduction**

Inventory management is an important determinant in the operational success, efficiency, and competitive advantage of businesses today. Small businesses have operational challenges which can be exacerbated by a lack of resources, technical skills and reliance on outdated inventory practices such as keeping manual records or employing a spreadsheet application which can be prone to human error, lack real-time visibility, and struggle to scale as businesses grow. Furthermore, existing inventory management software can be expensive, inflexible, and demanding in terms of technical knowledge, creating a barrier for many small organisations.Recognising this gap, this project proposes a cloud-based inventory management application to provide small businesses without existing IT infrastructure a fast, real-time solution with visibility and control of their inventory management. In particular, the web application will be a secure, scalable, and real-time inventory management system built with a React.js front-end and Flask back-end which will use Firebase for authentication and Firestore as a real-time database. The application will allow users to view, create, read update and delete (CRUD) inventory documents on any device connected to the internet. This improved inventory management process was built primarily with the user experience in mind while maintaining strong security standards using role-based access control (RBAC) and secure authentication navigational flows.Key features of the platform include real-time inventory stock updates through Firestore listeners, bulk CSV file uploads to manage stock for all items, role-based user access, and the ability to produce customisable reports using Chart.js visualisations. This style of operating may help small business owners maintain inventory stock levels to operate efficiently and effectively while being able to anticipate their operational needs. More importantly, to facilitate data driven intellectual property they investigate or propose solutions to inventory management issues without having to have advanced technical skills.Our goal in focusing on an inventory management application, with real time tracking, secure data handling and easy to navigate interfaces is to modernise and create contemporary standards of inventory management practices for small businesses. As a cloud-based solution, we recognise that remote working and scalability is becoming increasingly necessary for small businesses and this cloud infrastructure creates a platform for small businesses to educate their employees and future proof their business without costly infrastructure investment. Overall, the application will allow small businesses accessibly develop operational efficiencies and competitiveness that promotes higher strategic decision making in a digital age.

### **1.1 Aims**

This project aims to deliver real time insights and optimised inventory management to minimise stock variances, reduce operational delays and provide the business owners to make data driven decisions. Secondly, the system will enhance the ability for business to be agile and the owners to proactively respond to market needs while keeping stock at optimal levels. Furthermore, it tries to build a decision-making process using interactive dashboards and meaningful reports to present decisive inventory data. This project intends to make the system still easy looking at modern web technologies, secure and scalable.

The key areas include:

* Making inventory tracking easier for non-technical users through a straightforward interface.
* Using cloud technology to ensure secure, real-time data synchronisation and remote access.
* Providing scalability to accommodate business growth with larger inventories and more users.
* Introducing flexible reporting tools that adjust to changing business demands, supporting strategic planning.
* Implementing robust authentication measures to protect sensitive business information.

### **1.2 Objectives**

The project goals are established to make sure they are measurable and practically achievable:

* Implement a secure, role-based user authentication system (RBAC here Admin, Manager, or Staff have separate access to the system features. As a result, security is improved by granting secure access and restricting system functionality to user types. This limits the number of people who can perform important actions and reduces the risk of a data breach due to the complexity of keeping track of individuals and accounts.
* Provide a clear, responsive UI, using React and Bootstrap, to create a simple and fast experience to reduce the onboarding timeframe and improve efficiency for non-technical users.
* Use Firebase real-time updates, to update inventory levels to make sure the stock level is instantaneously synced to all user devices. This is necessary to ensure an accurate representation of stock is available. Hence, all users will all have the same stock or inventory information, regardless of location.
* Implement advanced data analytics using Chart.js, to provide interactive dashboards to visualise inventory levels over time, sales performance, and items running low on stock so elements of proactive decision-making are possible. Visualisation will assist business owners in making critical decisions.
* Deploy a scalable, cloud-based architecture ensuring the system can handle increased usage and larger datasets through platforms like Firebase. This scalability is essential as businesses expand their product lines and user base.
* Provide customisable and exportable reporting functionalities allowing users to export reports in formats such as CSV, Excel and PDF. These generate unique reports to meet their own business objectives. For data analysis and reporting, this functionality will assist in the ease of information preparation for internal and external purposes.

## **Survey of Literature/ Information Sources**

Small and medium-sized enterprises (SMEs) frequently encounter persistent structural challenges that hinder their operational effectiveness and competitiveness. Kittisak [1] discusses how SMEs typically face limited financial resources, inadequate management knowledge, and underdeveloped IT infrastructures, as originally identified by Buyong et al. These constraints often result in unclear task distribution among employees, leading to inefficiencies such as multitasking and disorganised workflows. In the context of inventory management, such systemic limitations contribute to inventory inaccuracies, stockouts, and delayed restocking processes. Recognising these barriers, the current system design focuses on minimising complexity, automating key inventory processes, and providing a real-time database backend to support SMEs operating under resource constraints.

**Challenges of Manual Inventory Management**

Many small businesses still use manual inventory management because they think using software will be too expensive and complicated. Mat et al. [2] found that SMEs often believe inventory software costs too much and are not aware of how much it could help their business. Their study also showed that small and micro businesses usually focus more on marketing and running their shop, rather than setting up proper inventory controls. Many small businesses also do not have trained staff who know how to manage stock properly, such as calculating stock levels or keeping inventory records. Because of this, small businesses often face problems like stock mistakes, waste, and delays. To help solve these problems, the system developed in this project is designed to be low-cost, easy to use, and simple to set up, making it suitable even for businesses without technical experience.

**Benefits of Real-Time Inventory Systems**

Real-time inventory management systems have demonstrated significant benefits for small and medium-sized enterprises (SMEs) in terms of operational efficiency, stock control, and cost reduction. Ugbebor et al. [3] conducted a comprehensive review on IoT-integrated inventory management systems, highlighting that real-time monitoring and automated stock control improved inventory accuracy by 25–35%, reduced carrying costs by 20–30%, and decreased stockout incidents by up to 45%. The research also reported that cloud-based platforms enhanced inventory visibility and supported better working capital management. These findings reinforce the importance of implementing real-time, cloud-based inventory systems for SMEs, aligning with the present project's adoption of Firebase Firestore real-time listeners to provide immediate inventory updates and accurate stock visibility for small business operations.

**Cloud-Based Solutions for SMEs**

Cloud computing offers small businesses a way to access powerful software services without needing to invest heavily in hardware, software, or IT teams. Otuonye [4] explains that cloud-based enterprise solutions allow SMEs to automate their operations, reduce costs, and gain real-time visibility into their business activities. By using cloud services, SMEs can avoid the need to purchase expensive servers and can instead focus on running their business through simple internet access. The cloud provider handles infrastructure maintenance, which helps lower operational costs and technical barriers. In line with these findings, the system developed in this project uses a cloud-based backend with Firebase to deliver real-time inventory management while keeping the system affordable and easy to maintain for small businesses.

**Usability for Non-Technical Users**

Usability is an important factor that influences whether small and medium-sized enterprises (SMEs) successfully adopt new technologies. Scholtz [5] found that when systems are easy to use, SMEs are more likely to accept and use them regularly. The study showed that simple and understandable interfaces make employees more confident and reduce the time needed to learn new systems. Without good usability, small businesses often hesitate to adopt technology, fearing it will be too complicated. Based on these findings, the inventory system developed in this project was designed with a clear, simple, and responsive interface, using React.js and Bootstrap, to help non-technical users easily manage inventory without feeling overwhelmed.

**Security and Role-Based Access (RBAC) Importance**

Security is very important when designing systems that manage sensitive data, like inventory and user information. Bindiganavale and Ouyang [6] explain that Role-Based Access Control (RBAC) is a powerful way to manage security in enterprise applications by limiting user permissions based on their role. Instead of giving all users full access, RBAC ensures that users can only perform actions that match their responsibilities. This reduces the risk of accidental mistakes or security breaches. In this project, RBAC is used to create different access levels for Admin, Manager, and Staff users, helping to protect important operations like user management and inventory control.

## **Design and Implementation**

With consideration to the main goals of the project, simplicity, accessibility, scalability and real time functionality, the selected technologies and architectural structure were picked carefully to satisfy these requirements.The system is divided into three main parts: the frontend (implemented in React.js), the backend (written in Flask and as a RESTful API), and cloud hosted data storage and authentication service provided by Firebase.

Frontend is developed with a clean, simple interface, letting anyone without technical knowledge to work with it easily to efficiently manage inventory items, visualise analytical information and produce reports that can lead to proactive decisions for the business. The backend is responsible for handling data validation, business logic as well as communication between the frontend and Firebase. The database and authentication layer of the database supports real time data synchronising over user devices, user credential protection and scalability.

Following this, in the subsequent sections I explain the architecture and interactions between these components along with a comprehensive explanation of my decisions for choosing these technologies.

### **3.1 System Architecture**

The inventory management system adopts the Model-View-Controller (MVC) architecture with three distinct layers: frontend serves as display (view), backend handles control (controller) as well as database serves as repository (model). This specific architectural choice was made because this design split the components making it easier to be independently developed and diagnosed. The architectural structure is illustrated in **Figure 3.1.1**.

A diagram of a computer server

AI-generated content may be incorrect.

***Figure 3.1.1: System Architecture Overview***

This diagram displays inventory management system core operations which demonstrate how React accesses Flask backend authentication through secure APIs that interface with Firebase real-time database and administrative API. The Real-time updates are possible because the frontend uses the Firebase SDK to trigger UI changes immediately after data modification.

#### 3.1.1 Interactions Between Components

The interactions between these components, previously shown in **Figure 3.1.1**, are described in detail below:

**Frontend (React) and Backend (Flask)**

The React frontend functions as the user-facing layer, enabling users to interact with the system through intuitive interfaces. It includes components for user authentication (login/signup), inventory management, user management, dashboards and report generation.After the user performs an action, such as logging in or submitting an inventory update form, React uses Axios to send an asynchronous HTTP requests to the Flask backend. The data exchange is lightweight, readable, and easy to handle because of the structure of these requests using JSON.The database is never read directly by React. It communicates exclusively through the Flask API, bringing all business logic under one umbrella, reducing surface area to potential client-side abuse, as well as enabling us to perform better data validation as well as better audit logging. This separates data validation, business rules, and access control into the backend so there can be security, maintainability, and consistency.

**Backend (Flask) and Firebase (Firestore + Authentication)**

The backend acts as the intermediary logic and validation mechanism that links the frontend to Firebase services. The backend receives frontend requests to execute several functions:

* The backend checks incoming data to confirm it makes logical sense (e.g. it stops negative quantity values).
* Authenticates users using Firebase Authentication.
* Processes CRUD operations for inventory and user data via the Firebase Admin SDK.

NoSQL document-based Firestore provides an effective structure for inventory records through its collection and document system which suits flexible yet structured data management. The Flask framework uses Firestore database for creating new items and updating stock records and logging activities while fetching data for reporting purposes. Each of our top-level documents (companies) within the Firestore database stands for a hierarchical structure where company has subcollections: users, and inventory. Metadata, like the role, email and status are stored in the user document and the details about product like name, quantity, price and timestamp are stored in inventory document.

The system uses JWT (JSON Web Tokens) to deliver authentication services that operate without state. Once a user successfully logs in, Flask generates a token that the frontend application saves locally and uses in further requests. The backend uses this to authenticate users while also verifying their roles for implementing Role-Based Access Control (RBAC). For example, when a manager logs in, they are unable to view any routes about user management and can only see features related to inventory. Both on the frontend (through conditional rendering) and backend (through JWT claims validation), ensuring that users only access the functionalities permitted by their roles.

**Frontend (React) and Firebase (Real-Time Updates)**

The real-time update functionality of this system is enabled by Firebase's WebSocket-like infrastructure. The React frontend subscribes to Firestore document modifications using real-time listeners, particularly on pages such as the dashboard and inventory list.Data modifications in Firestore by the backend automatically get transmitted to every client in real time. This ensures that:

* The inventory holds constantly updated information.
* The interface updates automatically.
* Teams that work at the same time benefit from a live collaborative interface.

Real-time behaviour speeds up decision-making and decreases operational delays which proves highly beneficial for small businesses that depend on accurate stock information.

### **3.2 Frontend Design**

The Frontend was designed with a strong emphasis on usability, accessibility, responsiveness, and clarity. Recognising that small businesses usually do not have a dedicated technical team; the aim was to create an interface that is both efficient and easy to use by non-technical people. Using React.js, the frontend leverages a component-based architecture to get modularity, code reuse and maintainability. Axios is used to facilitate API communication due to its support for interceptors and automatic JSON transformation, which simplifies error handling and token-based authentication.

In this section, I run through the structure of the frontend, justifications for the technologies and design principles I have practised.

#### 3.2.1 Technology Justification

Building scalable and maintainable web applications requires choosing frameworks that support modular design and efficient performance. Mansour [7] highlights that frameworks, like React.js, offer significant advantages in building scalable applications due to their component-based architecture and virtual DOM efficiency. These features allow developers to create reusable components, reducing code repetition and improving system flexibility. For small businesses, a scalable frontend is important to allow easy expansion without major system redesign. In this project, React.js was selected to build the frontend because it provides a lightweight, modular, and responsive interface suited for SMEs requiring simple yet powerful web applications.

In modern web applications, communicating efficiently with the backend is critical. Axios, a popular HTTP client for JavaScript, simplifies this process compared to the native Fetch API. According to apidog.com [8], Axios offers built-in features such as automatic JSON data parsing, better error handling, request timeout support, and simpler syntax for sending asynchronous HTTP requests. These improvements help developers avoid common problems like handling nested promises or manually parsing responses. Axios also supports request and response interceptors, making it easier to manage authentication tokens and error messages globally in an application. In this project, Axios was integrated into the React.js frontend to streamline API communication with the Flask backend. By using Axios, the system ensures fast, reliable data exchange between the client and server, improving the overall performance and responsiveness of the inventory management platform.

#### 3.2.2 Structure of React Components

The React frontend is modular, and each functional unit of the UI is divided into components. As the name implies, this component driven architecture splits development into several relatively small and easily managed components, creating a codebase that is easier to scale and debug.

**Authentication Page**

The system supports Email/Password as well as Google Sign-In through a single component which conditionally renders the login or sign-up form depending on the user’s action. Error handling and input validation are performed locally using React state hooks.

**Figure 3.2.1** illustrates the user registration interface, where new users input their name, company name, email, and password to create an account.

