# PROJECT

## Warehouse Management System

Submitted By,
Shubham Lathiya (SCA24801030058),
Priyank Mangaroliya (SCA24801030064)

Guided By,
Prof. Abhijita Puhan,
Dr. Kamlesh Meshram

For partial fulfillment of the requirements
For the Degree of Master of Computer Application
School of Engineering and Technology,
Pimpri Chinchwad University.
November, 2024.

# Acknowledgment

We extend our heartfelt gratitude to all those who contributed to the successful completion of our project on the Warehouse Management System.

We sincerely thank Prof. Abhijita Puhan, whose guidance and encouragement have been invaluable throughout the project. Their expertise and constructive feedback helped us navigate challenges and improve our work.

We are grateful to school of engineering and technology for providing the necessary resources and a conducive environment for carrying out this project. The infrastructure and support from the institution played a significant role in achieving our objectives. Lastly, we acknowledge the efforts of our team members and the support of our peers and family, whose cooperation and motivation inspired us to excel in this endeavour.

# Abstract

This Warehouse Management System (WMS) project aims to enhance the efficiency, accuracy, and productivity of warehouse operations by addressing challenges such as inaccurate inventory records, late order processing, and labor-intensive tasks. Designed for a single warehouse, the WMS enables real-time inventory tracking, streamlined order fulfillment, and optimized space utilization.

The project employs an Agile methodology, ensuring iterative development and client feedback integration. The system utilizes a robust tech stack, including Flask for the backend, MongoDB for scalable data storage, and HTML, CSS, and JavaScript for a responsive user interface. Key features include user management, inventory tracking, automated order processing, efficient picking and packing, shipping, and detailed reporting with analytics.

Expected outcomes include reduced errors, minimized costs, and data-driven decision-making, resulting in improved operational efficiency, accurate order fulfillment, and enhanced customer satisfaction.

# Index

# 1) Introduction

## 1.1) Problem Statement

- Managing warehouse operations effectively is challenging. Common issues include inaccurate inventory tracking, inefficient use of storage space, slow order processing, and a lack of real-time visibility into stock levels. These problems lead to higher costs, delays, and dissatisfied customers.

- A Warehouse Management System (WMS) is needed to automate tasks, improve inventory accuracy, optimize space utilization, and streamline order fulfilment. This will enhance efficiency, reduce costs, and improve customer satisfaction.

## 1.2) Objective

- This Warehouse Management System (WMS) is designed to help warehouse operations be more efficient, accurate, and productive. Using this system, user can track inventory in real time, streamline order fulfillment, and optimize warehouse space. WMS reduces operational costs, minimizes errors, and improves workflow by automating key tasks like inventory receiving, picking, packing, and shipping.

- We want to solve common problems such as inaccurate inventory records, late order processing, and labor-intensive operations. It'll also give businesses data-driven insights to make better decisions. In the end, we want to make sure orders are fulfilled accurately and on time, so we can maximize warehouse efficiency and profits.

## 1.3) Scope

- The scope of this Warehouse Management System (WMS) is limited to managing the operations of a single company's warehouse, focusing on inventory management, order processing, picking, packing, shipping, reporting and analytics. It will generate detailed reports and analytics for better decision-making.

- The project will not include multi-warehouse management, or complex integrations. It will not include return and replace options.

## 1.4) Organization of the Report

- **Introduction:** Describes warehouse management challenges, objectives of the proposed WMS, and its scope, focusing on improving efficiency, reducing errors, and optimizing operations for a single warehouse.

- **Methodology:** Explains the Agile development approach and the tech stack, including Flask for the backend, MongoDB for scalable storage, and a responsive UI using HTML, CSS, and JavaScript.

- **System Design and Features:** Covers system architecture and core functionalities like user management, real-time inventory tracking, automated order processing, shipping, and detailed analytics.

- **Implementation:** Highlights the development process, focusing on inventory management, order workflows, and reporting integration.

