# Table of Figures

[Figure 1 : Password Security and Validation in models.py of app.py 5](#_Toc182838515)

[Figure 2 : SQL Injection Prevention in models.py and product.py of app.py 6](#_Toc182838516)

[Figure 3: CSRF Protection in app.py and axiosConfig.js 7](#_Toc182838517)

[Figure 4: File Upload Prevention in app.py and axiosConfig.js 8](#_Toc182838518)

[Figure 5 Token-Based Authentication in rbac\_app.py, LoginForm.js, App.js and axiosConfig.js 10](#_Toc182838519)

[Figure 6: Account Lockout Mechanism in models.py in RBACservices 11](#_Toc182838520)

[Figure 7: 2FA in models.py and rbac\_app.py in RBACservices 12](#_Toc182838521)

[Figure 8: HTTPS Enforcement in app.py 14](#_Toc182838522)

[Figure 9: Input Validation in models.py of app.py 15](#_Toc182838523)

[Figure 10: RBAC in app.py (require rbac\_app.py functionality) 16](#_Toc182838524)

[Figure 11: Secure Error Handling in axiosConfig.js 17](#_Toc182838525)

[Figure 12: Password Reset in rbac\_app.py 17](#_Toc182838526)

[Figure 13: CORS in app.py 18](#_Toc182838527)

[Figure 14: User Enumeration Prevention in rbac\_app.py 18](#_Toc182838528)

[Figure 15: Activity Logging Prevention in rbac\_app and app.py.py 19](#_Toc182838529)

[Figure 16: Login Page 21](#_Toc182838530)

[Figure 17: Admin Dashboard 22](#_Toc182838531)

[Figure 18: Warehouse Management Page 23](#_Toc182838532)

[Figure 19: Warehouse Creation Modal 24](#_Toc182838533)

[Figure 20: Warehouse Update Modal 24](#_Toc182838534)

[Figure 21: Inventory Management Page 26](#_Toc182838535)

[Figure 22: Inventory Update Page 27](#_Toc182838536)

[Figure 23: Order Management Page 28](#_Toc182838537)

[Figure 24: Invoice Generation 29](#_Toc182838538)

[Figure 25: Add Item to Order Modal 30](#_Toc182838539)

[Figure 26: Returns Management Page 31](#_Toc182838540)

[Figure 27: Product Management Page 32](#_Toc182838541)

[Figure 28: Add New Product Modal 33](#_Toc182838542)

# **Setup Requirements**

**Step 1: Add the ca.crt File to Trusted Root Certificate Authorities**

To ensure HTTPS works properly, you need to trust the ca.crt file located in RBACservice/certs. Follow these platform-specific steps:

**Windows:**

1. Open the **Run** dialog (Windows + R) and type mmc, then press **Enter**.
2. In the MMC console, go to **File > Add/Remove Snap-in**.
3. Select **Certificates** and click **Add**, then choose **Computer account** and click **Next**.
4. Select **Local Computer** and click **Finish**.
5. Expand **Trusted Root Certification Authorities** and right-click **Certificates**. Choose **All Tasks > Import**.
6. Browse to RBACservice/certs/ca.crt and follow the prompts to import the certificate.
7. Restart your application or service if needed.

**Linux:**

1. Copy ca.crt to the /usr/local/share/ca-certificates/ directory.
2. Update the certificate store.
3. Restart your application or service.

**MacOS:**

1. Open **Keychain Access** from **Applications > Utilities**.
2. Drag and drop RBACservice/certs/ca.crt into the **System** or **Login** keychain.
3. Right-click the certificate, select **Get Info**, and under **Trust**, set **When using this certificate** to **Always Trust**.
4. Close the dialog and enter your password if prompted.
5. Restart your application or service.

**Step 2: Install Dependencies**

To ensure the backend and frontend run smoothly, install the required dependencies:

**For the Backend (Main Service):**

1. Open a terminal and navigate to the backend directory.
2. Install Python dependencies: **pip install -r requirements.txt**

**For the RBAC Service:**

1. Navigate to the RBACservice directory.
2. Install Python dependencies: **pip install -r requirements.txt**

**For the Frontend:**

1. Navigate to the React application directory.
2. Install Node.js dependencies: **npm install**

**Step 3: Update Environment Variables and db\_config**

1. Open the .env file and db\_config (easy to change) in the backend and RBAC service directories.
2. Replace MAIL\_USERNAME and MAIL\_PASSWORD with your own credentials:
   * Create a new email account or use an existing one.
   * Generate an app-specific password (if applicable, e.g., for Gmail).
   * Update the .env file:
     1. MAIL\_USERNAME= ’your-email@example.com’
     2. MAIL\_PASSWORD= ’your-app-password’
3. Save the changes.

**Step 4: Modify create\_user Emails for Testing**

1. Open rbac\_app.py in the RBAC service directory.
2. Locate the create\_user function.
3. Replace the dummy example emails with yours to test the password reset functionality:

create\_user('admin', 'youremail@gmail.com', 'Admin123!', roles['Admin'], two\_factor\_enabled=True)

1. Drop the tables each time you modify this by running the database reset commands.

**Step 5: Run the Application**

1. Start the backend (main service): **python app.py**
2. Start the RBAC service: **python rbac\_app.py**
3. Start the frontend: **npm start**

Your application should now be running correctly.

# **Security Features**

This section is a brief look at the security features integrated into our application, focusing on how these measures are implemented across various components. While we will provide specific examples and indicate where these features are demonstrated, it is important to note that they are applied consistently throughout the application. This includes the main Flask backend (app.py, models.py, and APIs), the RBAC service (rbac\_app.py, models.py), and the React frontend (App.js, axiosConfig.js, LoginForm.js, etc.).

The communication flow between components is structured as follows:

* The React frontend interacts with the main Flask backend for all API calls.
* The main Flask backend communicates with the RBAC service for authentication-related operations.

This architecture positions the main Flask backend as a proxy for the RBAC service, adding an additional layer of security by isolating the RBAC service from external connections. Only the main Flask backend can directly access the RBAC service, ensuring a controlled and secure interaction.

A video demonstration showcasing part of the app’s security functionality and briefly explaining the code will be included in the submission zip file.

## **Password Security and Validation**

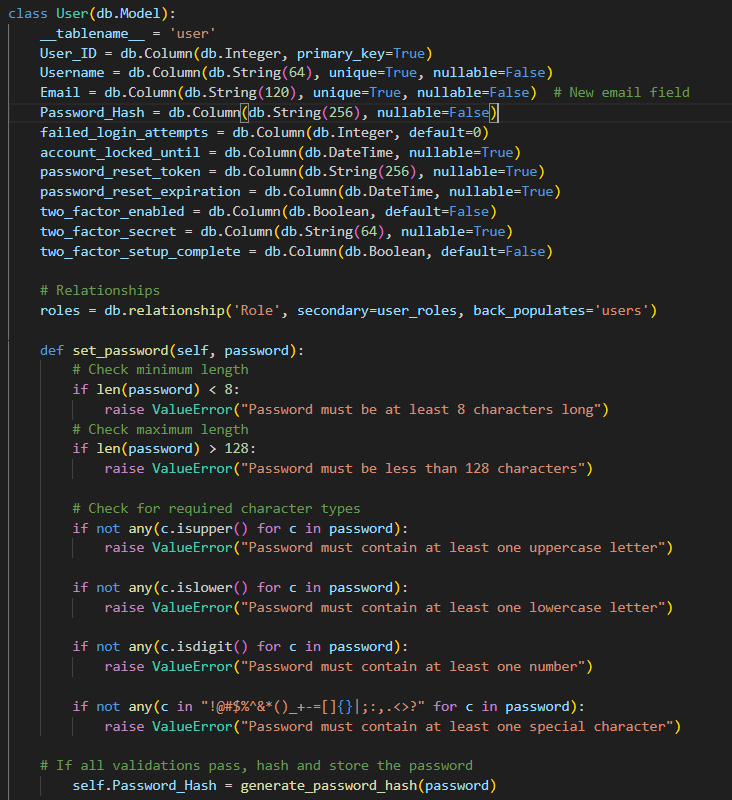


Figure 1 : Password Security and Validation in models.py of app.py

### **Password Hashing:**

* Passwords are hashed using **generate\_password\_hash**, which uses scrypt for hashing, before storing them in the database. ([OWASP Password Storage Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html))
  + This prevents storing plaintext passwords and protects user credentials in case of a database breach.

### **Password Validation:**

* The code enforces strong password policies by checking for minimum and maximum lengths and the inclusion of uppercase letters, lowercase letters, numbers, and special characters. ([Microsoft](https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-10/security/threat-protection/security-policy-settings/password-must-meet-complexity-requirements))
  + This mitigates cryptographic failures by ensuring passwords are strong against brute-force attacks.

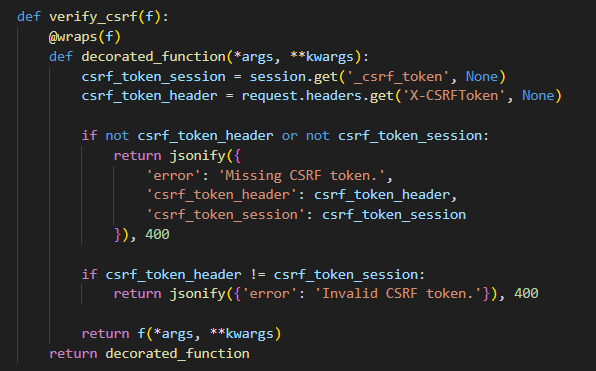
## **SQL Injection Prevention:**

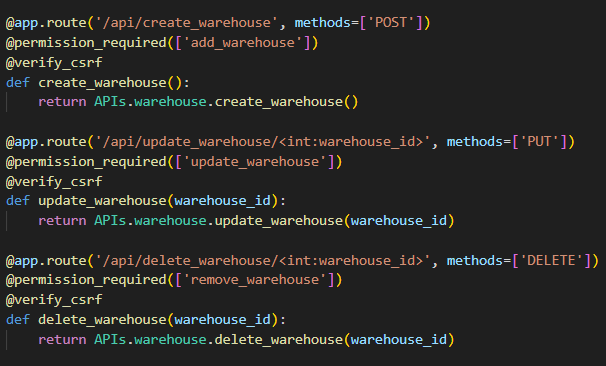
## **A computer screen shot of a program code Description automatically generated**

Figure 2 : SQL Injection Prevention in models.py and product.py of app.py

* *The usage of* SQLAlchemy ORM abstracts database queries and automatically uses parameterized queries
* *This* helps prevent SQL injection attacks by not directly including user input in SQL statements.