When signing up with email and password, we first create the user account using Firebase Authentication and then send out an email verification link throughout the system. Before the user can log in, they must verify their email. The system will return an error message to the user if an initial login is attempted before the verification process is complete and inform the user to verify first.

**Figure 3.2.2** shows the login interface, which includes both email/password login and the option to authenticate using a Google account.

After a successful login or sign up, the backend will issue a JWT token which will be stored inside the browsers local storage to make future authenticated API request. This token allows for a role-based access control and session persistence.

A screenshot of a login form

AI-generated content may be incorrect.

***Figure 3.2.1: Sign-Up Interface (Email/Password Registration)***

A screenshot of a login form

AI-generated content may be incorrect.

***Figure 3.2.2: Login Interface with Google Sign-In***

**Dashboard**

Once logged in, the dashboard acts as a central hub for all users. While the dashboard itself remains consistent across roles, navigation and access to certain pages are restricted based on user role. This role-based behaviour is managed using React Context, which makes authentication data available across all components and distributes role, UID, and company information based on the authenticated user.

As shown in **Figure 3.2.3**, the dashboard presents accessible navigation links to inventory, analytics, task management, and user settings.

All features are available to the admin including user management and advanced reporting. However, Managers and Staff cannot access administrative routes such as user management. This is enforced on the front end using React Router, conditional rendering and on the backend by JWT role validation. This approach improves security and user experience, ensuring that authorised users only see features relevant to their role, reducing access errors and UI clutter.

A screenshot of a dashboard

AI-generated content may be incorrect.

***Figure 3.2.3: Admin Dashboard Overview with Navigation*** *Options*

**Inventory Management Page**

The Inventory Management page is a core operational interface which is available and accessible to Admin and Managers only. This area is intentionally restricted from being accessible by staff users to avoid unauthorised changes of critical stock data. This page is built around the InventoryList component that fetches inventory data from the backend and displays them in a responsive, searchable, and sortable table. As shown in **Figure 3.2.4**, the interface allows users to view current inventory with key item details clearly displayed.

A Real time listener in Firestore monitors any updates, whether from current user or others, are reflected instantly, without any manual refreshes. This real-time behaviour supports concurrent collaboration among team members.

Stock modifications are completed with a reusable InventoryForm component presented in a modal popup. This form provides consistent logic for either creating a new item and modifying an existing item with client-side validation to ensure no invalid data is entered, such as negative stock values, or poorly formatted fields. The CSVUpload component allows Admin and Managers to bulk upload inventory records in a single operation, with appropriate built-in validation on the structure and contents of the file to reduce errors.The layout is designed using Bootstrap to ensure responsiveness across devices, providing a better user experience. The use of successful and error indicators ensure that the inventory management functionality is clear and usable even to inexperienced technical users.

A screenshot of a inventory management

AI-generated content may be incorrect.

***Figure 3.2.4: Inventory Management Page with Table View (Admin/Manager Only)***

**Manage Users Page**

The Manage Users page is an Admin-exclusive page, designed to give full visibility and control over the users registered under the same company. Since there is only one Admin per company by design, this is the central place from where we manage all the user roles and responsibilities. It displays some of the user’s data, i.e. name, email, role, and status and offers role modification actions, as shown in **Figure 3.2.5**.

Admins can promote users to Managers, demote them to Staff, or remove them from the company altogether. To ensure deliberate changes, each action requires password confirmation before proceeding, enhancing system security. These actions are validated and processed through protected backend routes and are immediately reflected on the frontend using Firestore real-time listeners.

This provides organisational flexibility without sacrificing control. It is especially useful for small business owners concerned with delivering system access and role distribution without removing the ownership of these aspects. The provision of this fine-grained control provides support for the broader RBAC system strategy used in the system’s secure design.

A screenshot of a computer

AI-generated content may be incorrect.

***Figure 3.2.5: Manage Users Interface – Admin Role-Based Access Controls***

**Reports and Analytics Page – Admin and Manager only**

Small businesses can benefit from this page, which converts raw inventory data into specific insights to provide meaningful reporting without technical complexities. As seen in **Figure 3.2.6**, the page features filter options, such as custom date ranges, and a clean layout that guides users through selecting the report type. On this page Admins and Managers receive real-time inventory alerts which display as both graphical visualisations and tabular reports for better decisions.

Users can apply either predefined date parameters or established ranges before selecting from among available reports that cover inventory actions and sales trend monitoring. The frontend generates multiple Axios calls to the Flask backend after receiving a request to query Firebase Firestore and perform aggregation calculations to return the filtered datasets to the frontend. The tables, shown in **Figure 3.2.7**, present structured information along with item-level monitoring that shows which users altered or updated products along with quantity fluctuations and price changes.

The page also combines raw data with interactive charts that function through Chart.js. The application shows time-based quantity data using line graph elements simultaneously as bar chart visualisation displays popular items, as illustrated in **Figure 3.2.8**. These graphical displays help users quickly understand business performance patterns, identify stock alerts, and detect sudden spikes in sales data without manual data analysis.

Data export options such as CSV, Excel and PDF allow users to retrieve their data and includes both charts and summary information. The page features real-time low-stock alerts while presenting comprehensive reporting capabilities that upholds system principles of operational intelligence and clarity and usability.

A screenshot of a computer

AI-generated content may be incorrect.

***Figure 3.2.6: Reports Page – Filter Options and Date Range Selection***

A screenshot of a computer

AI-generated content may be incorrect.

***Figure 3.2.7: Reports Page – Structured Tables with Item-Level Tracking***

A screenshot of a computer

AI-generated content may be incorrect.

***Figure 3.2.8: Reports Page – Chart.js Visualizations (Trends and Top Products)***

**Tasks and Notifications Page**

The system has a Tasks feature that users can access through a lightweight interface. The tasks feature stores its data in Firestore to provide basic organisation for instructions and reminders.

Figure 3.2.9 shows the Tasks & Notifications interface, which is available for all authenticated user accounts and shows user-specific tasks as well as system-generated notifications like low-stock notifications, promoting transparency and team awareness.

Only Admins and Managers can create new tasks. A task creation form is displayed within the same page, allowing the Admins and Managers to input task parameters and assign roles to the team. Figure 3.2.10 shows the form for creating a new task. Staff users do not see this form and have visibility on the assigned tasks and the system notifications only.

A screenshot of a computer

AI-generated content may be incorrect.

***Figure 3.2.9: Tasks and Notifications Page – Staff View (Read-Only Access)***

A screenshot of a computer

AI-generated content may be incorrect.

***Figure 3.2.10: Tasks Page – Task Creation Form (Admin and Manager Only)***

#### 3.2.3 Design Principles

The system follows key design principles such as:

* **Clarity:** The interface becomes easy for first-time users to understand because it combines visual hierarchy with a uniform layout design.
* **Responsiveness:** The application achieves cross-device compatibility through its development with Bootstrap and CSS flex/grid layouts.
* **Consistency:** The application presents similar button positions combined with standard forms and coordinated colours throughout all sections.
* **Feedback:** When users perform form submissions along with logins and inventory changes, they receive instant feedback about outcome success or failure.
* **Accessibility:** The application remains accessible to assistive technology because semantic HTML tags work together with appropriate aria-labels (Accessible Rich Internet Applications).

### **3.3 Backend Implementation**

The backend is developed using Flask because it functions as a lightweight modular Python microframework that handles both basic design requirements and Python system environment integration (.env files). Core operations reside within the backend which enforces authentication protocols while maintaining control over data access rights and validation procedures. The backend infrastructure contains server-based operations for roles as well as inventory management and CSV handling, and keeps these processes safely confined within the server environment.

In this section, I describe the backend implementation, justifications for the technologies and security practices.

#### 3.3.1 Technology Justification

Flask is a lightweight Python web framework that enables small to moderately sized applications to be built with little overhead. Ghimire [9] indicated that unlike a full-stack framework like Django, Flask enables developers to use components at their discretion given the increased flexibility. Flask's ability to have bare minimum components is well-suited for projects where speed of development, customisation, and simplicity are a consideration. For small business, a lightweight backend will reduce complexity and hosting costs. In this project, Flask was selected to implement the RESTful API backend, providing the basic structure for routing, user and permission authorisation and secure communication without unnecessary overhead. The lightweight aspect of Flask ensures that the overall system will remain more maintainable and extensible, moving forward.

Authentication is an important aspect of web application security, and the chosen implementation will affect optimal performance, and scalability of the system. Balaj et al. [10] suggest that token-based authentication, such as JWT (JSON Web Token), has multiple advantages over traditional session-based authentication methods. The key feature of JWT over session-based authentication is it is stateless, meaning no session storage is required on the server side, which aids scalability and reduced network server load. Tokens are self-contained; they contain all the necessary information required for the user and can be validated without a session query against a database. This project implemented token (JWT) authentication to enable secure, and scalable login access for multiple users, as well as role-based access to the application, with the added advantage of token storage on local storage of the client machine. The server can authenticate without having to query the session database to track user access. Using JWT facilitates multiple concurrent users within the backend while ensuring secure communication of sensitive data between the client and server.

#### 3.3.2 RESTful API Route Structure

The system design utilises a RESTful API architecture to divide its components into logical resources through defined endpoints. REST conventions guide the routes through predictable URL paths where HTTP methods (GET, POST, PUT and DELETE) represent individual operations. This structure ensures consistent behaviour while providing simplified frontend connections and improved maintainability levels.

For example:

* Through the endpoint **POST /api/signup** users register by completing registration and receive assigned roles and follow email verification procedures.
* Both **POST /api/login** and **/api/google-signin** authenticate users before issuing JWT tokens which serve for protected session management.
* Routes such as **GET /api/inventory** allow the management of inventory by enabling **POST /api/add-inventory**, **PUT /api/update-inventory/<item\_id>** and **DELETE /api/delete-inventory/<item\_id>** for respective CRUD functionality.
* The API route **POST /api/upload-csv** enables bulk inventory updates through structured files which benefits companies with extensive stock quantities.
* Users can access business intelligence metrics through **GET /api/reports** and **GET /api/analytics** which show sales data together with performance reports and low-stock advisory information.

The logical association between backend routes and system resources matches industry standards, therefore, simplifying testing, debugging and documentation of the backend. Each endpoint has one distinct responsibility with automatic validation systems and access control features which protect against misuse.

|  |  |  |
| --- | --- | --- |
| Endpoint | Method | Description |
| /api/signup | POST | Registers a new user, assigns role (Admin or Staff), and triggers email verification. |
| /api/login | POST | Verifies credentials and returns a JWT token on success. |
| /api/inventory | GET | Fetches all inventory items for the authenticated company. |
| /api/inventory | POST | Creates a new inventory item |
| /api/inventory/:id | PUT | Updates an existing inventory item. |
| /api/inventory/:id | DELETE | Deletes an inventory item |
| /api/reports | POST | Fetches filtered analytics data based on query parameters. |
| /api/tasks | GET | Retrieves all tasks for the current company. |
| /api/tasks | POST | Allows Admins/Managers to create new tasks. |
| /api/users | GET | Returns all users within the same company |
| /api/users/promote | POST | |  | | --- | |  |  |  | | --- | | Promotes a staff user to manager | |
| /api/users/demote | POST | Demotes a user |

**Table 3.3.1: Core Flask REST API Endpoints and Descriptions**

#### 3.3.3 JWT Authentication and Role-Based Access Control

The backend employs JWT authentication as its main access control to ensure both security and stateless sessions. After successful user login or signup, the backend produces a JWT token which contains essential user information consisting of Firebase UID and role type (admin, manager, staff), company association and expiration timestamp set at 24 hours. The token receives authentication through digital signature that uses a server-side secret known as JWT\_SECRET (using the HS256 algorithm to sign the payload, which includes user details and an expiry date, ensuring tokens are tamper-proof).

The browser stores the issued token safely in the local storage while sending it with every new request by using the Authorisation header. The JWT validation decorator on server protection routes obtains and verifies tokens in order to grant endpoint access.

An ordinary decoded payload appears as shown below:

A black background with white text

AI-generated content may be incorrect.

This implementation model removes session management because it handles transitions between distributed systems and various devices with ease.

RBAC (Role-Based Access Control) provides a system capability to enforce action restrictions for different user types in the platform. During registration the system administers user roles between admin, manager and staff groups based on the application permissions granted to each member. For example:

* User management operations along with user promotion and demotion tools exist only for admin users.
* The system allows users who have assigned roles of Manager or Admin to update inventory records and generate reports.
* All staff members encounter restrictions, prohibiting them from entering administrative sections where sensitive actions exist.

The application utilises two different areas to deploy these rules for system access:

* The frontend components use React Context together with conditional rendering to protect unauthorised functionality.
* The backend system checks JWT contents to validate user authorisation for requested operation execution.

The dual security system forms an obstacle for unauthorised data attacks regardless of frontend modifications.

#### 3.3.4 Security Practices

The system implements several security measures that authorise and authenticate approved users for access to protected data operations. These security practices defend user data and inventory records at company-level, especially in multi-tenant platforms.

Email verification serves as an initial system security measure. Users who register with email and password receive a verification link from SMTP through the Firebase Admin SDK system for account registration. Users must verify their account through email before the system allows access to the login page. System security is improved through this process because fake accounts cannot be created while the verification system ensures that only reachable legitimate users obtain access.

Role-Based Access Control (RBAC) works as an access control mechanism. The application system grants users admin, manager and staff privileges that enforce their respective access rights to specific routes with related data. The permission to access tools which control company management rests solely with admin role users who can perform functions for assigning user roles, removing users and creating system-wide reports. The system restricts access by checking role claims within JWT tokens, both in the backend and through frontend interface limitations.

Scoped data access provides essential protection to the system. Each user possesses access to a single company under which their Firestore queries must operate on subcollections belonging to that company. The system prevents users from reaching or modifying any data which belongs to businesses other than their own. The backend system uses the company name from validated JWT tokens to reject all attempts at manual API queries against information belonging to different organisations.

The expiration time of JWT tokens is set to 24 hours to provide time-limited access that requires regular refreshes. A valid JWT token is necessary for every secured endpoint request where token validation occurs before operations can continue.

In addition to access control, the backend also validates all inputs for completeness and logic errors. The system runs various checks that stop negative inventory levels and confirm the correct format of CSV documents and perform field type validation. The validation system prevents damaged requests which could either spoil the data or make the system unsteady.