- **Expected Outcomes:** Discusses benefits such as enhanced efficiency, reduced costs, data-driven decisions, and improved customer satisfaction.

- **Limitations and Future Scope:** Outlines the exclusions, like multi-warehouse management and return processing, with recommendations for future enhancements.

- **Conclusion:** Summarizes the importance of the WMS in addressing operational challenges and achieving better performance.

# 2) Literature Survey/Review

## 2.1) Overview of existing

- Warehouse Management Systems (WMS) streamline inventory, order fulfillment, and logistics through various types of solutions. Legacy systems are robust but lack flexibility, while cloud-based WMS offers scalability, real-time updates, and accessibility. ERP-integrated WMS ensures unified business operations, and AI-powered systems optimize processes with automation and demand prediction. Industry-specific solutions address compliance and traceability, while open-source WMS provides cost-effective, customizable options.

## 2.2) Project Reviewed

- The project focuses on developing an efficient Warehouse Management System (WMS) to optimize inventory control, order processing, and logistics. It integrates advanced technologies like cloud computing for scalability, AI for automation and demand forecasting, and IoT for real-time tracking. Existing systems such as legacy, cloud-based, ERP-integrated, and industry-specific WMS were reviewed to emphasizing the need for a flexible, scalable, and cost-effective solution.

## 2.3) Identification of gaps

- Key gaps in existing WMS include limited scalability in legacy systems, high implementation costs, and challenges with integration and customization. Many systems lack real-time visibility, have complex adoption processes requiring technical expertise, and security concerns in cloud-based solutions. Addressing these gaps can enhance WMS accessibility, efficiency, and adaptability.

# 3) System Analysis

## 3.1) Requirements Specification

### 3.1.1) Functional Requirements

### Admin and Manager:

1) **Login:** Allows the admin and manager to securely access the system using authentication.

2) **View Dashboard:** Displays an overview of key warehouse metrics and activities in a user-friendly interface.

3) **Manage Users:** Enables the admin to add, update, and delete user accounts and roles within the system.

4) **Manage Products:** Provides functionality to add, update, and delete product details.

5) **Manage Stock:** Allows the user to track inventory levels and update stock details in real-time.

6) **View Clients:** Displays a list of clients associated with the warehouse operations.

7) **Manage Orders:** Facilitates tracking and delivery of orders, ensuring timely and accurate order fulfillment.

8) **View Reports:** Generates detailed analytics and reports for decision-making.

9) **View Profile:** Lets the user update their profile information and reset passwords.

10) **Manage Permission:** Enables the admin to assign and manage user access permissions within the system.

## Employee:

1) **Login:** Allows the employee to securely access the system using authentication.

2) **View Orders:** Allows the employee to check details of received orders, including status and packing orders.

3) **Manage Tasks:** Employees can view assigned tasks, update task status, and mark tasks as completed to streamline daily operations.

4) **View Profile:** Lets the user update their profile information and reset passwords.

**Supplier:**

1) **Login:** Allows the supplier to securely access the system using authentication.

2) **View Orders:** Allows the supplier to check details of received orders, including status and delivery requirements.

3) **Deliver Order:** Provides functionality for suppliers to mark orders as delivered after completing the shipment.

4) **Receive Payment:** Enables the supplier to confirm and track payments received for delivered orders.

5) **Print Invoice:** Allows the supplier to generate and print invoices for delivered orders.

6) **View Profile:** Lets the user update their profile information and reset passwords.

## Client:

1) **Register:** Clients must register an account by providing details and verifying their identity through an OTP (One-Time Password).

2) **Login:** Clients log in securely using their credentials, with authentication ensuring authorized access.

3) **View Products:** Clients can browse and view a list of available products, including details like quantity and price.

4) **Add Orders:** Clients can place new orders by selecting products and specifying quantities.

5) **Track Order:** Clients can monitor the status and delivery progress of their orders.

6) **View Order History:** Clients can access a detailed history of past orders, including statuses and payment details.