## **Cross-Site Request Forgery (CSRF) Protection**





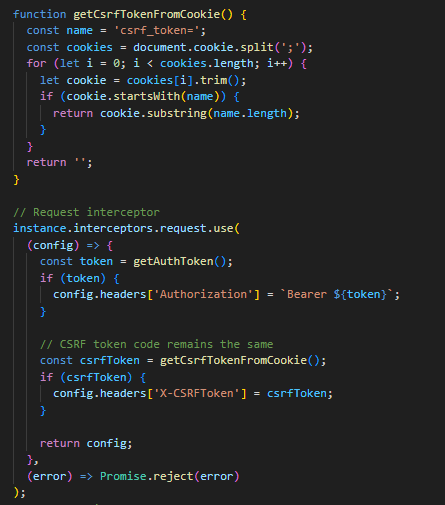


Figure 3: CSRF Protection in app.py and axiosConfig.js

* The **verify\_csrf** decorator is used to protect against CSRF attacks by requiring a token for form submissions (all APIs with POST, PUT or DELETE requests have this decorator for CSRF Protection).
* The frontend extracts the CSRF token from cookies and includes it in the X-CSRFToken header of HTTP requests. This ensures that state-changing requests are protected against CSRF attacks.

## **File Upload Injection Prevention**

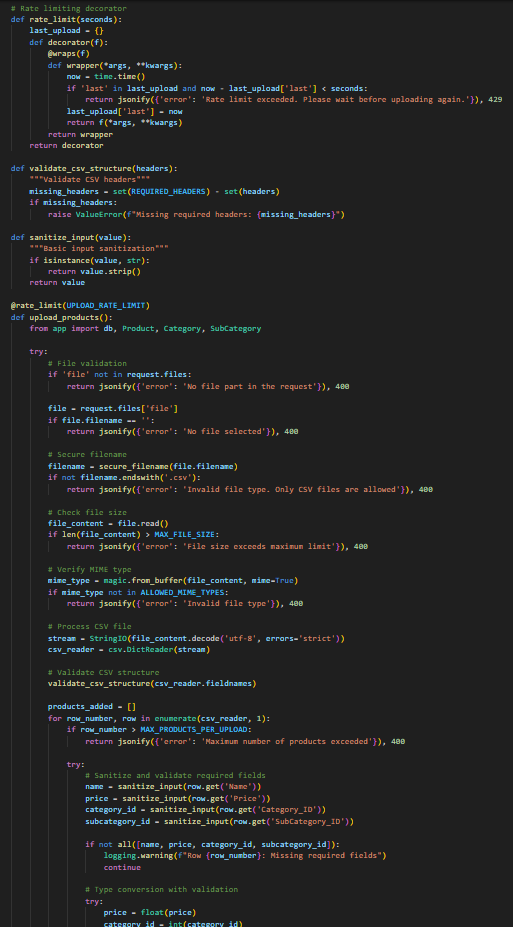


Figure 4: File Upload Prevention in app.py and axiosConfig.js

* The code has comprehensive file validation using magic numbers for MIME type checking, implemented rate limiting to prevent DoS attacks that will affect the dB, added input sanitization for all fields, improved error handling and logging, added transaction management, implemented CSV structure validation, added file size limits, used secure filename handling, added maximum products per upload limit, improved type checking and validation, and added proper session management. SQLAlchemy already handles SQL injection protection through parameterized queries.

## **Authentication and Session Management**

### **Token-Based Authentication:**



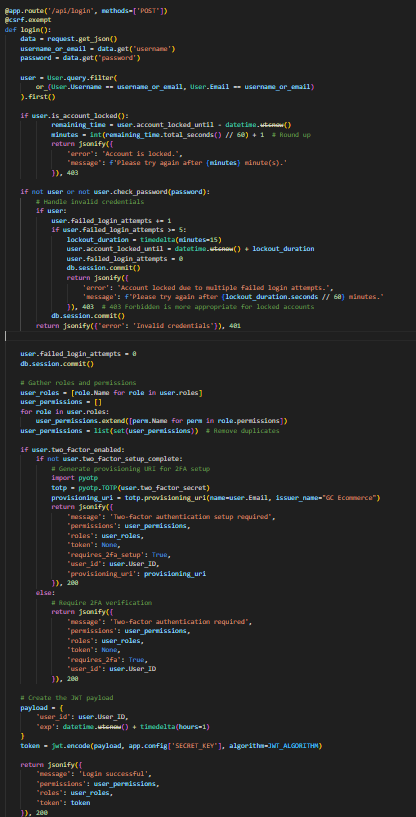


Figure 5 Token-Based Authentication in rbac\_app.py, LoginForm.js, App.js and axiosConfig.js

* RBAC generates JWT tokens for authentication based on the user id and sends it to the the main app.py to forward it to react in the Authorization header
* Frontend implements token-based authentication, storing the token securely in **localStorage** and including it in the Authorization header for authenticated requests.
* The SECRET\_KEY is used by Flask to sign session cookies, ensuring that the data stored in the session is secure and has not been tampered with.
* Session Cookie Settings:
* HTTPOnly Flag: This flag is set on cookies to prevent client-side scripts (like JavaScript) from accessing them. This mitigates the risk of cross-site scripting (XSS) attacks stealing session cookies.
* Secure Flag: Ensures that cookies are only sent over HTTPS connections, preventing them from being intercepted over insecure channels.
* Protects against session hijacking by ensuring session cookies cannot be easily accessed or stolen.
* Maintains the integrity and confidentiality of session data.
* Ensures that sessions are properly invalidated upon logout, preventing unauthorized reuse.

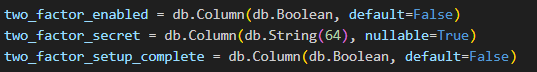
## **Account Lockout Mechanism**

## 

Figure 6: Account Lockout Mechanism in models.py in RBACservices

* Tracks failed login attempts and locks the account after too many failures.
  + Prevents brute-force attacks by limiting login attempts.

**Two-Factor Authentication (2FA)**



A screen shot of a computer program

Description automatically generated

Figure 7: 2FA in models.py and rbac\_app.py in RBACservices

**Location in Code:**

* Backend:
  + models.py (User model updates)
  + rbac\_app.py (Endpoints for 2FA)
* Frontend:
  + LoginForm.js (Handling 2FA in login flow)

**Details:**

Two-Factor Authentication (2FA) adds an extra layer of security by requiring users to provide two forms of identification before accessing their account. In your codebase, 2FA is implemented both on the backend and frontend.

**Backend Implementation (models.py and rbac\_app.py):**

* **User Model Enhancements (models.py):**

The User model is extended with additional fields to support 2FA:

* + two\_factor\_enabled: A boolean flag indicating whether the user has 2FA enabled.
  + two\_factor\_secret: Stores the secret key used to generate 2FA verification codes.
  + two\_factor\_setup\_complete: Indicates whether the 2FA setup process is complete for the user.
* **2FA Endpoints (rbac\_app.py):**
  + /api/setup-2fa: This endpoint initiates the 2FA setup process. It generates a unique secret key for the user and provides a provisioning URI (often in the form of a QR code) that the user can scan using an authenticator app like Google Authenticator or Authy.
  + /api/verify-2fa: This endpoint verifies the 2FA code entered by the user. It checks the code against the secret key stored in the user's record to ensure it matches the expected value generated at that specific time.

**Frontend Implementation (LoginForm.js):**

* The LoginForm component handles the authentication flow. After the user submits their username and password, the backend responds indicating whether 2FA is required.
* If 2FA setup is needed, the component renders the TwoFactorSetup component, guiding the user through scanning the QR code and entering the verification code.
* If 2FA is enabled and required, the component renders the TwoFactorAuth component, prompting the user to enter the current code from their authenticator app.

**Security Benefits:**

Implementing 2FA significantly enhances account security by requiring two forms of authentication:

1. **Something You Know**: The user's password.
2. **Something You Have**: The user's mobile device with an authenticator app.

Even if an attacker compromises the user's password, they cannot access the account without the second factor. This reduces the risk of unauthorized access due to password theft, phishing attacks, or credential stuffing.

## **HTTPS Enforcement**

A screen shot of a computer code

Description automatically generated



A black background with white text

Description automatically generated

Figure 8: HTTPS Enforcement in app.py

* **Flask-Talisman Setup:** Flask-Talisman ensures that all incoming requests are redirected to HTTPS and helps set various HTTP headers that strengthen security, such as HSTS (HTTP Strict Transport Security).
* **SSL Certificate Configuration:** The application specifies the paths to the SSL certificate and key files, enabling it to serve over HTTPS.
* **Security Benefits**:
  + Data Encryption: Encrypts the data transmitted between clients and the server, protecting sensitive information from eavesdropping.
  + Data Integrity: Prevents data manipulation during transmission.
  + Authentication: Confirms the server's identity to the client, reducing the risk of man-in-the-middle attacks.