The security measures form a zero-trust security model that implements the following sequence of protection:

* Access to the platform becomes available only after successful verification of users.
* Users from authorised roles alone have access to perform essential tasks within the system.
* Users can access company-specific data alone through the system but cannot reach other information.
* Every input data requires validation before it undergoes processing.

The security measures create an environment that protects the backend infrastructure while maintaining durability and strength when servicing business scenarios with multiple users.

#### 3.3.5 Core Functionalities

The system performs every operation from user account creation to role administrator functions, as well as current inventory updates and data-based reports in real-time. The business logic receives centralised management within the backend system for the purpose of maintaining both consistent data along with secure access control and ensuring data integrity.

1. User Registration and Company Admin Logic

The website implements the **/api/signup** endpoint to manage email-based password accounts. During user sign-up the backend system investigates whether the registered company exists in Firestore. When no company exists in Firestore, the backend both generates a new company entry while making the registering user an admin. Each company can only have one admin within this design model. The system assigns new users who sign up under the same company the default staff placement. When registration succeeds the Firebase Admin SDK sends a verification email to users. Log-in access is limited only to verified email users and the server enforces this requirement during the login process.

1. Google Sign-In and Fallback Handling

Through the **/api/google-signin** endpoint users can authenticate using OAuth2 procedures with their existing Google account. The backend performs immediate verification of the token when the user has an existing registration with a system company before issuing a JWT. When users attempt first-time sign-in the backend system asks for their full name and company name because no matching records exist at the time of authentication.

The system creates a new document for the specified non-existent company and makes the user become the initial administrator. The default staff member status is assigned to users who enter an existing company within the system. The fallback flow enables users to complete registration and authorisation processes correctly through which every user receives the right company and role assignments with a smooth and friendly authentication experience.

1. Inventory CRUD Operations with Upsert Logic

The inventory management endpoints of the system include **/api/inventory** and four additional routes **/api/add-inventory**, **/api/update-inventory/<id>** and **/api/delete-inventory/<id>**. The system routes empower both Admins and Managers to execute CRUD actions for handling stock items.This system utilises an upsert logic to decide between creating new entries and updating existing ones when correlating fields like name with category and supplier. The system design prevents identical products from generating multiple record entries.The backend processing of updates includes price\_diff and price\_change calculation to benefit future reports and analytics. The system implements server-side validation to detect absent data fields as well as detect negative quantity values and validate user access through JWTs to protect inventory modification capabilities.

1. Bulk CSV Upload with Validation

Large inventory uploads happen through the **/api/upload-csv** endpoint that accepts well-structured CSV and Excel file formats. A Pandas library function validates and reads data that users upload through the backend system. Data processing occurs after receiving user input because the system verifies necessary columns exist, verifies data types and establishes logical patterns before starting update processes. The system cuts down data entry duration while maintaining high-quality output.

1. Reporting and Analytics

The API provides two routes through **/api/reports** and **/api/analytics** which accept date parameters to retrieve inventory data summaries for customers. The system provides reports that show item-level histories combined with stock trends and low stock alerts and information about top-selling products. The server executes aggregations both for maximised performance and consistent analytics that remain independent from frontend modifications.

1. Email Notifications via SMTP

Through smtplib, the backend system delivers email messages when users perform essential tasks such as the verification process and password recovery and administrative actions. The SMTP credentials are secured through environment variables while email bodies create automatically to match each situation.Multiple security layers protect the core functions. Meaningful business operations such as role validation, analytics generation and bulk upload processing occur server-side using protected and authenticated procedures. A data API is responsible for serving as the main security foundation that both safeguards organisation workflow systems and establishes secure multi-user capabilities.

### **3.4 Database and Authentication**

The inventory management system uses Firebase Firestore for database operations, and authentication through Firebase Authentication. The database infrastructure relies on Firestore because it delivers scalable real-time data management alongside adaptive schemas which meet small business needs. Firebase Authentication served as the selected authentication option due to its seamless compatibility with identity providers along with simplicity and its secure nature.

In this section, I display the backend structure with visual examples and justification for the technology choices.

#### 3.4.1 Technology Justification

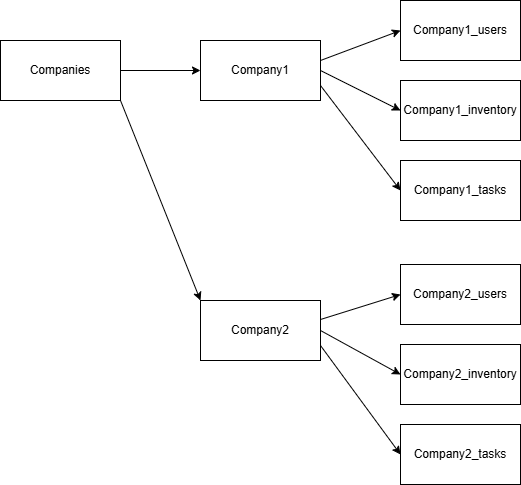
Firestore is a NoSQL serverless database developed by Google, designed to support real-time data syncing and offline capabilities across web and mobile applications. Kesavan et al. [11] describe Firestore as a scalable, developer-friendly solution that simplifies backend complexity by handling infrastructure, scaling, and data synchronisation automatically. The paper highlights Firestore's real-time listeners, which allow applications to receive immediate updates when data changes, improving responsiveness and user experience. For small businesses, these features are especially useful because they eliminate the need for manual refreshing and complex database setups. In this project, Firestore was selected to manage inventory data in real time, allowing multiple users to view and interact with up-to-date stock information instantly. Its serverless nature also reduces operational overhead, making it well suited for SMEs that require efficient yet low-maintenance data solutions.

Firebase Authentication offers a secure and user-friendly way to handle user login and registration processes in web applications. Wieruch [12] explains that Firebase Auth supports multiple sign-in methods, including email/password, Google OAuth, and other providers, while handling session management and security behind the scenes. These features make Firebase especially useful for small business applications, as it reduces the complexity of implementing secure authentication from scratch. The system developed in this project uses Firebase Authentication to enable both traditional email/password login and Google sign-in, simplifying access for end-users while maintaining security. The built-in integration with Firebase Firestore also ensures that authenticated users can securely interact with the database based on their assigned roles. This setup provides a low-maintenance, scalable, and secure foundation for managing users in a small business context.

#### 3.4.2 Database Structure

A multi-tenant hierarchical structure of Firestore controls the efficient and secure management of data in the database. Every company operates with its own root document in the primary companies’ collection, maintaining isolation for its data from all other companies. Each company has distinct subcollections within its document for saving users along with inventory items and additional data.

**Figure 3.4.1** illustrates the basic hierarchical framework:

****

***Figure 3.4.1: Firestore Structure: Companies with Nested Subcollections***

This structure provides crucial advantages. Security is ensured by the system's structure which separates individual company data from the rest of the database, therefore, protecting it from unwanted cross-company data flow. The database design enables smooth expansion of new companies and users because it permits instantaneous addition without needing schema changes.

#### 3.4.3 User Data Structure and Role Management

User data is stored at the following path:

companies/{companyId}/users/{userId}

Each document contains important information which helps identify users through their fields and manages their roles to provide system access. The essential Firestore data elements consist of the Firebase UID and email address while also including first name and last name personal information and assigned staff role type (admin, manager or staff), account status details (active or removed) and timestamp for the time of account creation.

**Figure 3.4.2** provides the composition of the User entity:

A screenshot of a computer

AI-generated content may be incorrect.

***Figure 3.4.2: Firestore User Entity Schema***

The system posts a simple, straightforward method to assign a role to users. The backend examines if a particular company exists when users first register for it. As a result of following this procedure, the registering user automatically qualifies as "admin" to receive full administrative control. A staff role automatically gets assigned to new users who join an existing company unless an administrator explicitly grants them manager privileges. Each business operates under managed distinct boundaries because role-based permissions are clearly defined.

A diagram of a system

AI-generated content may be incorrect.

***Figure 3.4.3: Role Assignment and Email Verification Signup Flow***

The figure displayed in **Figure 3.4.3** demonstrates how users get assigned either Admin or Staff roles depending on company presence during registration. The implementation showcases the SMTP email authentication steps provided by Firebase.

#### 3.4.4 Inventory and Task Data Structure and Implementation

Inventory Data Structure:

The inventory database system provides straightforward inventory management capabilities alongside transparent observational features for operational activities. Items are stored at the following path:

companies/{companyId}/inventory/{itemId}

The inventory documents require essential operational fields which include name, category, description, supplier, quantity, and price. The calculated fields in Inventory contain price\_change and price\_diff respectively to document price variations, enabling advanced analysis capabilities. The audit-related fields added\_at, updated\_at, added\_by and updated\_by create organisational transparency because they track inventory modification activities promoting accountability within the company.

With its adaptable NoSQL structure Firestore makes it possible to incorporate new fields and update existing ones easily. The database framework allows businesses to rapidly update their business process framework without any disruption to operational activities. Businesses can easily introduce item-specific data including expiration dates alongside storage requirements and handling specifications. The operational needs of small businesses find alignment with the flexible feature of this system because it enables rapid responses toward changing market requirements.

**Figure 3.4.4** displays how Firestore document’s structure inventory and tasks information. The database exhibits quantity, price, timestamps and audit information for inventory alongside priority, description, timestamps for tasks through company-defined collections.

Tasks Data Structure:

Tasks are stored at the following path:

companies/{companyId}/tasks/{taskId}

The structure of task documents contains fields that improve both task management and clear communication within teams. Essential attributes include:

* The title serves as a short descriptive label pointing out the primary concern of each task.
* The description displays detailed instructions or context necessary for task completion.
* Task priority can be established through a categorical urgency field where options include high, medium and low to enable efficient work prioritisation.
* The timestamp recorded at created\_at tracks when admin or managers generate new tasks thus enabling better follow-up and tracking of the tasks.

A screenshot of a computer

AI-generated content may be incorrect.

|  |
| --- |
|  |

|  |
| --- |
| ***Figure 3.4.4: Firestore InventoryItem and Task Entity Schema*** |

#### 3.4.5 Firebase Authentication Integration

Firebase Authentication serves as the authentication method because it offers exceptional security while remaining easy to integrate with Firebase services. Authentication through the identity management system works with two main methods: email/password and Google OAuth.

Users who sign up with an email must verify their account through a verification link delivered by SMTP which the system generates with Firebase Admin SDK. Once users finish the email verification process, they gain access while preventing attackers from making false accounts and unlocked doors for unauthorised entry.

Users experience easy access to Google OAuth sign-in because it leverages their existing Google accounts to speed up the onboarding process. Google-authenticated users who try to access the platform for the first time will need to provide further registration information since the backend will generate new company documents as needed before assigning suitable user roles according to company standing.

Users receive safe ID tokens from Firebase Authentication after completion of authentication. Before giving out tokens to users, the backend system performs verification then issues customised JWTs containing UID along with user role and company information and expiration date. The customised JWT tokens supervise authentication-related operations as well as authorisation access throughout the system.

A diagram of a software company

AI-generated content may be incorrect.

***Figure 3.4.5: JWT Authentication Sequence Flow***

A detailed visualisation found in **Figure 3.4.5** shows the progression through user login, Firebase identity check, backend JWT token creation and the deployment of tokens for subsequent API request authentication. The sequence provides both role-based access security and stateless authentication as its core functionality.

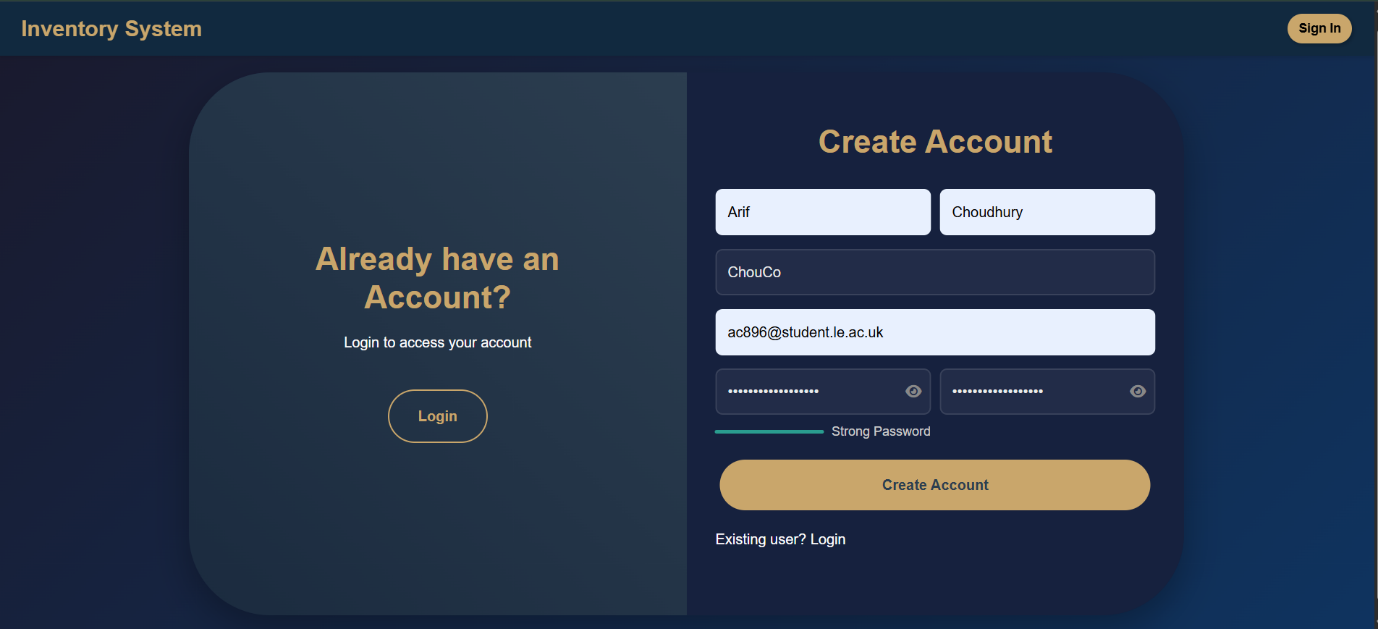
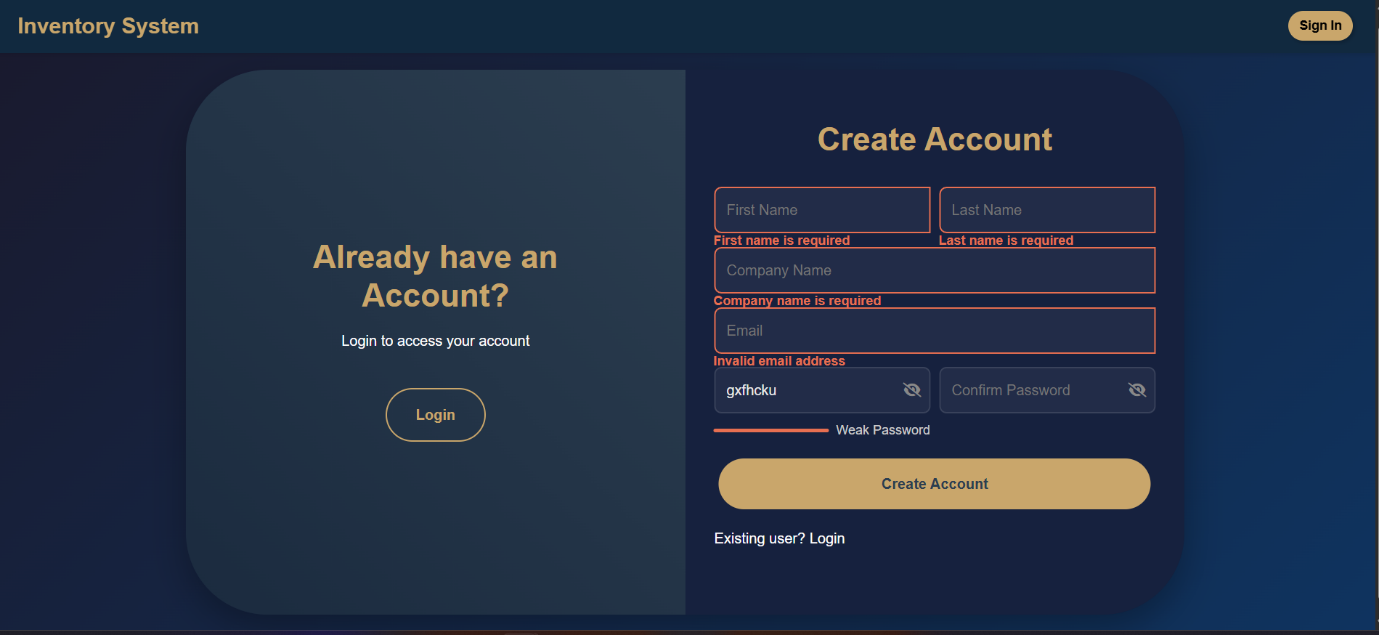
## **Results and Discussion**