7) **View Profile:** Lets the user update their profile information and reset passwords.

### 3.1.2) Non-functional Requirements

1) **Security and Data Protection:** All sensitive data, including user details, orders, and stock information, must be encrypted during storage and transmission. The system should implement role-based access control (RBAC) to ensure only authorized users can perform specific actions.

2) **Email Service:** The system should send email notifications to clients upon successful order placement, including order details and tracking links. Users should receive emails during registration, account updates, or password resets.

3) **Usability:** The interface should be intuitive and accessible, allowing users to perform tasks (e.g., tracking orders, managing stock) with minimal training.

4) **Availability:** The system must be available 99.9% of the time, ensuring continuous service with minimal downtime.

## 3.2) Feasibility Study

- **Technical Feasibility:** Python, MongoDB, and web frameworks (Flask) are used to build robust and scalable systems as part of the project. Using tools like PyCharm and MongoDB Compass speeds up the development process by enabling efficient coding, debugging, and data analysis.

- **Economic Feasibility:** Automating order processing and inventory management will improve warehouse efficiency, reduce operational costs, and enhance customer satisfaction. Open-source tools (Python, MongoDB) reduce development costs.

- **Operational Feasibility:** WMS features an intuitive interface designed for warehouse personnel and managers, enabling quick adoption. Role-based access control ensures proper and secure use of system features, minimizing operational risks.

## 3.3) Tools and Technologies

### Technologies:

- **Frontend:** HTML, CSS, and JavaScript will be used to create a responsive and interactive user interface.

- **Backend:** The backend will be handled by Python using Flask, providing a robust environment for API integrations and server-side logic.

- **Database:** Data will be stored in MongoDB for scalability, quick data retrieval, and inventory data handling.

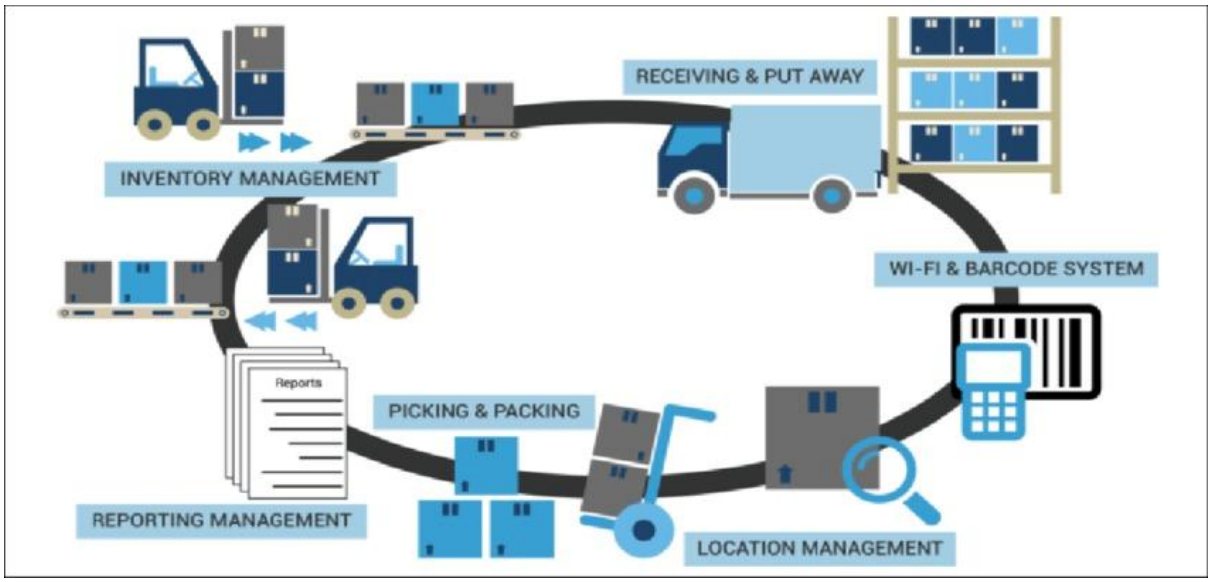
### Tools:

- **PyCharm:** The PyCharm IDE is an integrated development environment (IDE) for writing, testing, and debugging Python code. The WMS backend will be developed using PyCharm's features for efficient coding.