## **Input Validation and Sanitization**

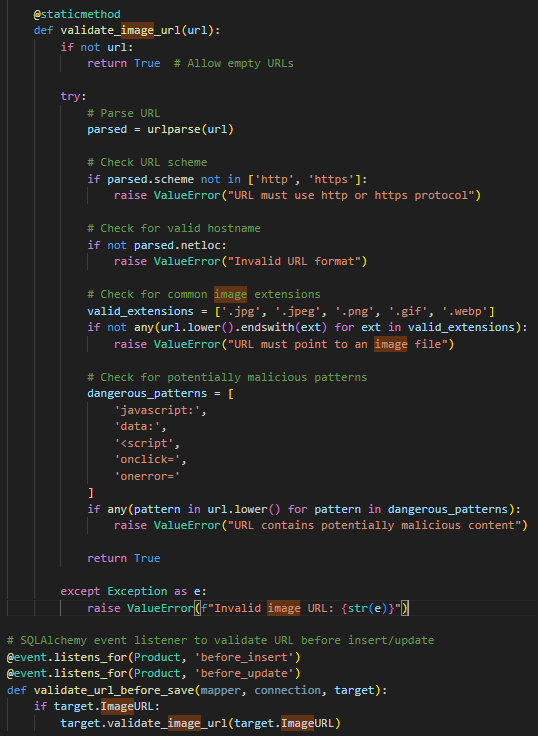


Figure 9: Input Validation in models.py of app.py

* **Validation Method:** The validate\_image\_url static method checks if the provided ImageURL is valid and correctly formatted.
* Security Benefits:
  + Prevents Malicious Input: Ensures that only properly formatted and safe URLs are stored, preventing the inclusion of scripts or malicious content.
  + Protects Database Integrity: By validating inputs before database operations, it maintains the integrity of the data and prevents storage of harmful data.

## **Role-Based Access Control (RBAC) System**





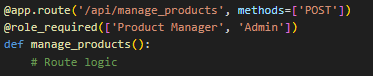


Figure 10: RBAC in app.py (require rbac\_app.py functionality)

* **Models (models.py):**
  + **User Model:** Contains user information and relationships to roles.
  + **Role Model:** Defines roles (e.g., Admin, User) and their associated permissions.
  + **Permission Model:** Specifies granular permissions (e.g., view\_product, edit\_product).
* **Role and Permission Assignment (rbac\_app.py):**
  + Functions like create\_roles\_and\_permissions() initialize roles and permissions in the system and assign them to users.
* **Access Control Decorators (app.py):**
  + **Role-Based Access**
  + **Permission-Based Access**
* **Security Benefits:**
  + Principle of Least Privilege: Users are granted only the permissions they need to perform their tasks, reducing the risk of abuse or accidental misuse.
  + Auditing and Accountability: Actions can be tracked based on roles and permissions, aiding in monitoring and forensic analysis.
  + Scalability: Roles and permissions can be managed centrally, making it easier to update access controls as the organization grows.

## **Secure Error Handling in Axios Interceptors**

## 

Figure 11: Secure Error Handling in axiosConfig.js

* **Logout Functionality:** The logout() function removes authentication tokens and redirects the user to the login page when an unauthorized or forbidden response is received.
* **Security Benefits:**
  + Prevents Unauthorized Access: Ensures that if the user's session becomes unauthorized (e.g., token expires), they are logged out and cannot continue accessing protected resources.
  + Protects Sensitive Information: Avoids displaying detailed error messages that could reveal internal workings of the application to potential attackers.
  + User Feedback: Provides a clear indication to the user that they need to re-authenticate.

## **Email Verification and Password Reset Tokens**



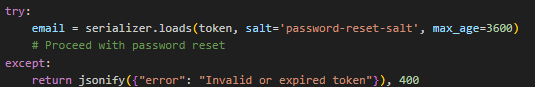


Figure 12: Password Reset in rbac\_app.py

**Password Reset Workflow:**

* **Request Password Reset (/api/request-password-reset):** The user submits their email, and if it exists, the application sends an email with a reset link containing the secure token.
* **Reset Password (/api/reset-password/<token>):** The user accesses the reset link, and the token is validated:

**Security Benefits:**

* **Time-Limited Tokens:** Tokens expire after a set time (e.g., 1 hour), reducing the window for potential misuse.
* **Token Validation:** Ensures that only valid tokens are accepted, and associated with the correct user.
* **Secure Communication:** Tokens are sent via email to the user's registered email address, which should be secured.

## **Cross-Origin Resource Sharing (CORS) Configuration**



Figure 13: CORS in app.py

* **Allowed Origins:** Only requests originating from https://localhost:3000 are allowed, which should be the domain where your frontend application is hosted during development.
* **Security Benefits:**
  + Prevents Unauthorized Access: Restricts which domains can make requests to your API, reducing potential abuse.
  + Supports Credentials: Ensures that authentication tokens and cookies are correctly handled in cross-origin requests.

## **Username Enumeration Prevention**

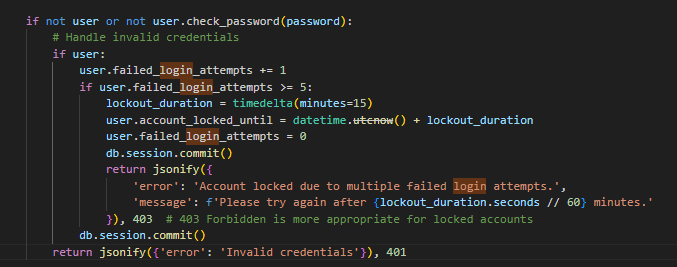


Figure 14: User Enumeration Prevention in rbac\_app.py

* The code prevents user enumeration by providing the same error response for both incorrect usernames and incorrect passwords. When authentication fails, it returns a generic "Invalid credentials" 401 message without indicating whether the user exists. This consistent response ensures that attackers cannot determine valid usernames based on the system's feedback, effectively mitigating user enumeration attacks.
* Security Benefits:
  + Prevents User Enumeration: Returns a generic "Invalid credentials" message regardless of whether the username exists, preventing attackers from discovering valid usernames.
  + Brute Force Protection: Implements an account lockout after 5 failed login attempts, mitigating the risk of brute-force password attacks.

## **Activity Logging**



A screen shot of a computer program

Description automatically generated

Figure 15: Activity Logging Prevention in rbac\_app and app.py.py

* The code logs user activity by capturing details of each API request made to the application. When a user interacts with the API, the app.py file sends information such as the user ID, endpoint accessed, HTTP method, and a timestamp to the rbac\_app.py service. The rbac\_app.py then records this information in the ActivityLog database table. This process allows the system to maintain a detailed record of user actions.
* **Security benefits:**
* **Accountability:** Keeps a record of user actions, helping to hold individuals responsible for their activities.
* **Audit Trail:** Provides a chronological log of events, essential for auditing and compliance purposes.
* **Intrusion Detection:** Helps identify unauthorized or malicious activities by monitoring unusual patterns.
* **Incident Response:** Facilitates quicker investigation and resolution of security incidents through detailed logs.
* **Regulatory Compliance:** Assists in meeting legal and industry standards for security and data protection.

## **Conclusion**

Though we’ve tried to include all we can in this section, We still highly recommend watching the video which goes more in-depth with the features and provides a live demonstration of how things connect as well as analyzing the code.