The section outlines the inventory management system features that were deployed and conducts an evaluation of their operational capabilities as well as data precision and correspondence to initial project requirements. The system implements crucial functionalities including authentication protocols, access controls based on roles, inventory tracking features, reporting functionality and assignment coordination through real user scenarios. The discussion provides feedback regarding the useful nature of the implemented design choices while noting opportunities to improve its functionality.

#### **4.1 Demonstrating Core Functionalities**

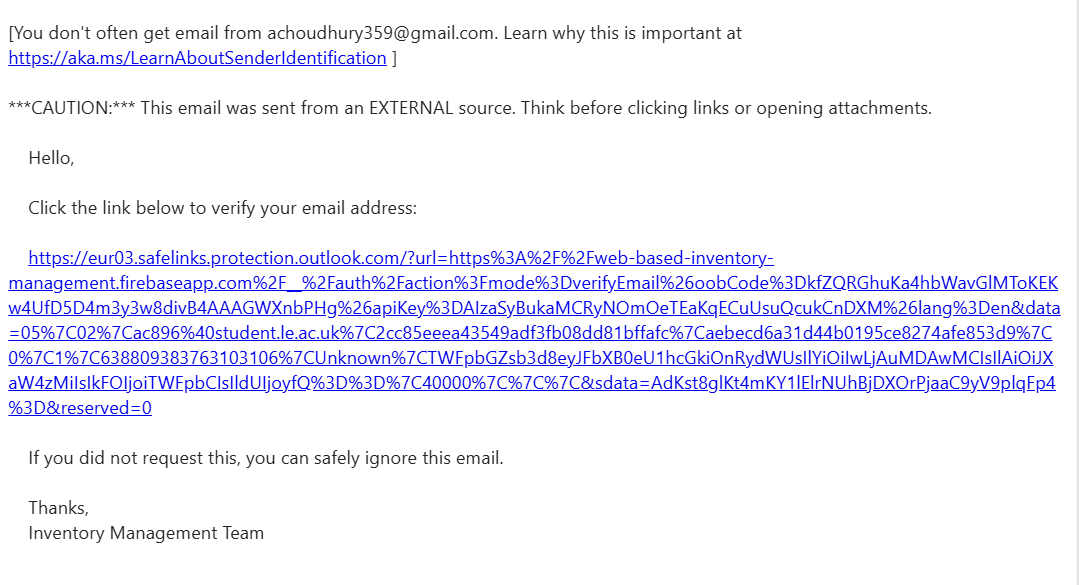
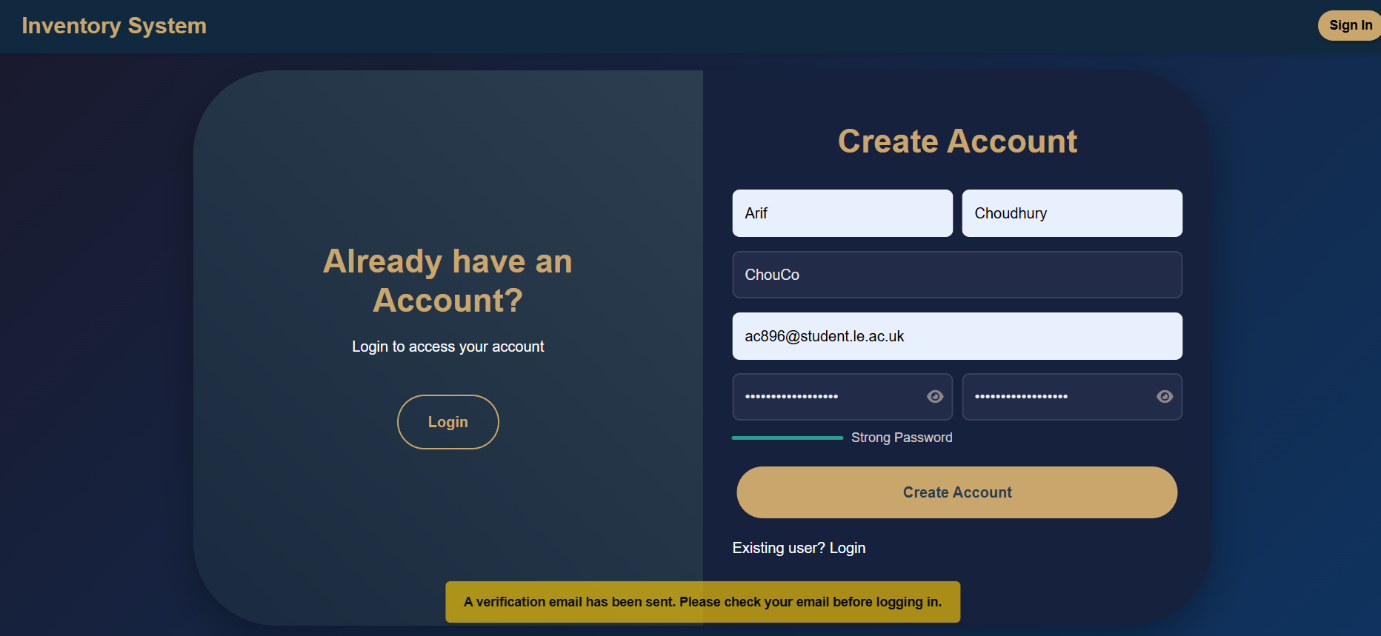
**Authentication and Registration**

Account holders can select from using either traditional form-submission authentication or they can log in with Google credentials. During registration **Figure 4.1** delivers visual error messages to verify necessary information and email address and password strength standards before users can establish their account.

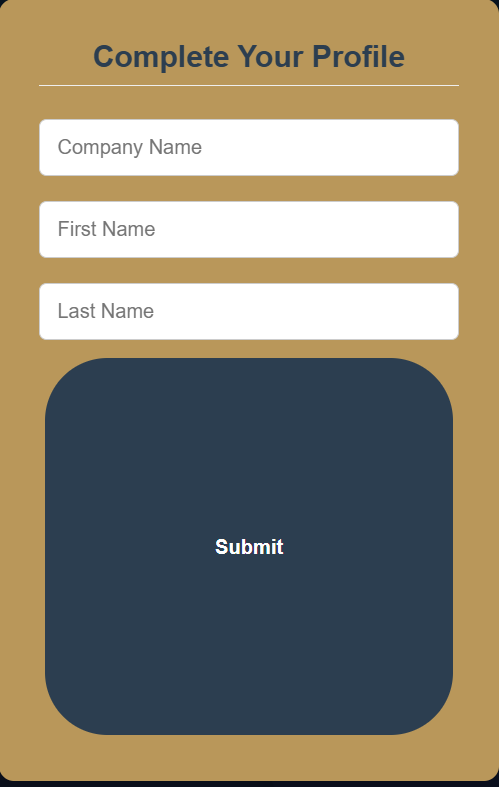
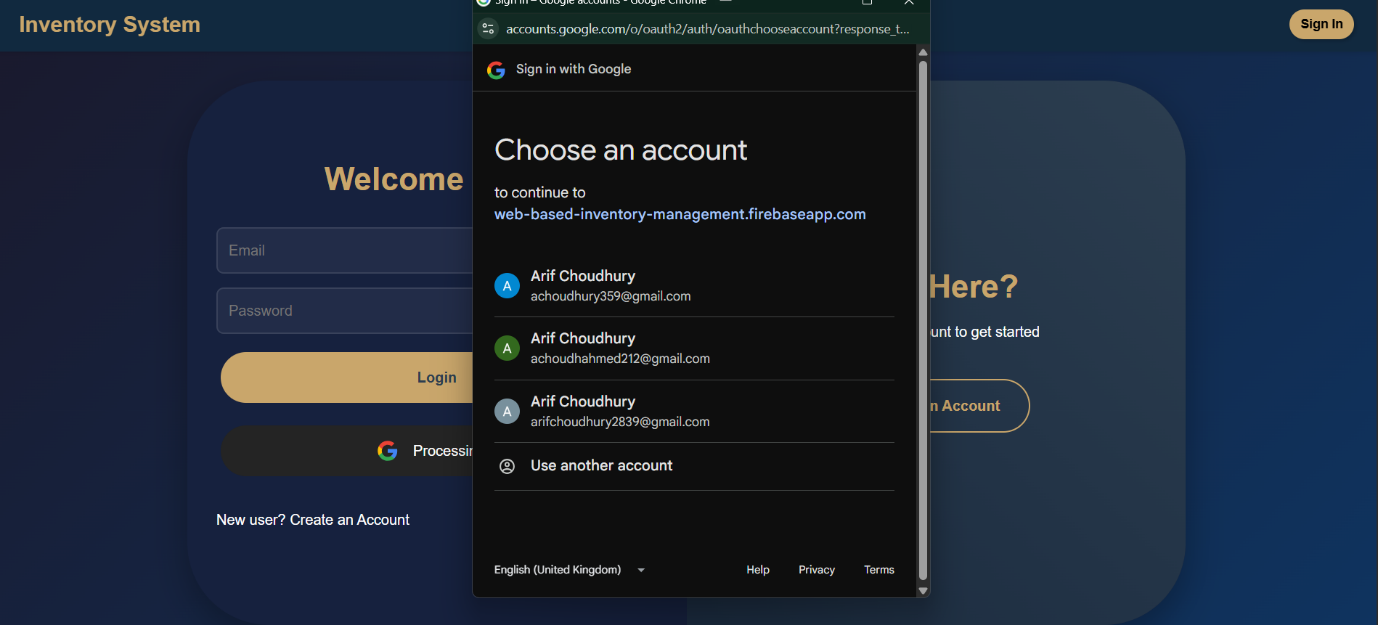


***Fig. 4.1: Sign-up form with validation***

Account verification emails sent to users upon successful registration need confirmation before an account can become active (**Figure. 4.2**).