# Web Application Section

## Login Page

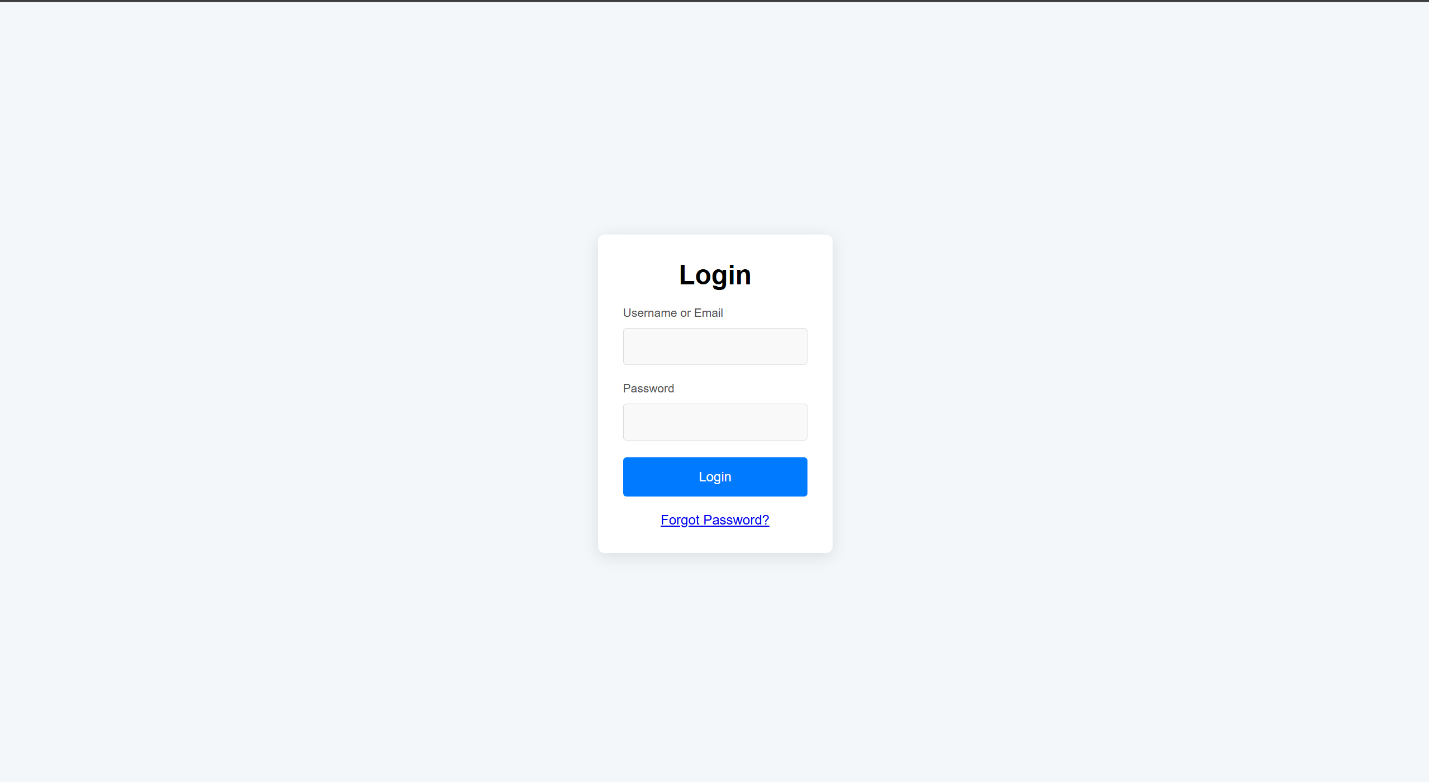


Figure 16: Login Page

The login page is a critical component of the application, serving as the primary entry point for users. The page features two input fields: one for the username or email and another for the password, both of which are required for authentication. The password field ensures confidentiality by masking the entered characters. Additionally, the page includes a "Forgot Password?" link, enabling users to recover their accounts if they forget their login credentials. The page is designed to provide a seamless user experience with input validation to prevent the submission of incomplete or invalid data. Error messages are displayed if the login attempt fails, ensuring that users receive clear feedback. The minimalistic design and focus on essential functionality make the login page intuitive and user-friendly while maintaining robust security practices.

## Dashboard

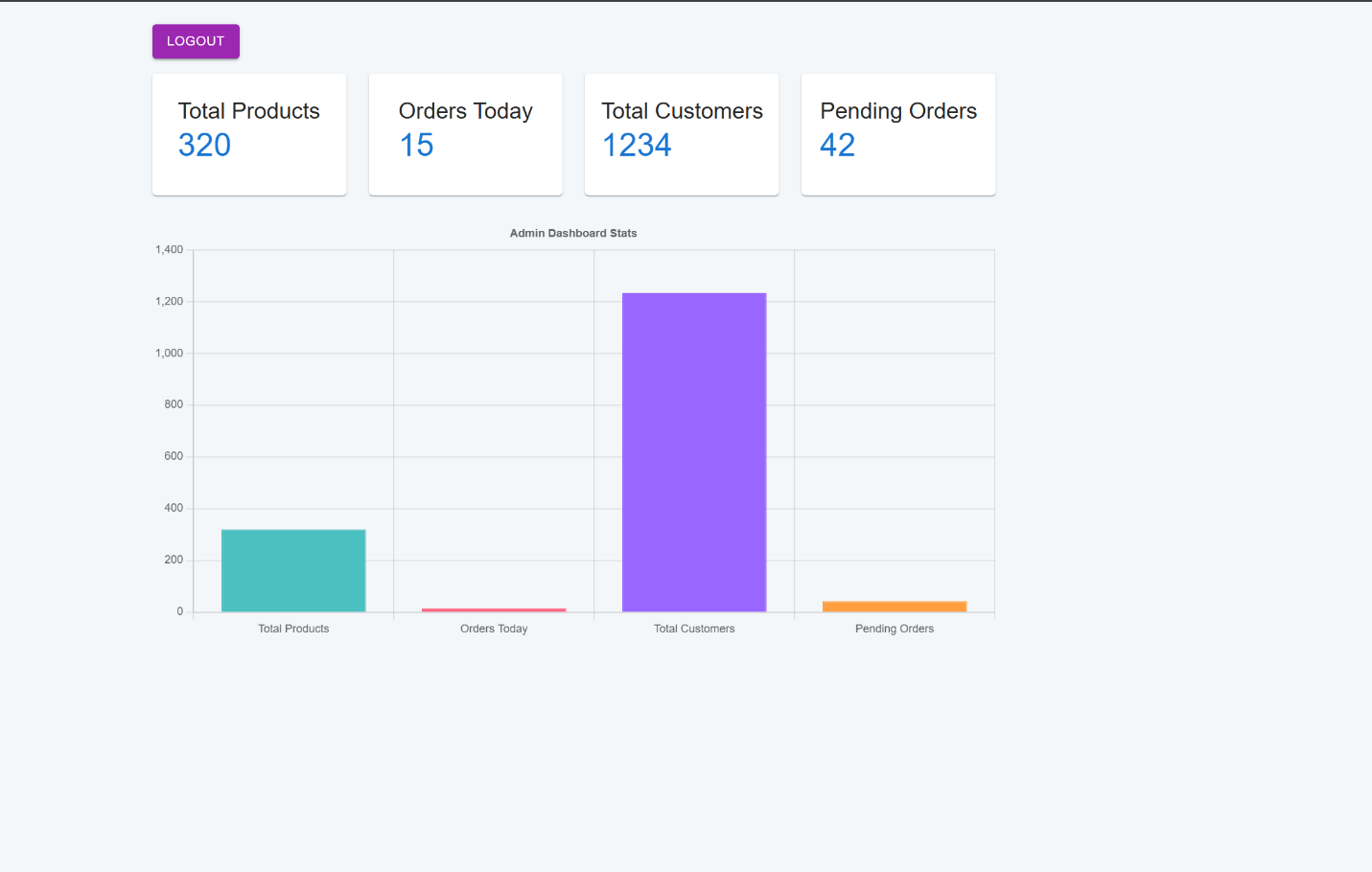


Figure 17: Admin Dashboard

The dashboard page serves as a central hub for administrators, providing an overview of the system's key metrics. At the top of the page, four cards display essential statistics: "Total Products," which shows the total count of products available in the system; "Orders Today," which highlights the number of orders placed within the current day; "Total Customers," which represents the number of registered users classified as customers; and "Pending Orders," which displays the count of orders awaiting processing or fulfillment. Below these metrics, a bar chart visually represents the same data, offering a clear and concise graphical summary that allows for easy interpretation of trends. The page also includes a "Logout" button, enabling administrators to securely terminate their session.

## Warehouse Management

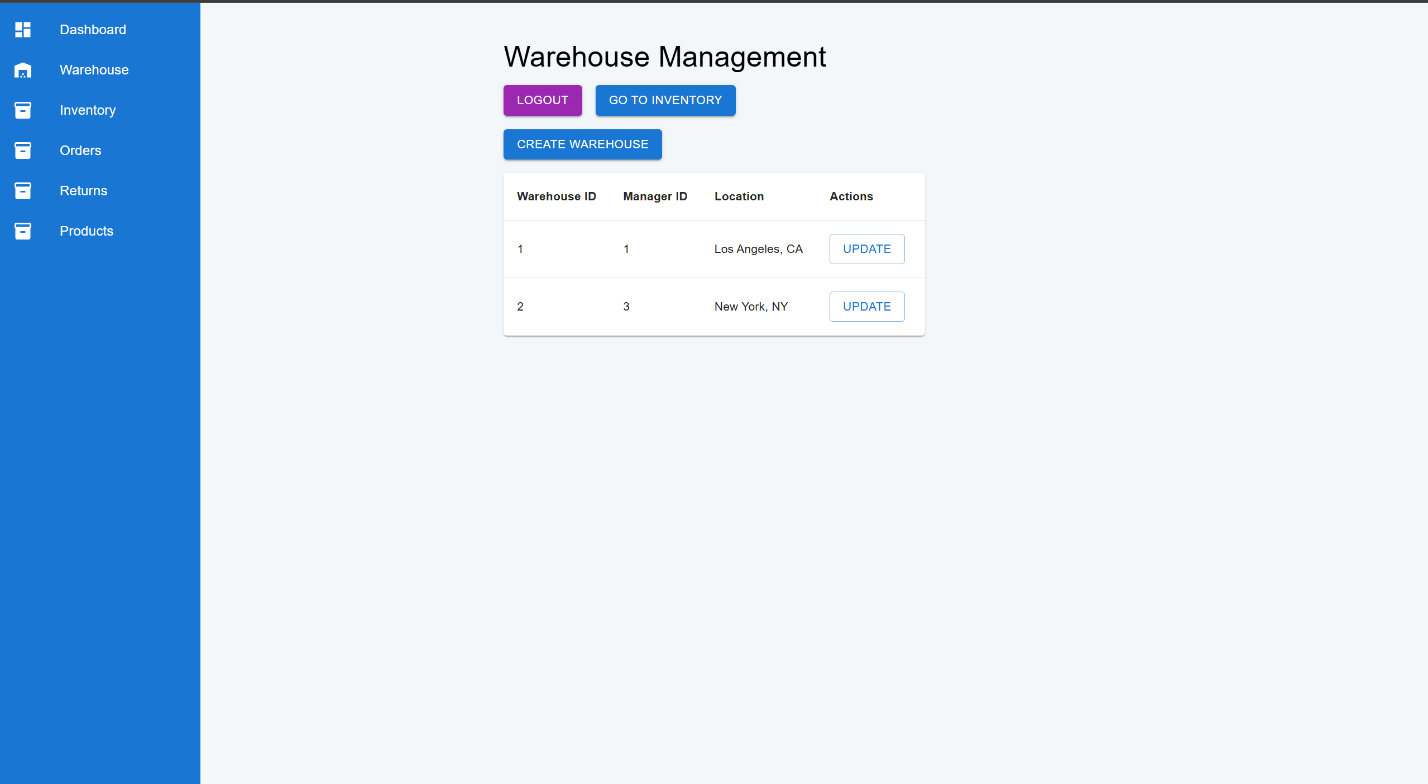


Figure 18: Warehouse Management Page

The warehouse management page is a dedicated interface for handling warehouse-related operations and data. It includes a sidebar navigation menu that allows administrators to seamlessly move between different sections of the system, such as the dashboard, inventory, orders, returns, and products. The primary section of the page features a structured table displaying all warehouses in the system, with columns for warehouse ID, location, and actions. Each row in the table corresponds to a specific warehouse, allowing administrators to quickly view and manage warehouse details. The "Actions" column provides interactive options to update existing warehouse entries, while a "Create Warehouse" button at the top of the page facilitates the addition of new warehouses to the system.

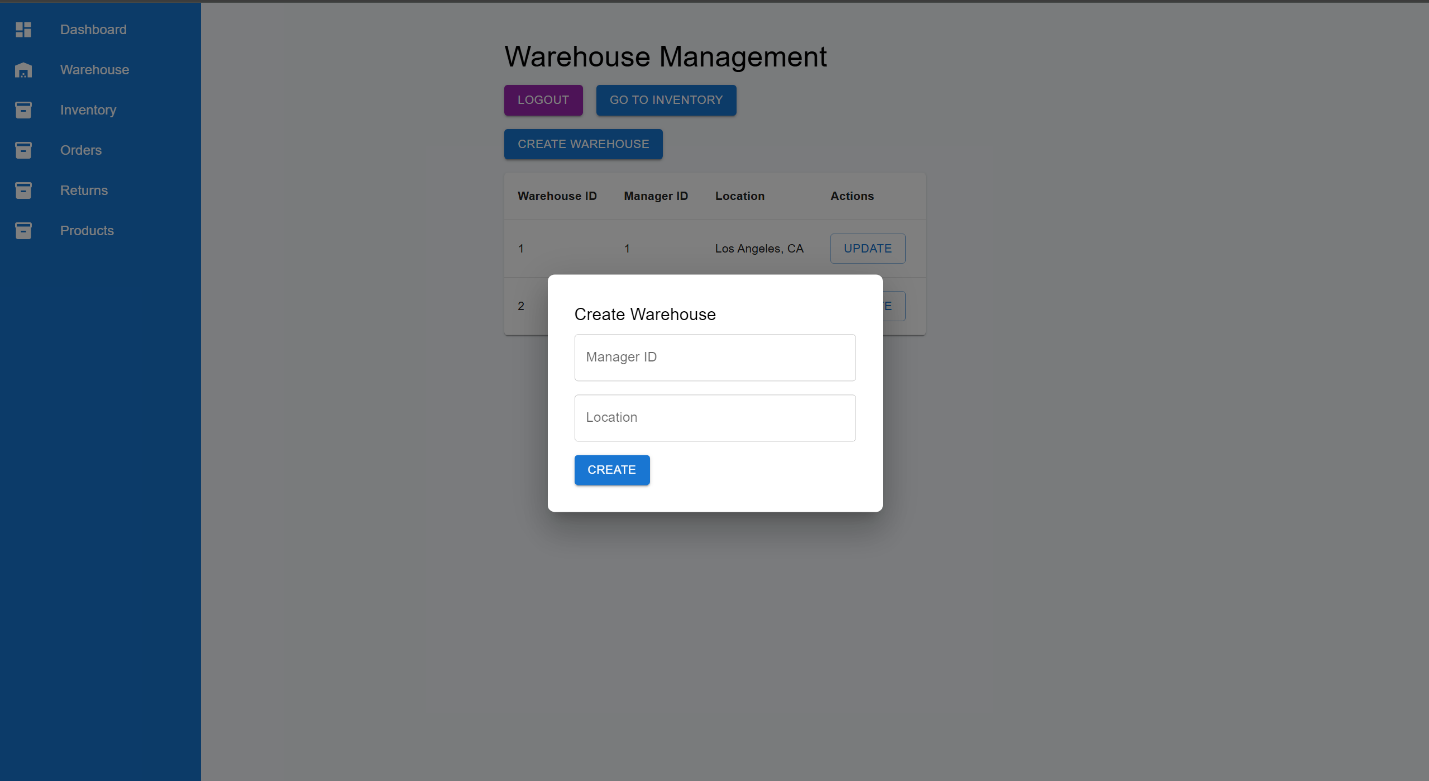


Figure 19: Warehouse Creation Modal

The warehouse management page includes a modal for adding new warehouses. Triggered by the "Create Warehouse" button, the modal contains fields for "Manager ID" and "Location," along with a "Create" button to submit the details. Upon submission, the new warehouse is added to the database and displayed in the table. The design ensures quick and efficient data entry without leaving the page.

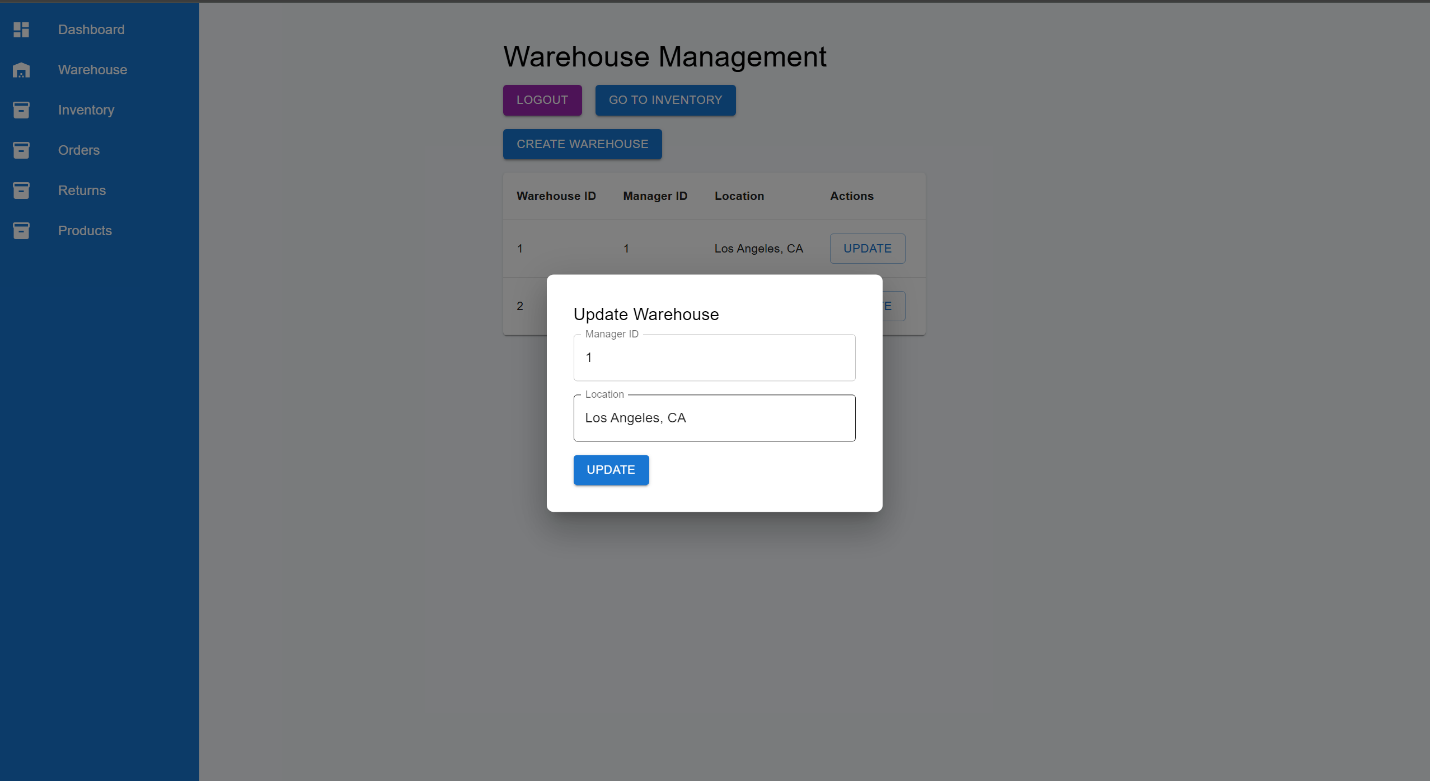


Figure 20: Warehouse Update Modal

The warehouse management page includes an update modal, triggered by the "Update" button in the ”Actions” column. The modal allows editing of "Manager ID" and "Location" fields for the selected warehouse. An "Update" button submits the changes, which are saved to the database and reflected in the table.