***Fig. 4.2: Email verification required before login is allowed.***

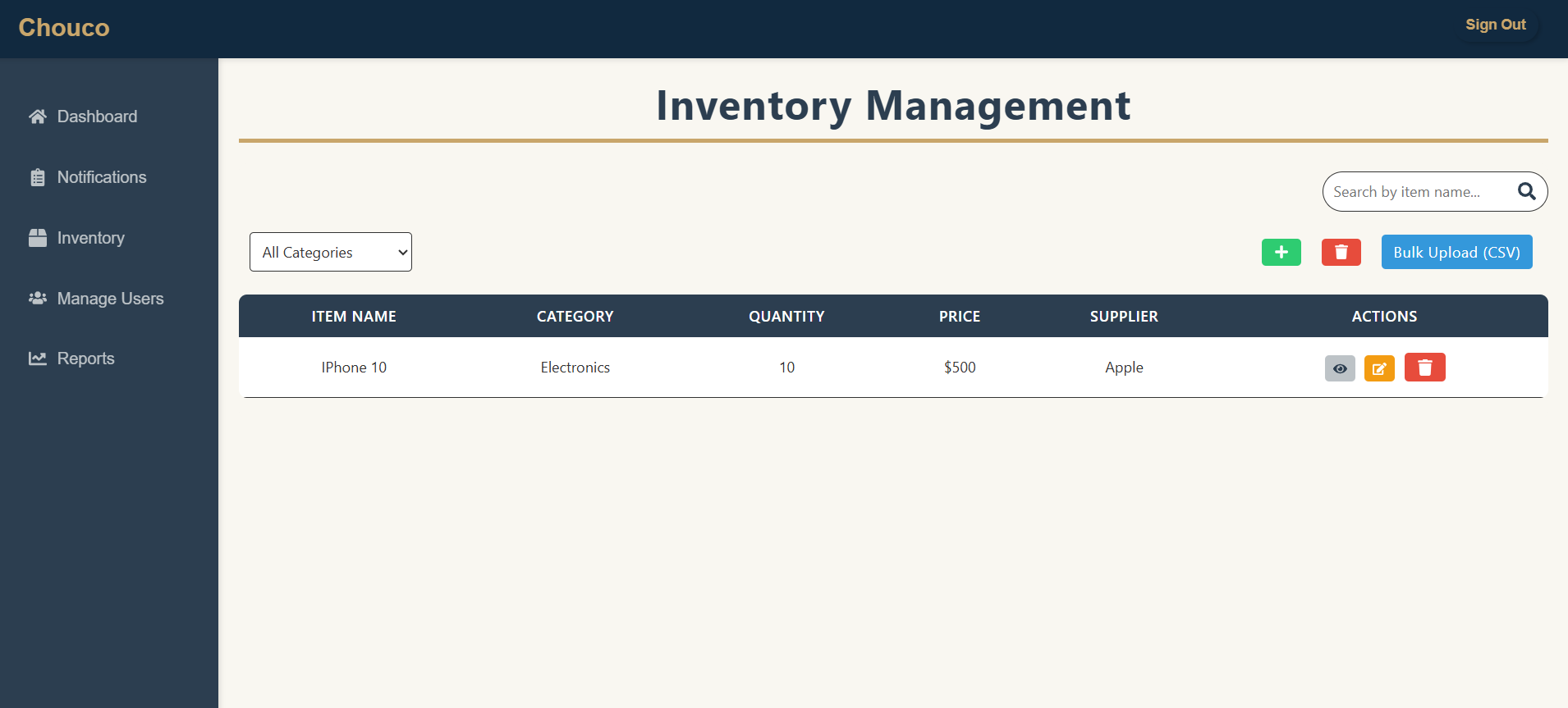
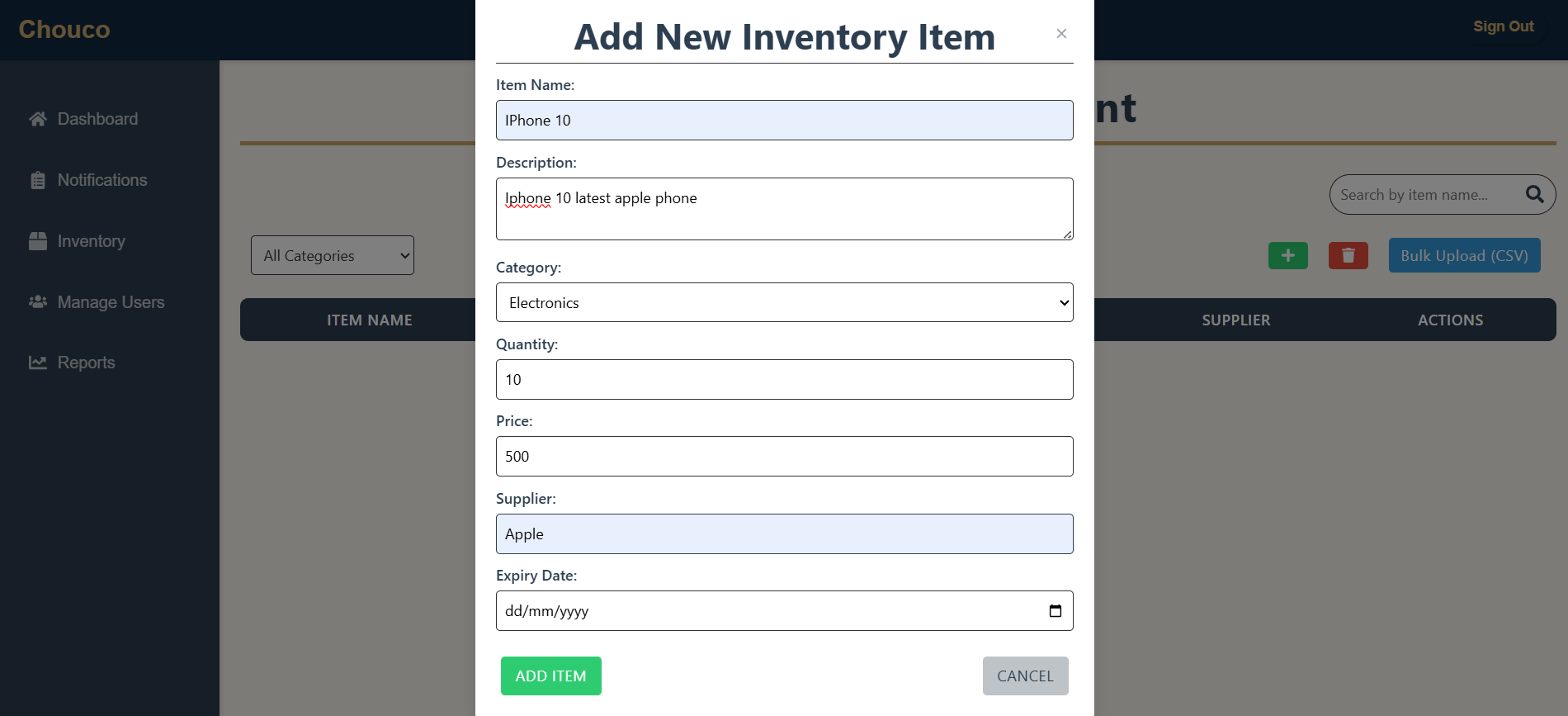
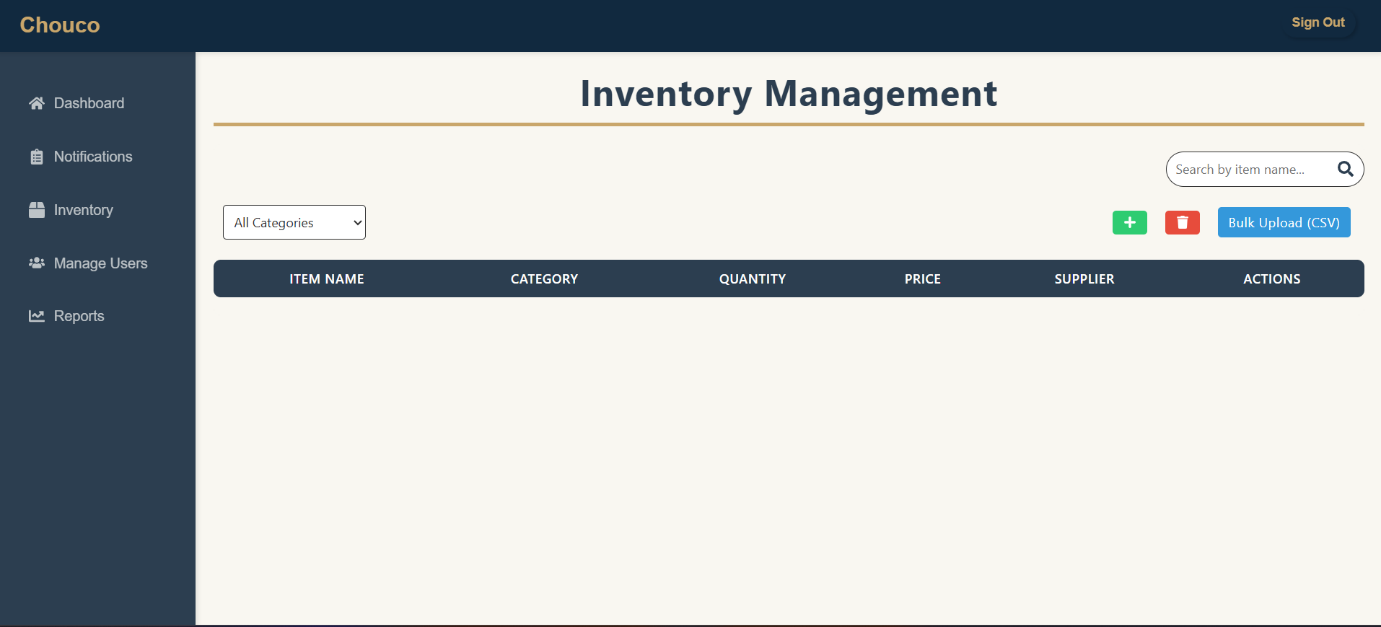


***Fig. 4.3: Additional profile information form shown to new Google users.***

Users can choose Google login when they want to access the system, and new users will need to create their company profile before continuing (**Figure. 4.3**).

**Inventory Management**

The Inventory Management page enables users to control inventory through modification or addition or removal of items because of its straightforward user interface as presented in **Figure 4.4** (see below). Initially, the table is empty. The system presents a modal window where users can add item details such as category and price and quantity and name before the data saves to the table.



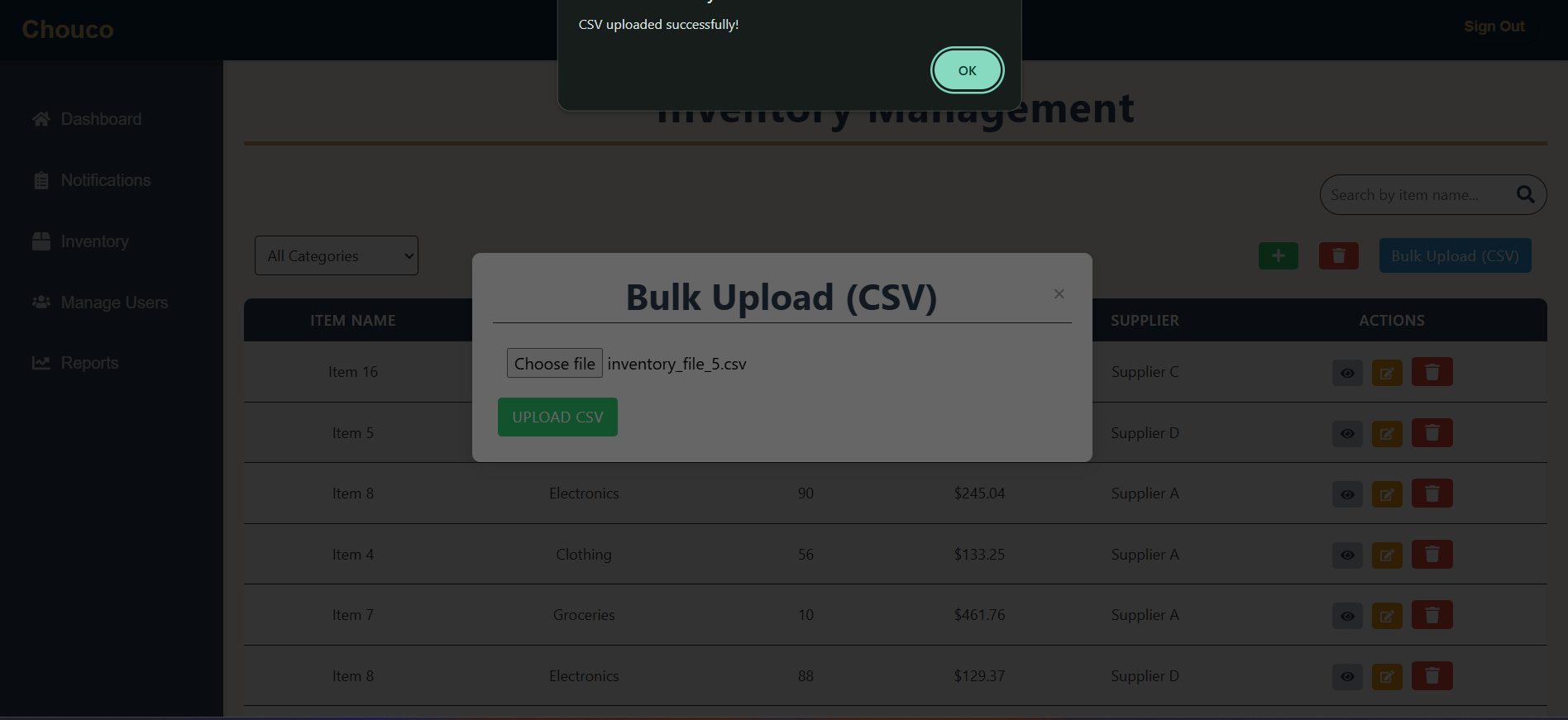
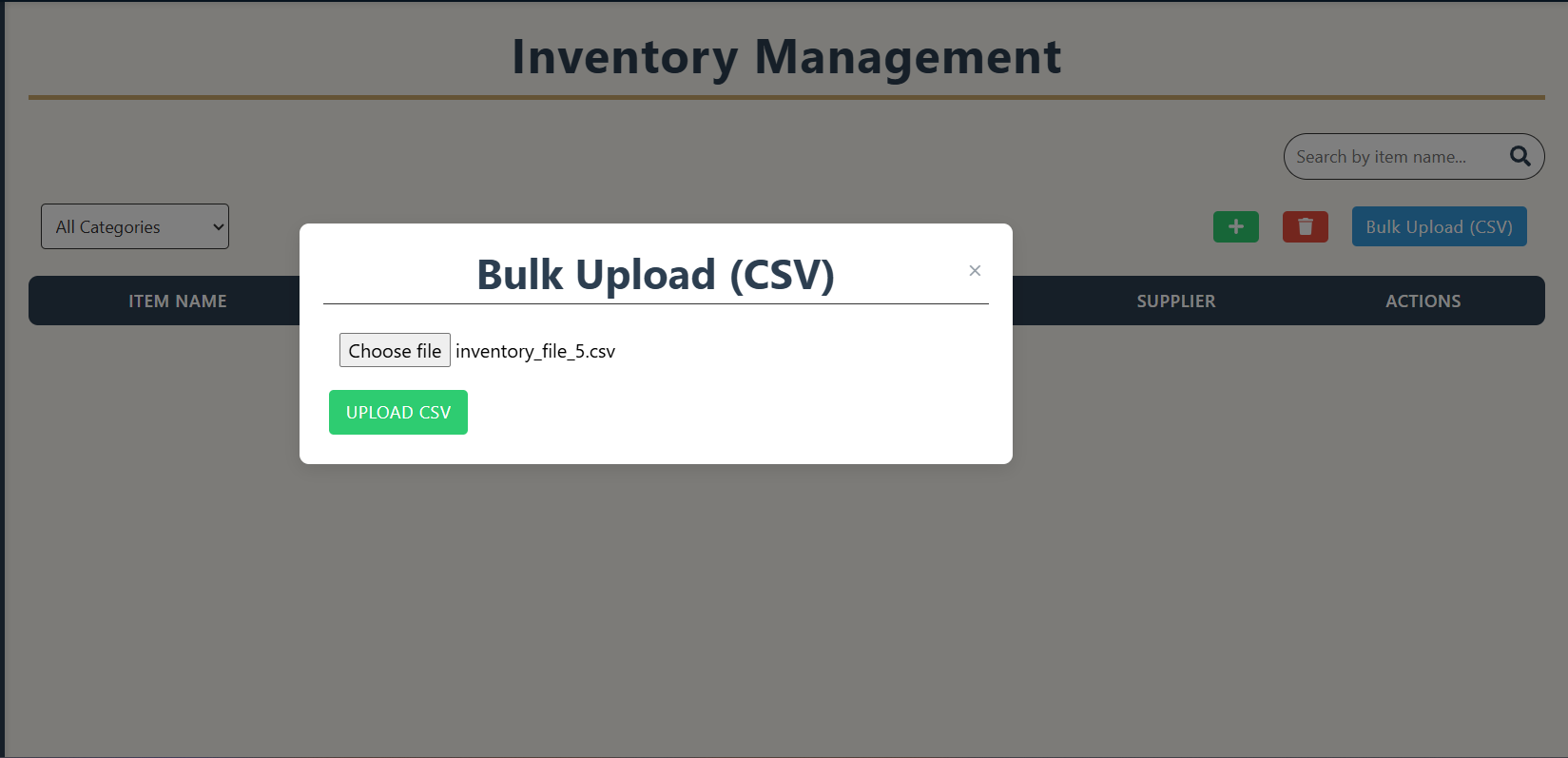
**Figure 4.4: The Inventory page shows an initially empty state before a new item is added through the modal form.**

A screenshot of a computer

AI-generated content may be incorrect.

***Figure 4.5: After adding, the item appears in the table and can be removed via a delete confirmation dialog.***

Before executing any deletion, the system requires confirmation through a prompt window (**Figure 4.5**). Once confirmed the display removes the selected item from the database while updating the table presentation.

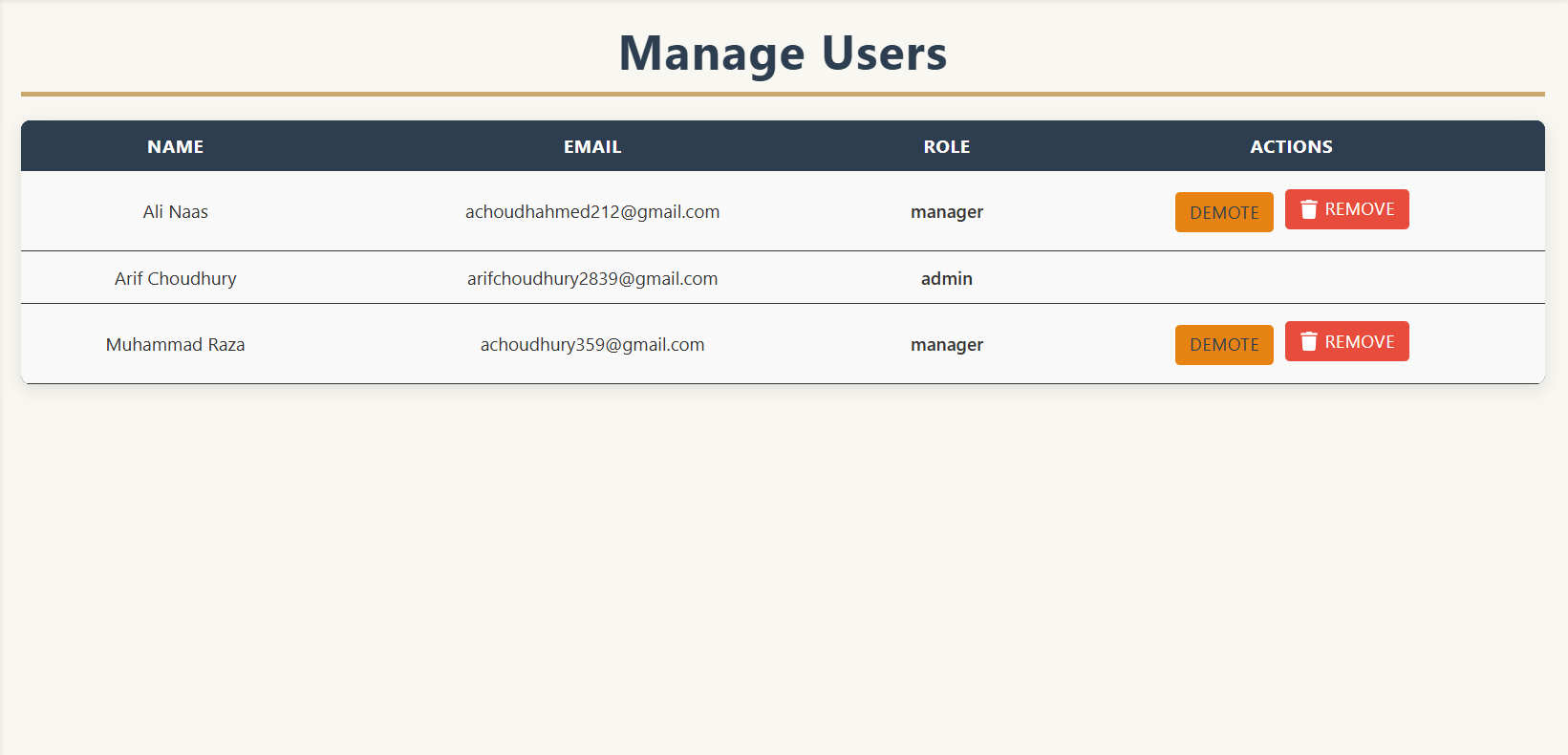
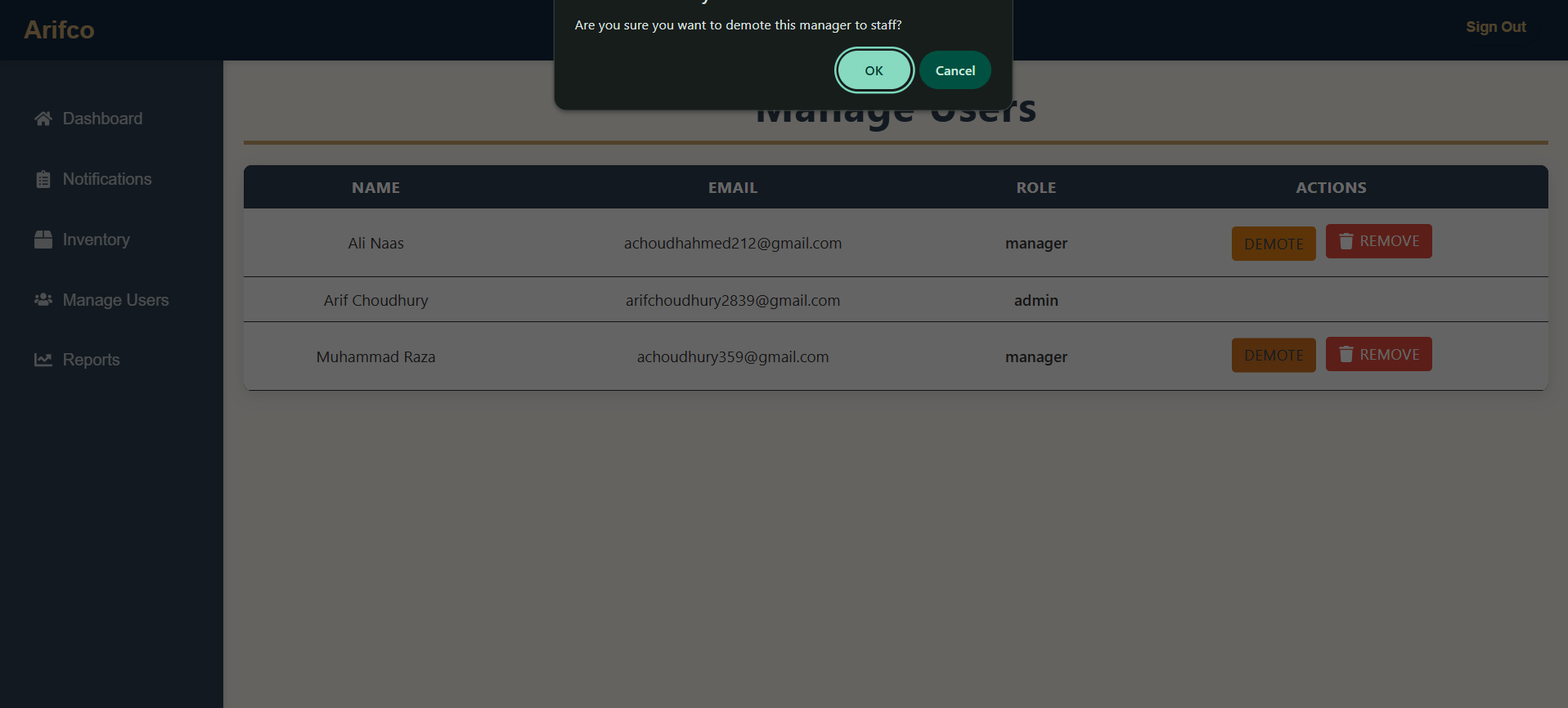
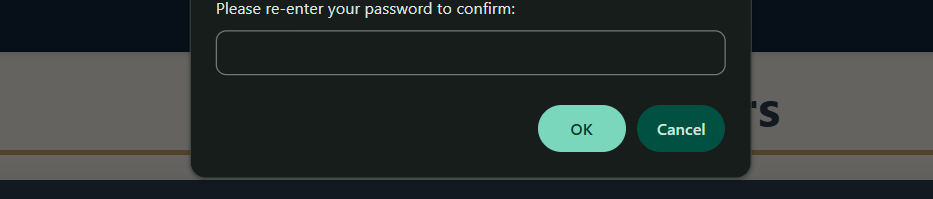


***Figure 4.6: Bulk inventory upload confirmation message after successful processing.***

Bulk Upload enables users to speed up inventory entry through the interface shown in **Figure 4.6.** A CSV file with multiple inventory entries is uploaded into the system which results in immediate generation of data from this file in the table format. The upload process finishes with a toast notification which follows by updated inventory appearing in the background.

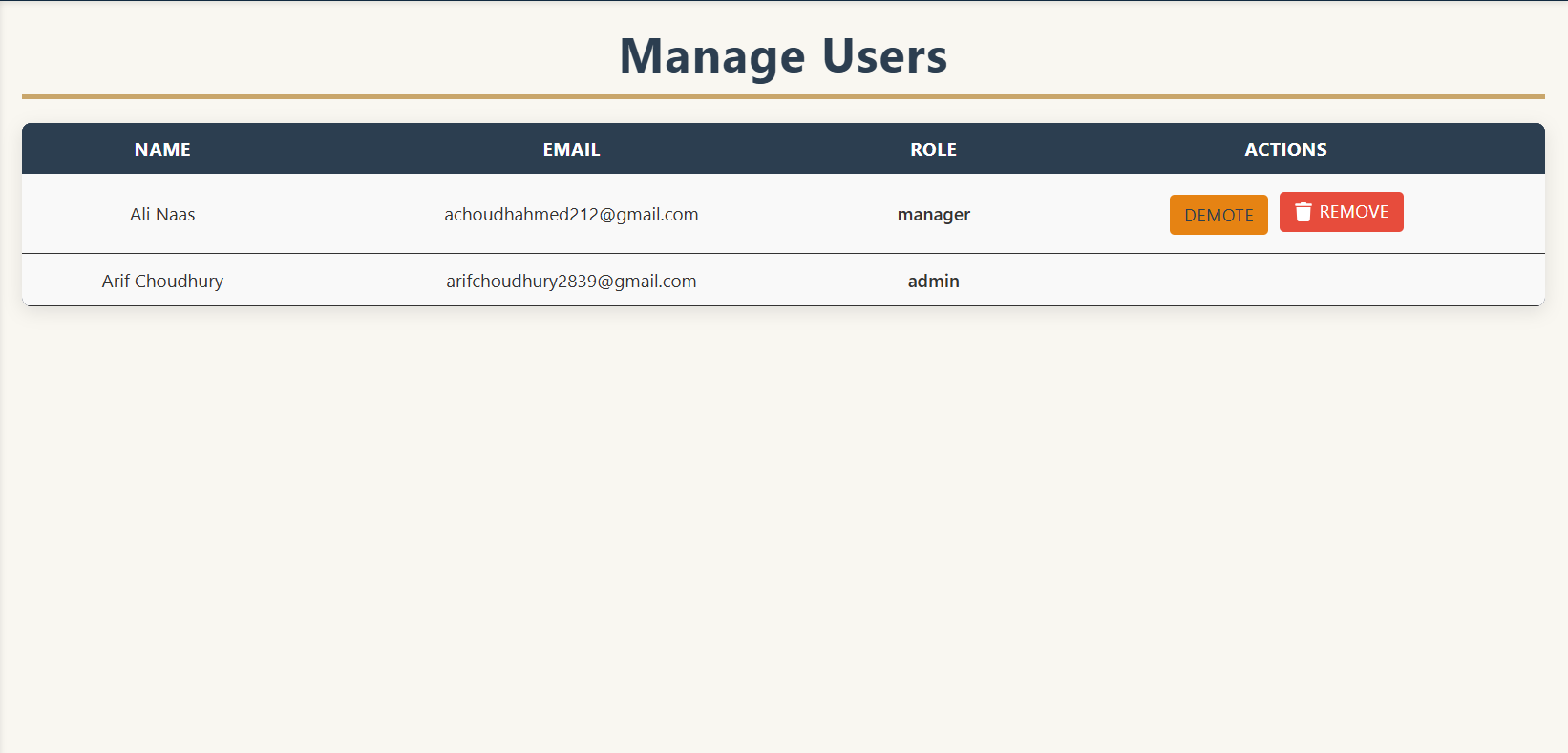
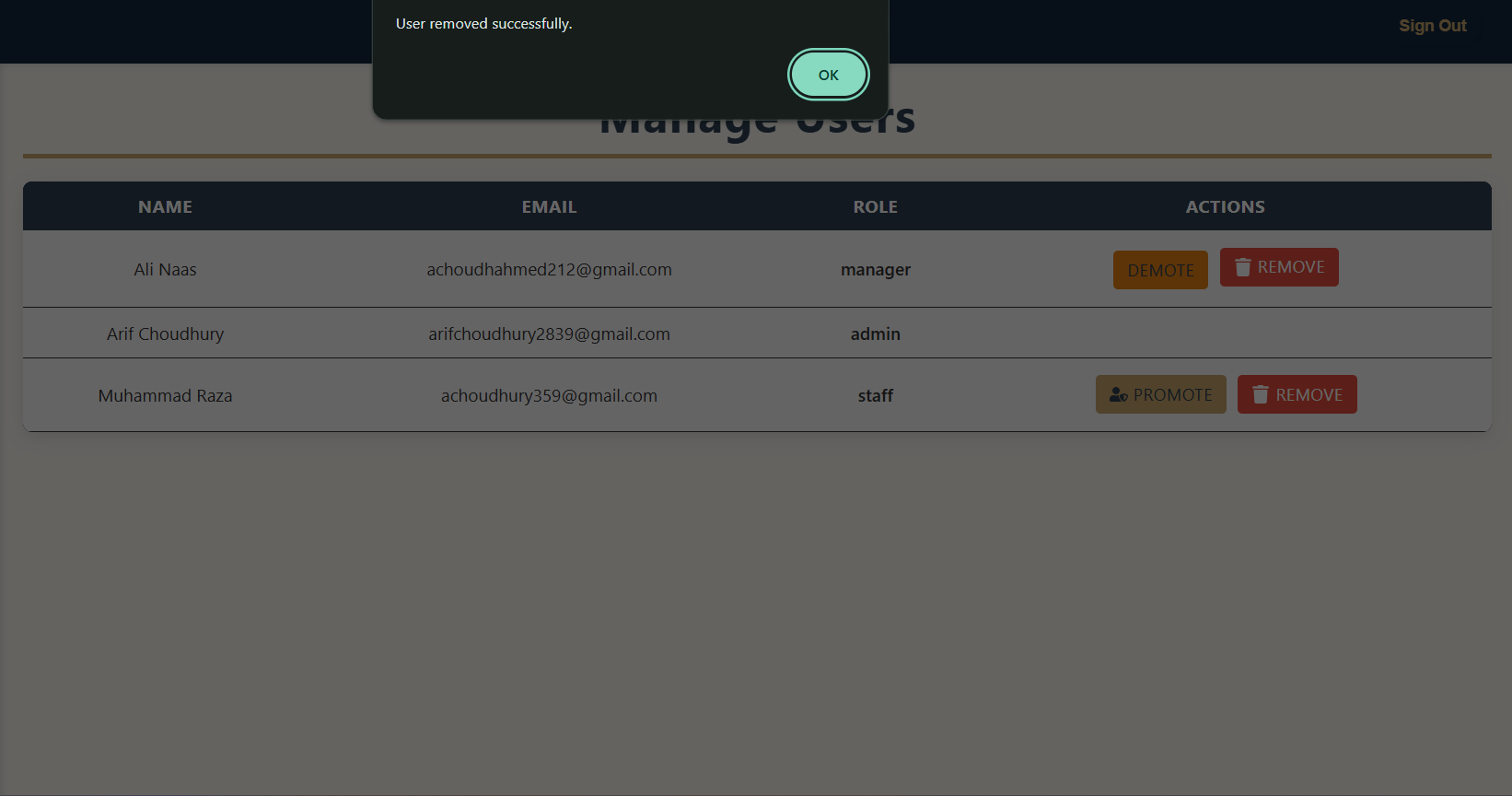
**Manage Users**