## Inventory Management

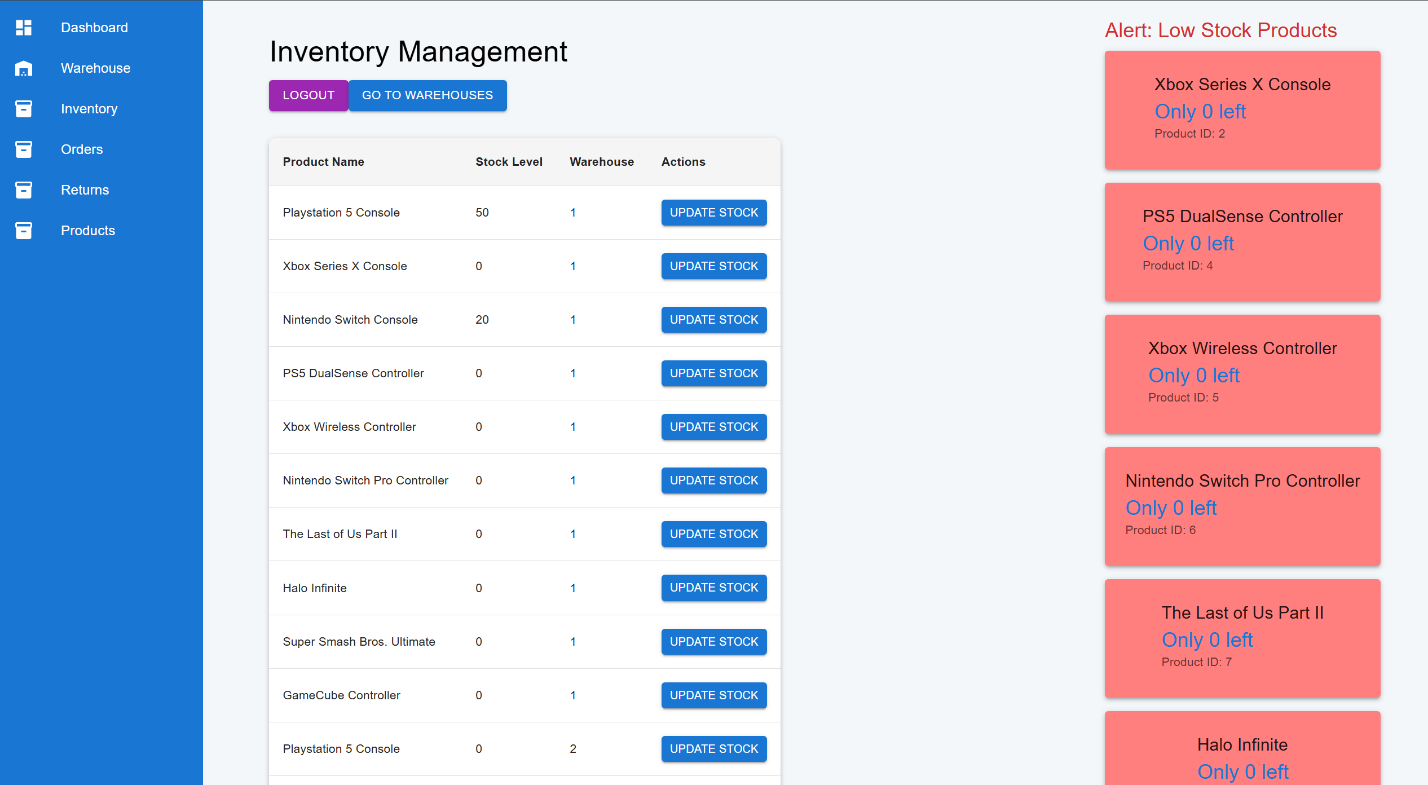


Figure 21: Inventory Management Page

The inventory management page provides a detailed view of product stock levels and includes an alert system for low-stock items. On the left, the inventory table lists products with columns for the product name, stock level, associated warehouse, and action buttons to update stock. On the right, a red-highlighted alert panel displays products with critically low stock, including their names and remaining quantities. This layout allows administrators to monitor inventory status immediately and quickly address stock issues.

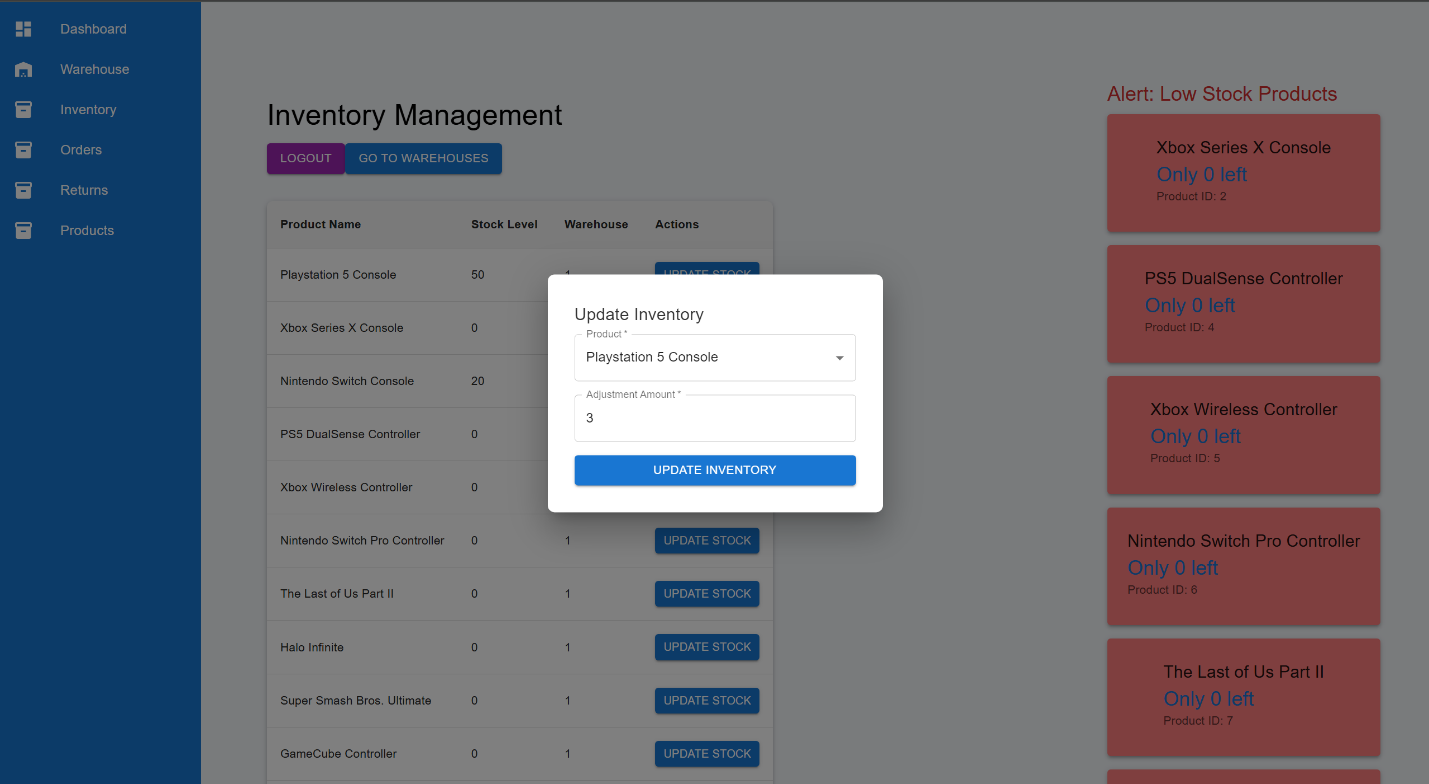


Figure 22: Inventory Update Page

The inventory management page includes an "Update Inventory" modal, triggered by the "Update Stock" button in the ”Actions” column. The modal allows administrators to update the stock level for a selected product and choose its associated warehouse. An "Update Inventory" button submits the changes, which are then reflected in the inventory table.

## Order Management

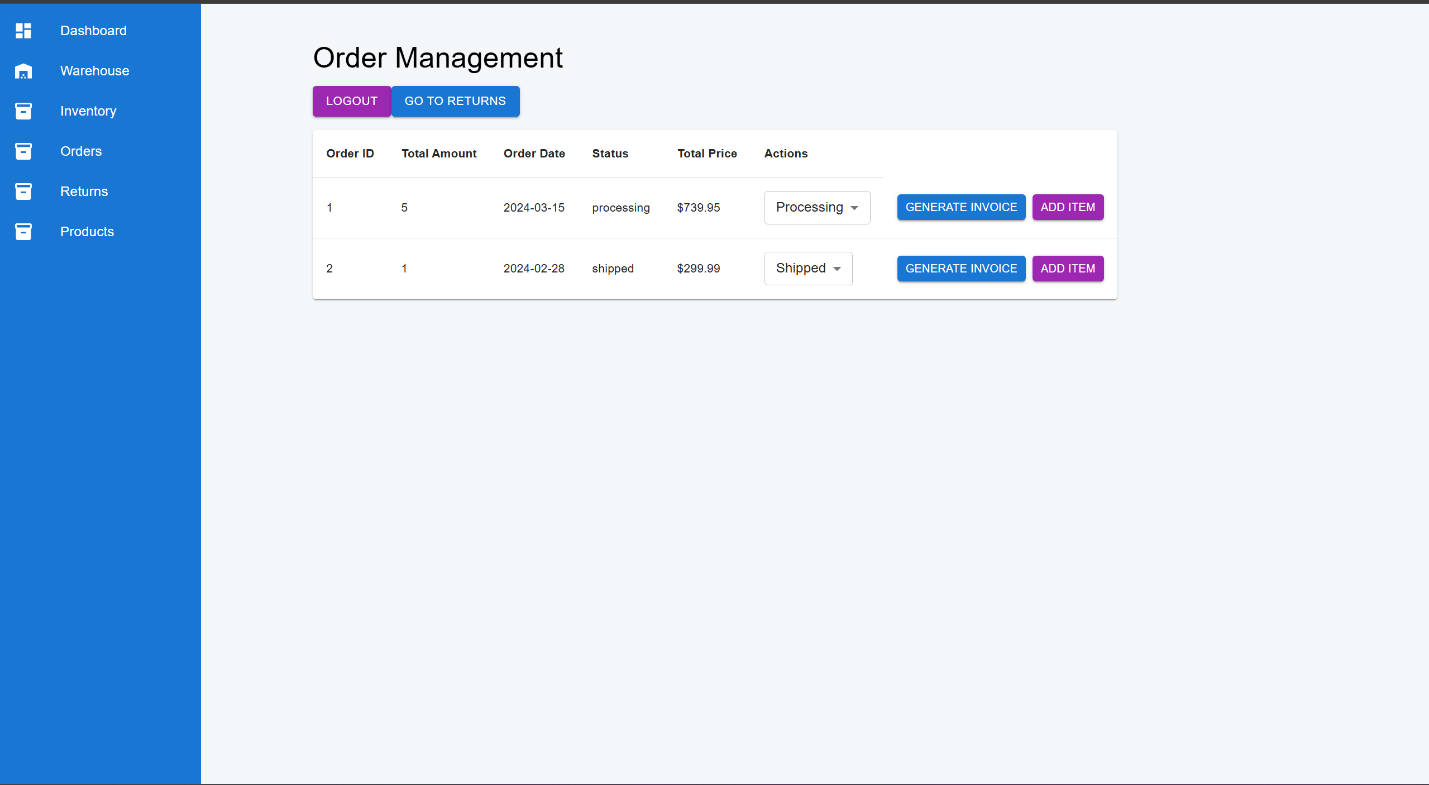


Figure 23: Order Management Page

The order management page displays a table of all orders with columns for order ID, total amount, order date, status, and total price. The actions column includes buttons to "Generate Invoice" and "Add Item," allowing administrators to create invoices or add products to specific orders. The page provides a clear overview of orders, their statuses, and tools for efficient order management.