The Manage Users section grants complete authority to admins who can manage different roles for their company members as well as remove them from the system. The user interface of **Figure 4.7** displays users while showing role status alongside demotion and promotion and removal buttons. The system requires admins to re-enter their password before executing role changes and to verify the operation for reinforced security measures. **Figure 4.7** display organisational changes through manager demotion which reduces the role to staff.



***Figure 4.7*** ***Admin promoting and demoting users with password confirmation.***

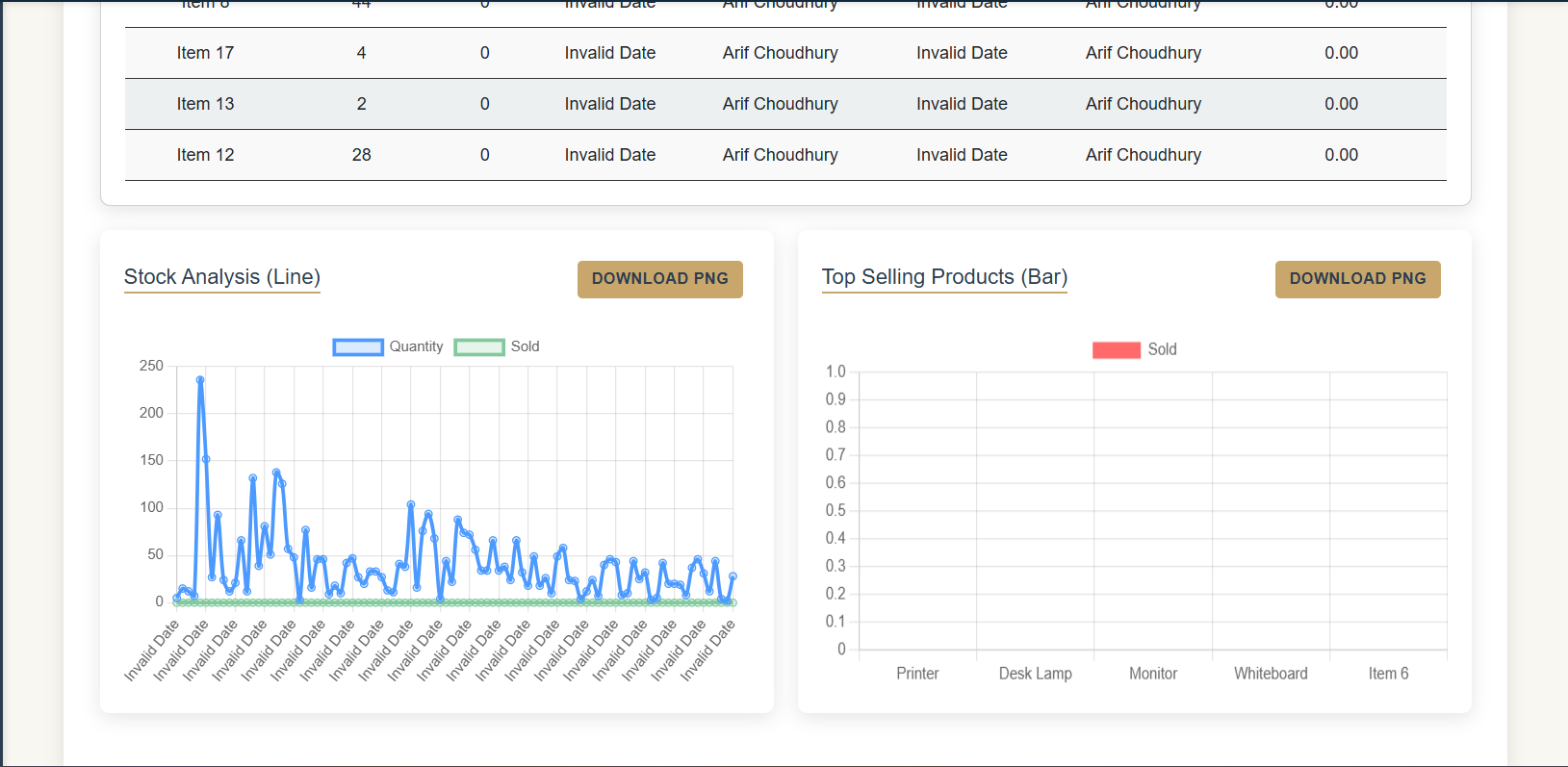
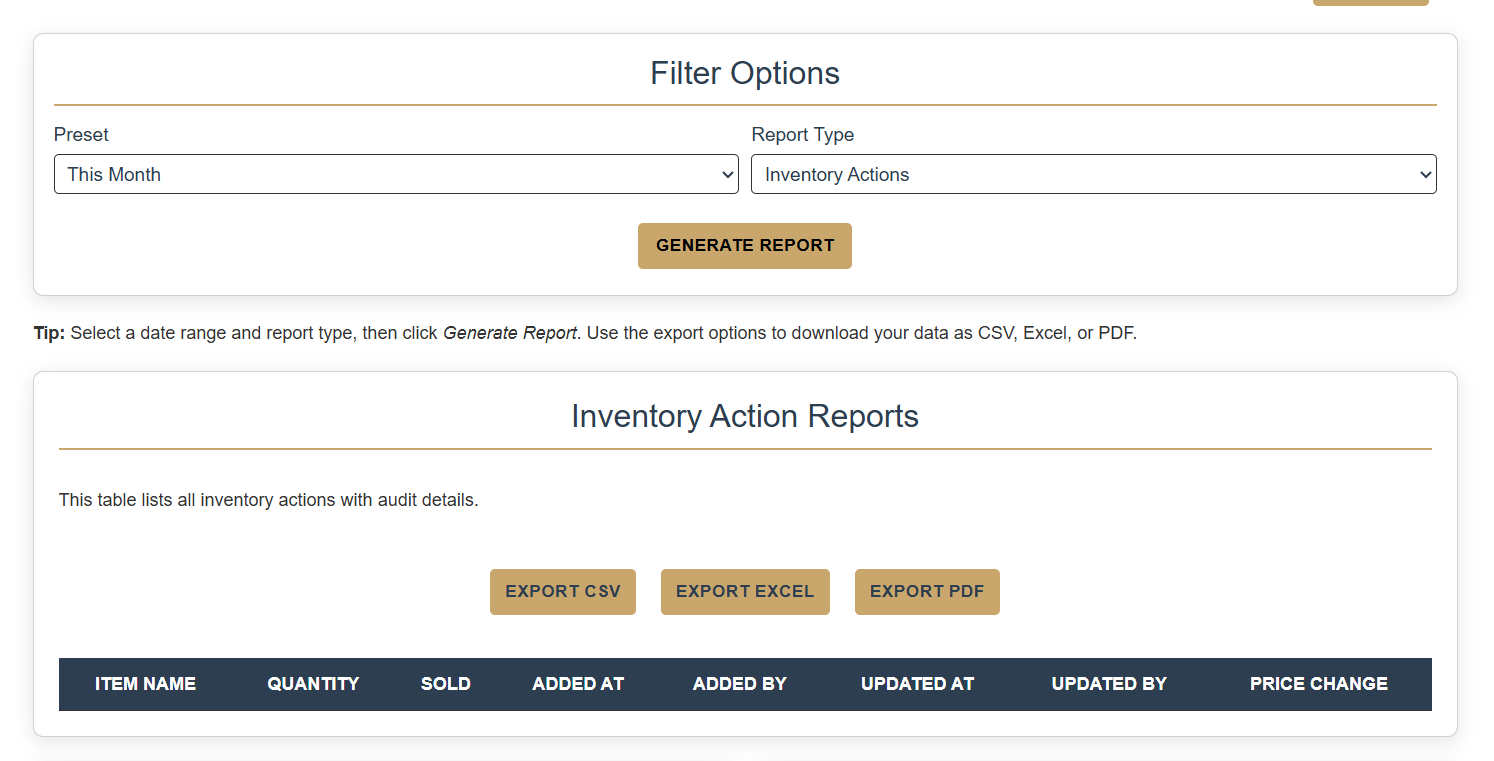
The application shows confirmation and visual feedback to users when they are removed from the company as shows in **Figure 4.8**. All role modifications appear instantly in real-time.



***Figure 4.8* *Removing a user from the company with confirmation.***

**Reports and Analysis**

The system includes a reporting page where Admins and Managers can filter data, view visual summaries, and export results. **Figures 4.9 to 4.11** showcase filtering, table output, CSV export, PDF export.

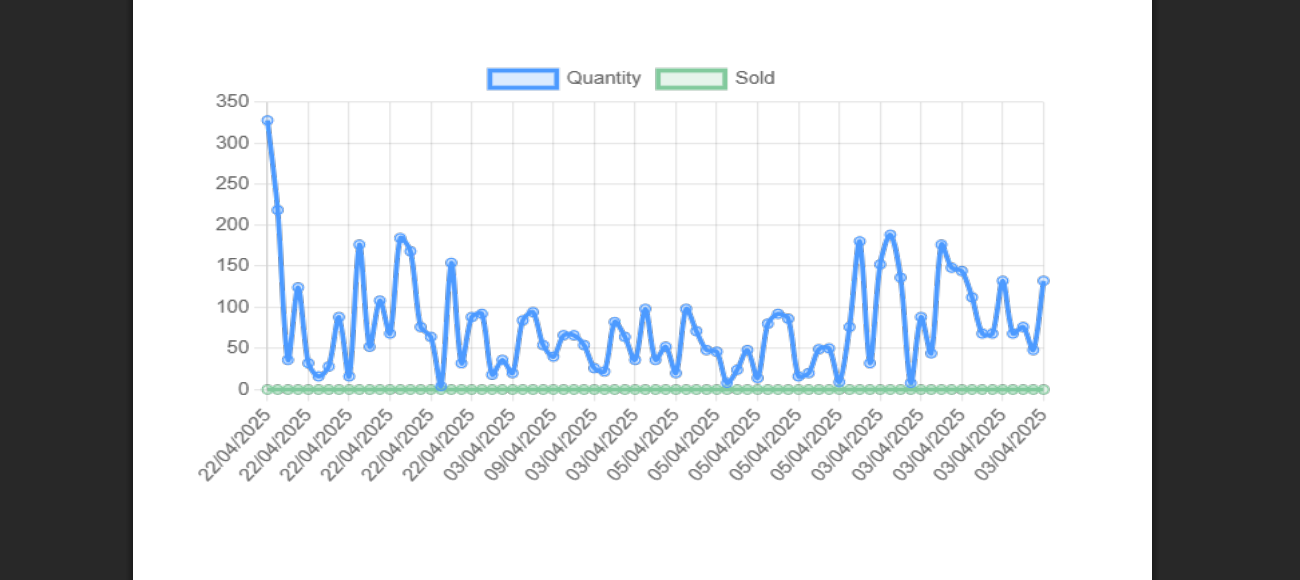
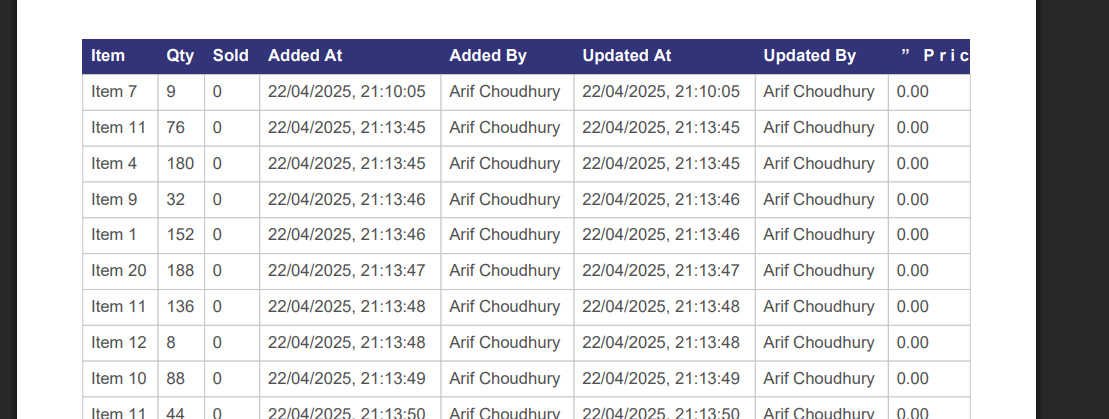


***Figure 4.9: Filtered reports with tabular and graphical views.***

A screenshot of a computer

AI-generated content may be incorrect.