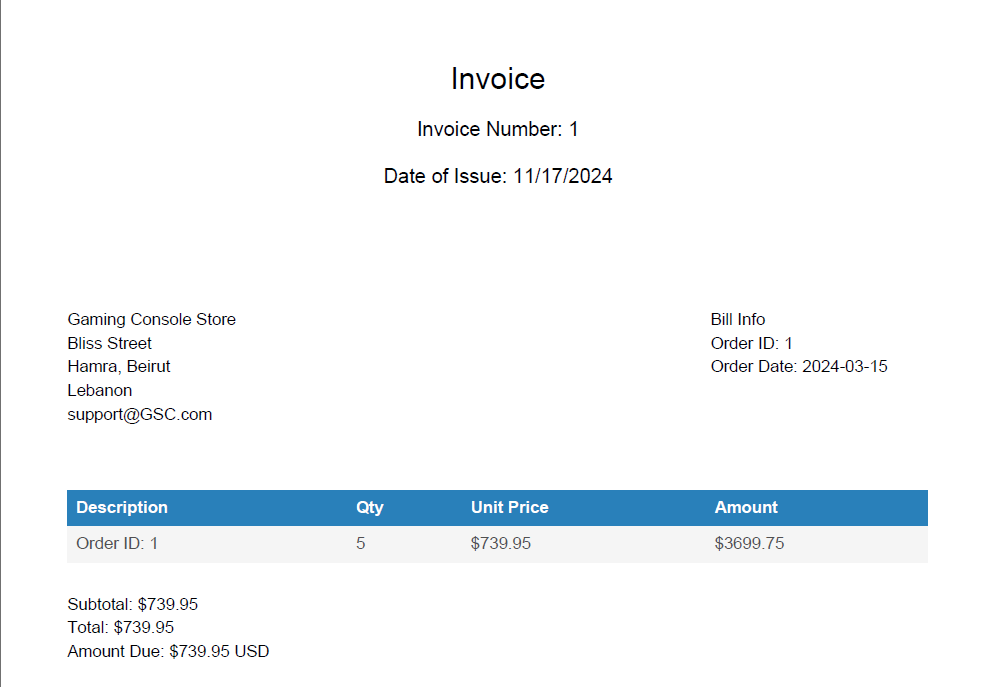


Figure 24: Invoice Generation

The invoice page provides a detailed summary of a customer's order. It includes the invoice number, the date of issue, and the store’s contact information at the top left. On the right, the bill details display the order ID and the order date. The main table outlines the order's items, showing the description, quantity, unit price, and total amount for each product. Below the table, the subtotal, total amount, and amount due are clearly displayed. The layout is professional and ensures all necessary billing details are presented in a concise format.

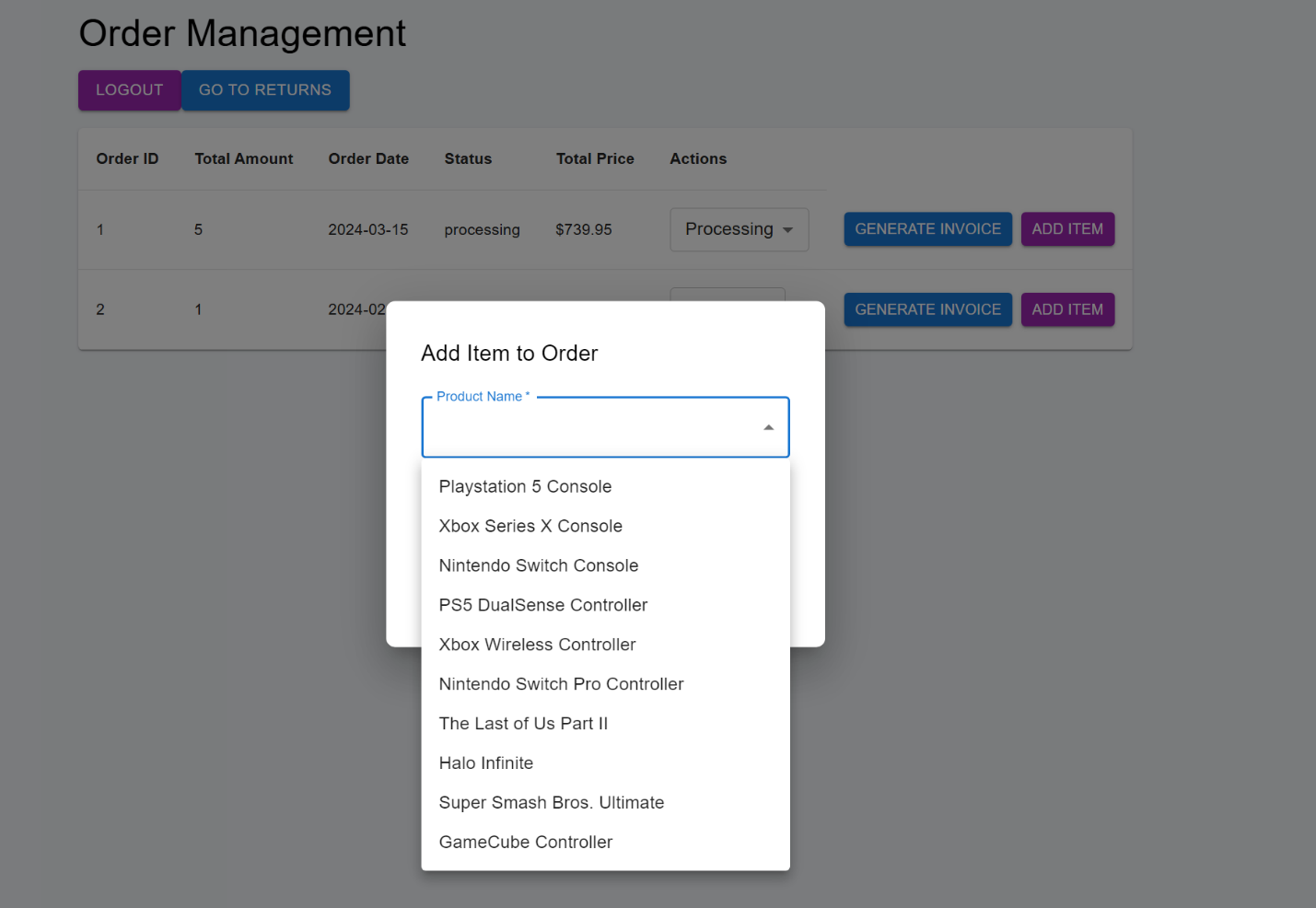


Figure 25: Add Item to Order Modal

The order management page includes an "Add Item to Order" modal, which is triggered by the "Add Item" button in the ”Actions” column. The modal features a dropdown menu listing available products, allowing administrators to select a product to add to the order. This ensures efficient management of order items while keeping the workflow streamlined. Once a product is selected, the item is added to the corresponding order in the database.

## Returns Management

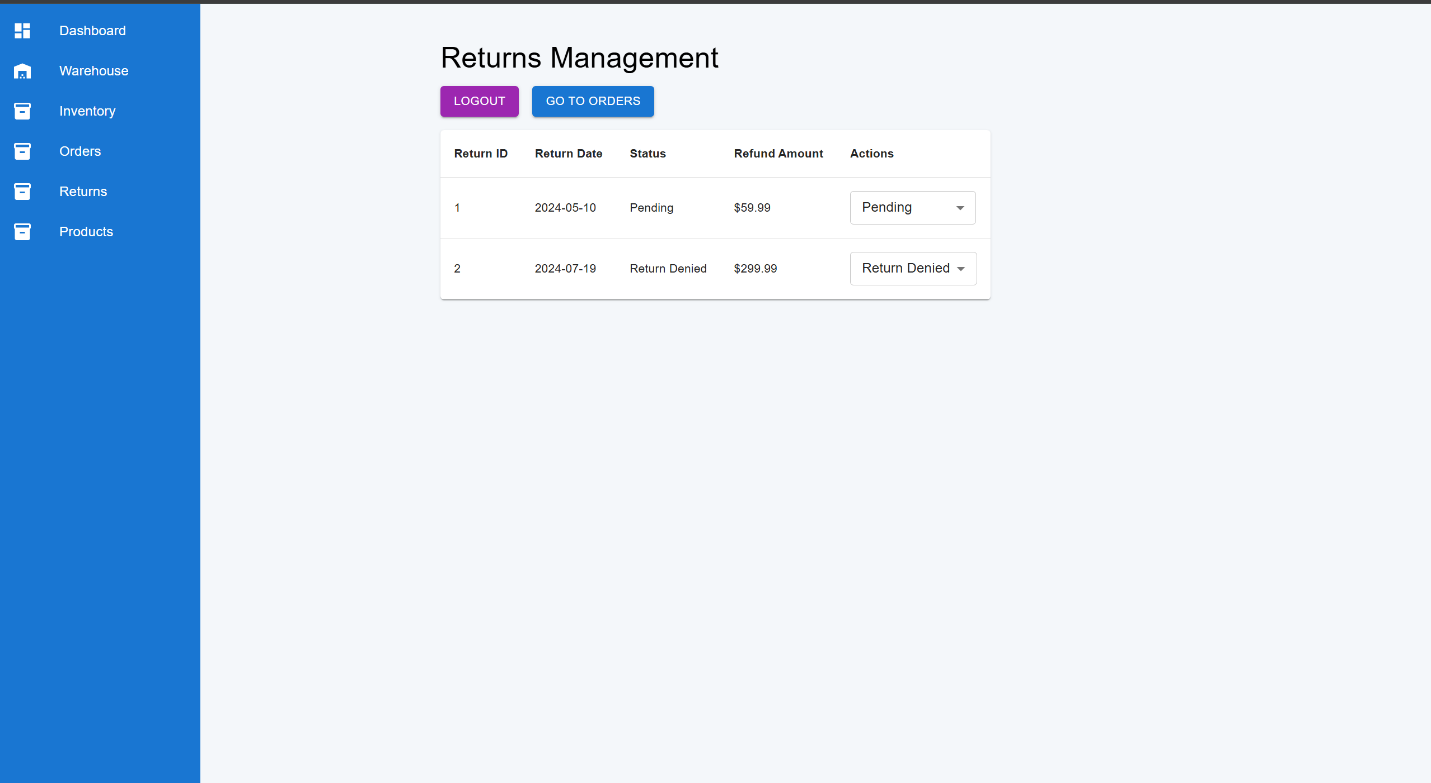


Figure 26: Returns Management Page

The returns management page provides an overview of customer returns. It includes a table with columns for return ID, return date, status, refund amount, and actions. The status column allows administrators to update the status of a return using a dropdown menu, while the actions column may include options like "Approve" or "Deny" the return. This page helps streamline the handling of returns and refunds efficiently.

## Product Management

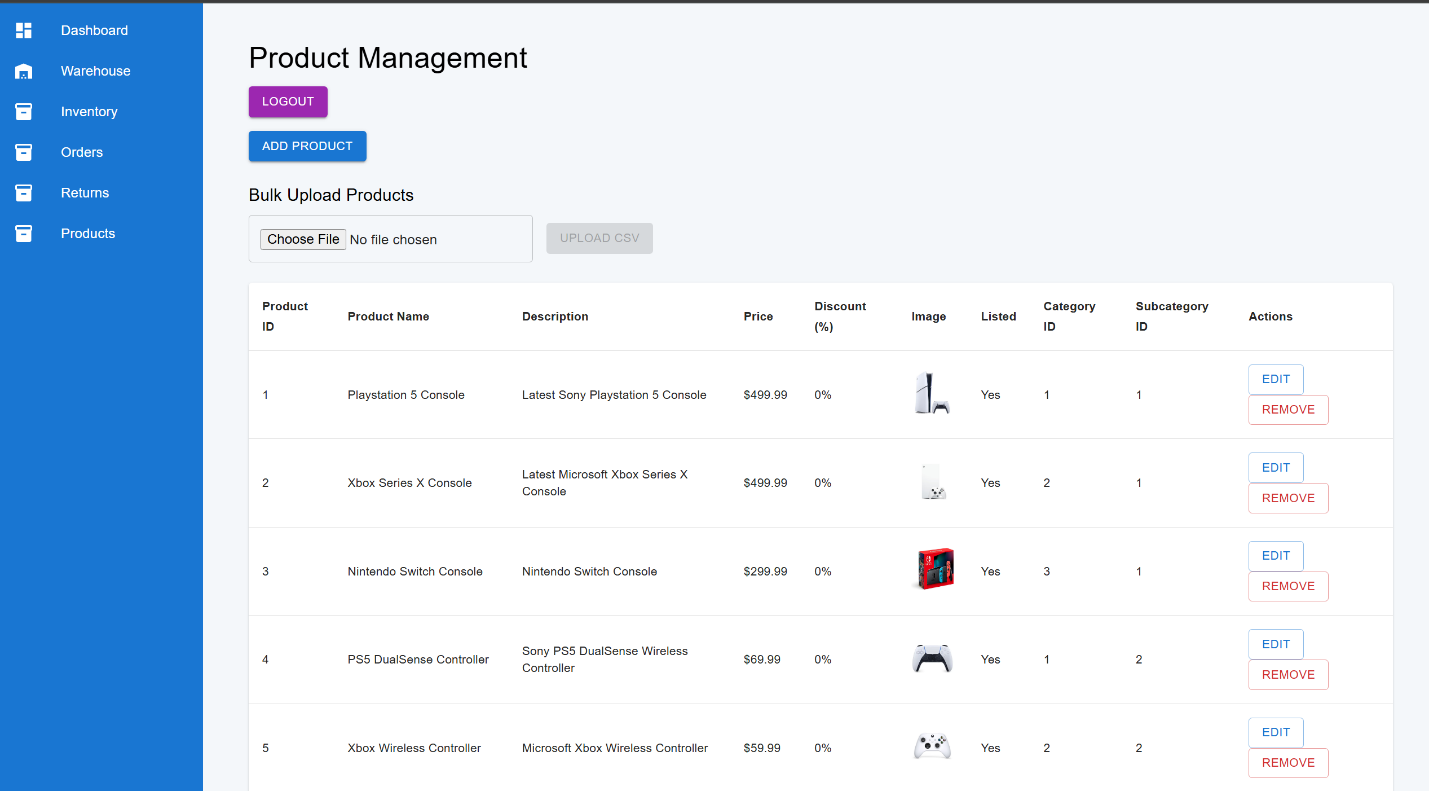


Figure 27: Product Management Page

The product management page allows administrators to manage the product catalog. The table lists products with columns for product name, description, price, discount, stock level, category, and actions. The actions column includes "Edit" and "Remove" buttons for updating or deleting products. At the top, there are options for adding new products individually using the "Add Product" button or uploading multiple products using the "Bulk Upload Products" feature. This page ensures efficient product management with a clear and structured layout.

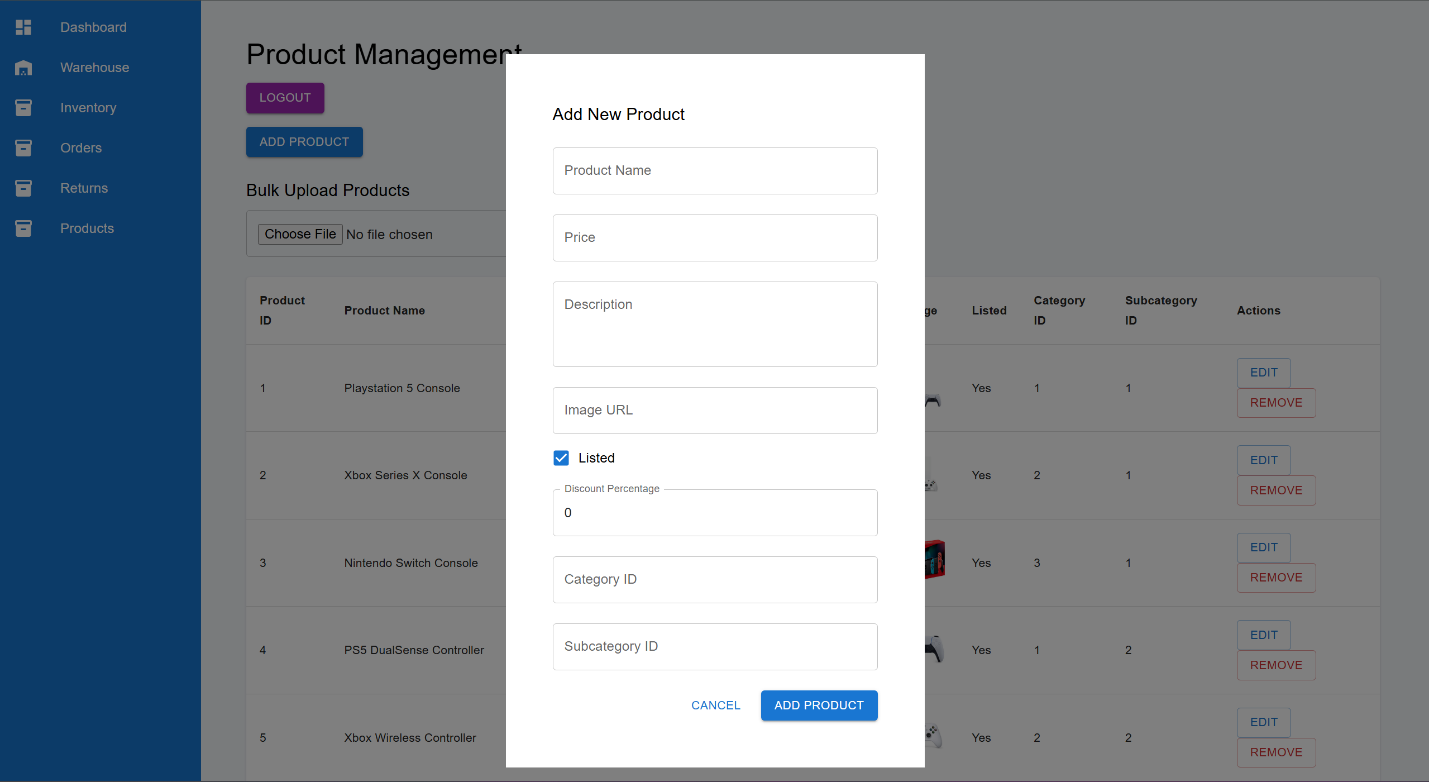


Figure 28: Add New Product Modal

The product management page includes an "Add New Product" modal, triggered by the "Add Product" button. The modal contains fields for entering product details, such as name, price, description, image URL, listed status, discount percentage, category ID, and subcategory ID. A "Cancel" button closes the modal, while the "Add Product" button submits the entered information, adding the product to the database. This feature streamlines the process of adding new products to the inventory.