***Figure 4.10: Exported report in CSV format.***



***Figure 4.11: Exported PDF report with tables and visualisations.***

**Notifications Page**

Admins and Managers can assign tasks using a simple task creation form (**Figure 4.12**). These tasks appear on the dashboard for all users, with priority markers. **Figure 4.13** shows the task view accessible by all user roles.

A screenshot of a computer

AI-generated content may be incorrect.

***Figure 4.12: Task creation form with title, description, and priority.***

A screenshot of a computer

AI-generated content may be incorrect.

***Figure 4.13: All-user view of assigned tasks and system notifications.***

#### **4.2 Discussion of System**

The designed system produced an interface that was responsive and easy to use, which provided many valuable opportunities for a user that operates a small business. The interface was able to be used by non-technical users by using a simple interface, simple colours, and an easy-to-follow navigation path. Overall, the usability of the system created features that produced clear error messages, data validation tests with immediate feedback during registration, inventory management, and report generation processes.

The system protected access and had control functions in place using role validation based on JWT standard. Users were given access routes appropriately based on roles assigned to them. The system maintained server-side RBAC functions and client-side functions using React Context and conditional rendering to maximise control. This meant that Admins managed users, Managers managed inventory, and Staff users had read-only permissions without being distracted by unwanted navigation or technical authorisation issues.

The system benefitted from Firebase Firestore listeners that enhanced real-time data synchronisation presence. Any transaction within the tasks or inventory sections populated in real time to users’ dashboards to assist in collaborative workplace management without the need for manual dashboard refreshes. The automated transactional updates to provide more up-to-date data to users, resulted in increasingly accurate working processes.

The system had several reliability mechanisms hard coded within its use. Users are required to verify their email for login access allowance, and all the elements in forms are required to be filled in. The CSV uploads include tests for file structure and field data inclusions to avoid data corruption. The extensive levels of reliability mechanisms upheld the integrity of the system and was also instrumental in creating user trust in the solution.

One downside was the lack of sales reporting functions. This lack of sale reporting capabilities created empty spaces in sales accounting graphs (see **figure 4.14**). The PDF reporting provided useful summation and guidance around inventory usages, but there were empty graphs related to sales (**figure 4.15**). The system has excellent potential for growth in the future, and the overall engagement of point-of-sale operations or e-commerce would engage the application’s use-ability potential.

A screenshot of a computer

AI-generated content may be incorrect.

***Figure 4.14: Empty graph output due to missing sales transactions***

A screenshot of a computer

AI-generated content may be incorrect.

***Figure 4.15: Report suggestions generated automatically in PDF export****.*

#### **4.3 Evaluation against project goals**

The projects aim focused on building a secure scalable user-friendly cloud-based inventory management system made for small business needs. After thorough evaluation, the system fulfils all the essential targets and requirements that were established at the outset.

RBAC proved to be a successful implementation. The system displays administrative permissions separately from manager permissions and staff permissions in both the frontend and backend. System users with approved authorisation are the sole entities who can access the system and execute sensitive operations to maintain secure information access and uninterrupted data integrity.

The user interface consisting of React components developed using Bootstrap and CSS provides organisations with an interactive experience alongside dynamic responsiveness. The system enables essential workplace operations ranging from authentication to inventory maintenance to user administration to report output routines through an interface that does not need technical expertise, therefore, meeting standards for non-technical business administrators.

Firestore listeners enabled real-time synchronisation for the application. The system shows inventory modifications directly to all devices and users without needing manual refresh actions. The system delivers responsive capabilities which provide direct insights to stock levels while showing business operations in real-time.

Chart.js provides the reporting and analytics module with visual dashboards which users can employ to filter data while monitoring inventory trends alongside detecting low stock quantities. Stock movements are effectively tracked through the system, yet sales metrics remain limited because of no built-in sales module.

Businesses reached cloud scalability through the utilisation of Firebase infrastructure. Users and inventories and companies can easily integrate into the hierarchical Firestore structure for data maintenance without merging information or restricting access control.

The system enables users to customise reports before exporting them into CSV or Excel or PDF formats. The system generates meaningful outputs that serve businesses for auditing needs as well as review purposes along with decision support.

The implemented system satisfies each of the project objectives. The application provides instant operational capabilities along with advanced security functions and capacity for expansion and user-friendly design which helps small companies manage their inventory requirements effectively.

#### **4.4 Challenges and limitations**

Various difficulties and constraints appeared throughout the development period of the inventory management system even though essential targets were achieved.

A major issue against the system development was the absence of point-of-sale (POS) or sales integration. The inventory tracking system operated properly yet the advanced analytics capabilities did not work as expected because the system lacked sales transaction information. The integration of visualisations became partial or generic in both Dashboard views and PDF report displays. The system requires either an integrated POS system or external sales functions to enable this analytical functionality to be fully enabled.

Another obstacle during implementation was the registration process for users who initially authenticate through Google. The native features provided by Firebase Authentication do not support collecting extra information such as company name and user roles during OAuth sign-in authentication. Users needed to complete an extended data entry process following Google sign-in because of the modified two-part authentication system.

The flexible design of Firestore helped rapid development and scalability yet introduced challenges to enforce strict structure policies and valid data check-ups at database level. The system needs precise backend programming logic to protect against invalid database inputs particularly during bulk CSV data uploads because improper file organisation can generate data mismatches.

The current system would benefit from additional features like inventory alert emails and the inclusion of product image storage and barcode scanning capabilities to improve functionality in small retail businesses.

The system delivered secure functionalities which matched project aims but the observed limitations can guide future improvements for deployment in real-world settings.

## **Critical Appraisal**

#### **5.1 Critical Analysis of this Project**

The development of this project turned out to be challenging since unanticipated obstacles required me to modify my implementation approach. I discovered Google OAuth implementation required more complicated implementation steps than I initially suspected. Before starting the implementation, I did not recognise that I needed additional information to handle Google users like company name, and their full name to store in firebase database.  
  
The JWT authentication system operated successfully in most cases, although it occasionally redirected users to the login screen during development. This typically happened when modifying code or Firestore rules, which temporarily disrupted the authentication state or delayed retrieval of user roles, resulting in an invalid or missing token. After several trials, I resolved this issue by verifying the role and status retrieval logic and ensuring proper synchronization between Firebase Auth and route guards. For future development, I plan to improve how user roles are stored and loaded, and make sure the JWT token is re-validated whenever the user navigates to a different page. This should prevent unexpected logouts or access issues.  
  
Overall, the system fulfilled all the functions I had included in my design scope, ranging from authentication features to role-based dashboards, inventory creation tools, bulk upload functionality with task management features and reporting features. However, integrating a full sales tracking or POS system was beyond the current project. This gap limited analytics features leaving areas such as top-selling items and revenue graphs empty.  
  
The decision to centre business logic inside Flask instead of allowing direct Firebase access proved to be a correct architectural move. The system brought enhanced backend features together with better audit capabilities and risk reduction practises. With hindsight, I would have enforced a stricter separation of concerns across frontend components earlier. Some components started to grow too large, mixing logic and UI, largely due to my initial inexperience with React and Flask. Adopting a more modular component structure earlier would have simplified testing and improved reusability

#### **5.2 Social, Commercial and Sustainability Context**

The system targets small businesses that lack internal IT teams. The application features an intuitive design and role-specific access rules together with its cloud-based structure meaning users can seamlessly install it without extensive training needs. The SaaS implementation becomes profitable through combination with sales capabilities while expanding its connections to EPOS systems and other third-party tools. Pricing strategies will create specific plans for different levels of inventory quantities or number of user accounts.

This system stands out compared to conventional inventory management technology because it is tailored for small businesses, operates in real-time and connects through Google accounts while letting users construct tailored reports using its user-centred interface. Social sustainability and operational efficiency are promoted through the platform which enables remote work and improved productivity by digitising inventory records.

The system faces two main weaknesses because it depends on Firebase and lacks offline access capabilities. Security functions rely on JWT tokens together with structured security rules in addition to the dynamic scalability of Firestore to prevent potential threats. The system has growth potential to become either an available business software alternative or a commercial product.

#### **5.3 Personal Development**

Working on this project really enhanced my development and programming abilities with React and Flask, while also improving my skills with Firebase Firestore, Firebase Authentication, JWT handling and error-oriented design. I learned how to create successful authentication flows, manage asynchronous processes, improve handling of client/client-server communication, and most importantly, design in a more user-centred way.

Over many months as I worked through my various development tasks, I also improved my own self-discipline and perseverance with the uncertainty of unexpected technical distractions. This experience has given me a significant boost in my developer confidence, and has developed my planning, problem solving and technical writing skills, that would be applicable to any future full-stack or product development role.

The project also allowed me to appreciate usability and accessibility even more. While designing interfaces for users who are not technical, it shaped several of my key decisions and pushed me to consider user experience at all levels of the system.

## **Conclusion**

The developed system created a secure time-sensitive inventory management solution that fulfils operational requirements of small businesses. The system employs React, Flask and Firebase integration to deliver a responsive cloud platform while helping users manage their inventory through CRUD functions and administer users at different permission levels and create detailed reports.

Users receive helpful feedback through responsive interfaces while benefiting from the system's solution of business issues related to lack of automation and insufficient technical skills. The module-based structure of this system distinctively partitions UI interface operations from API functionalities and database information updates, which boosted performance and made maintenance easier. Functional testing confirmed the stability of authentication flows, CSV validation, and role-controlled access, validating the system’s effectiveness in day-to-day business contexts.

However, the system does face limitations that include no ability to monitor sales activity nor support offline operations. The system remains incomplete because it lacks certain elements, but these areas can develop into commercialisation possibilities which will integrate into existing POS systems as a SaaS solution. The project reached its targets by providing a flexible solution with operational usability and sound technical reliability for delivering a concrete solution to small business settings.

## **References**

[1] A. Kittisak, "Challenges and strategies for inventory management in small and medium-sized cosmetic enterprises: A review," International Journal of Information Technology and Computer Science Applications (IJITCSA), vol. 1, no. 2, pp. 71–77, 2023.

[2] T. Z. T. Mat, M. Hashim, S. Saad, and M. B. Ismail, "Inventory management practices among small and micro businesses during COVID-19 pandemic," *Accounting and Finance Research*, vol. 12, no. 4, pp. 1–86, 2023.

[3] F. . Ugbebor, M. Adeteye, and J. Ugbebor, “Automated Inventory Management Systems with IoT Integration to Optimize Stock Levels and Reduce Carrying Costs for SMEs: A Comprehensive Review”, *JAIGS*, vol. 6, no. 1, pp. 306–340, Nov. 2024.

[4] A. I. Otuonye, "Cloud-based enterprise resource planning for sustainable growth of SMEs in third world countries," *International Journal of Computer Science and Information Security (IJCSIS)*, vol. 19, no. 5, pp. 68–84, 2021.

[5] P. Scholtz, "Does usability matter? An analysis of the impact of usability on technology acceptance in small and medium enterprises (SMEs)," *International Journal of Innovation, Creativity and Change*, vol. 11, no. 11, pp. 309–330, 2020.

[6] V. Bindiganavale and J. Ouyang, "Role-Based Access Control in Enterprise Application–Security Administration and User Management," CiteseerX, [Online]. Available: <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=4620e4355bdd71f280ba724a12b5234c305ff13f>. [Accessed: 28-Apr-2025].

[7] A. Mansour, "Building Scalable Web Applications: Researching Frameworks and Design Patterns," Bachelor's thesis, Lahti University of Applied Sciences, 2016.

[8] apidog.com, "Axios vs Fetch: Understanding the Differences," 2024. [Online]. Available: <https://apidog.com/blog/axios-vs-fetch/>. [Accessed: 28-Apr-2025].

9] D. Ghimire, "Comparative study on Python web frameworks: Flask and Django," Bachelor's thesis, Lapland University of Applied Sciences, 2020.

10] Y. Balaj, V. D. Dang, and M. H. Abdmeziem, "Token-Based vs Session-Based Authentication: A Survey," 2017. [Online]. Available: <https://www.researchgate.net/publication/320068250_Token-Based_vs_Session-Based_Authentication_A_survey>. [Accessed: 28-Apr-2025].

[11] R. Kesavan, D. Gay, D. Thevessen, J. Shah, and C. Mohan, "Firestore: The NoSQL Serverless Database for the Application Developer," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pp. 3376–3388, Apr. 2023.

[12] R. Wieruch, *The Road to React with Firebase*, Leanpub, 2019.