- **MongoDB:** The MongoDB Compass graphical user interface (GUI) allows users to query, aggregate, and analyze data visually. A user-friendly interface provides an intuitive environment for exploring and interacting with data.

# 4) Design

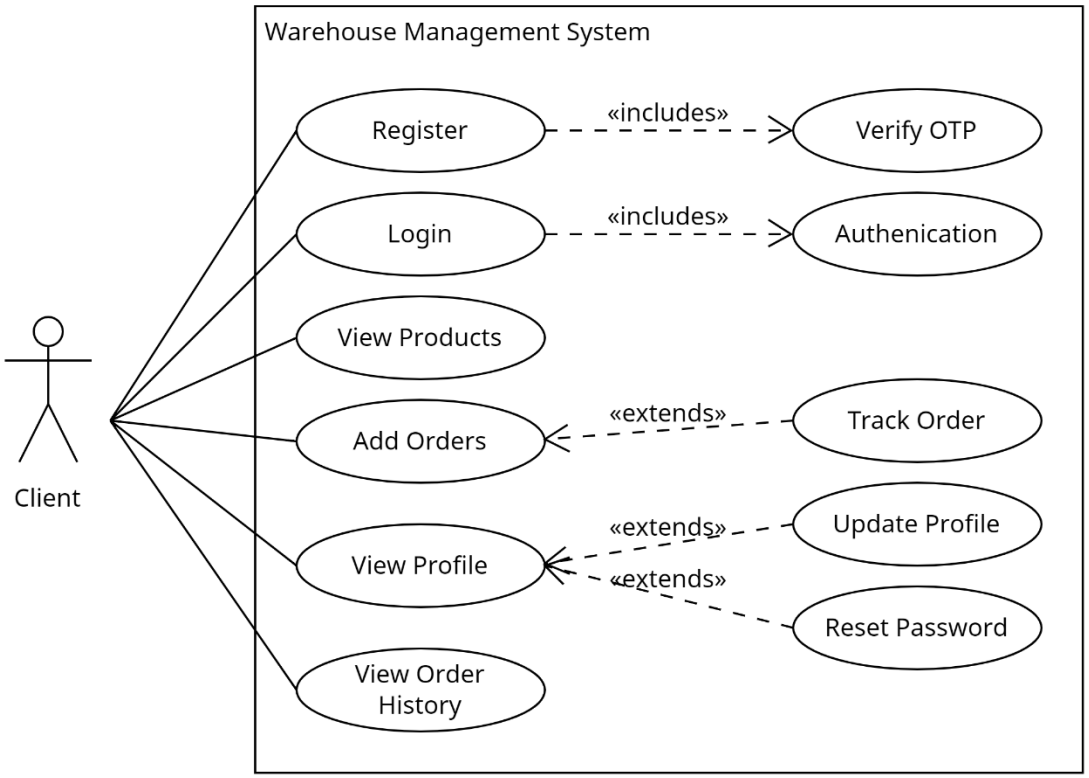
## 4.1) System Architecture




Warehouse Management System

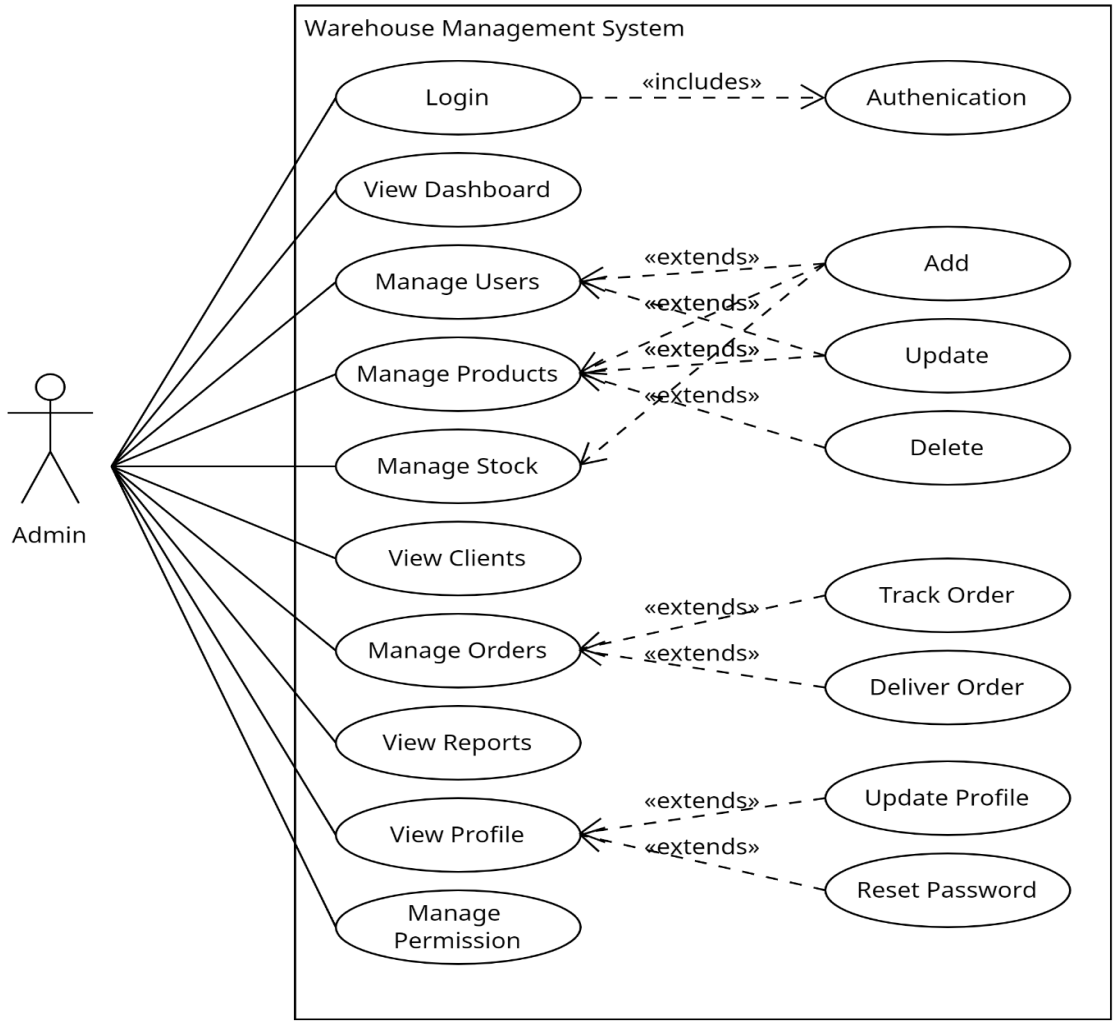
## 4.2) Diagrams

Client:



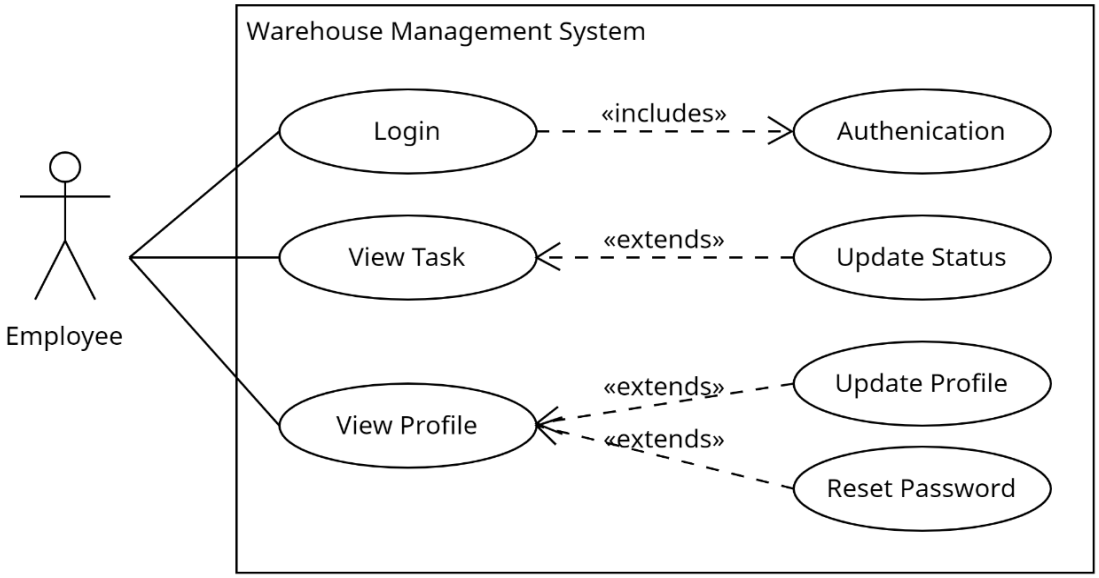

Client Use Case Diagram

## Admin:



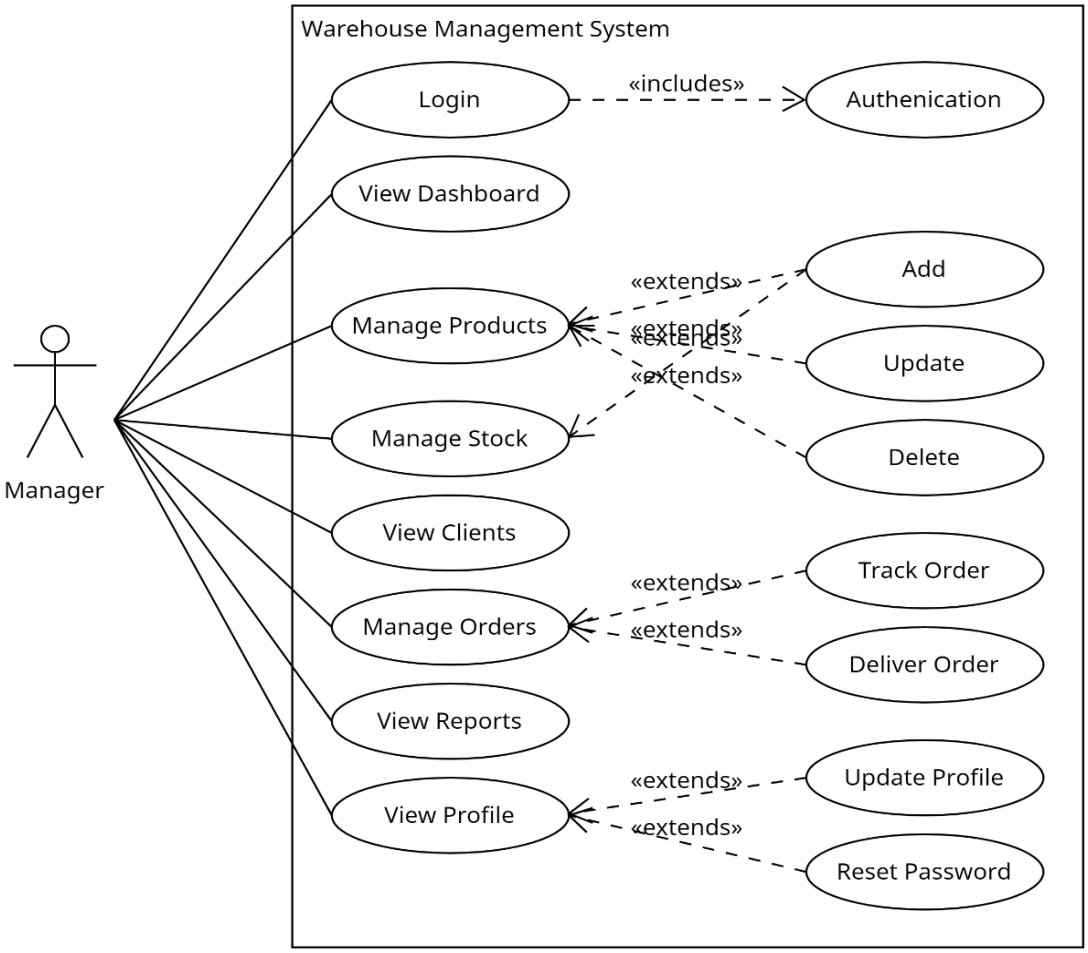

Admin Use Case Diagram

## Employee:



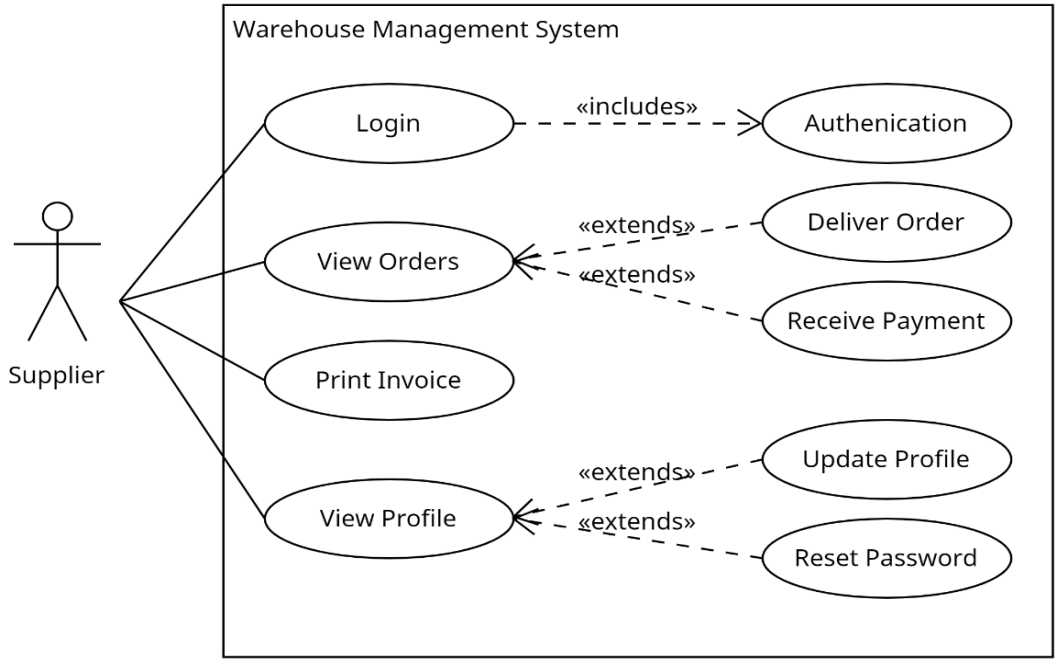

Employee Use Case Diagram

## Manager:




Manager Use Case Diagram

## Supplier:




Supplier Use Case Diagram

## 4.3) Database Design

**user**

| Attribute | Data Type | Constraints |
|---|---|---|
| _id | ObjectId | Primary Key, Not Null, Unique |
| full_name | String | Not Null |
| mobile | String | Not Null, Unique |
| email | String | Not Null, Unique |
| password | String | Not Null |
| role | String | Not Null |
| status | String | Not Null |
| city | String | Not Null |
| area | String | Not Null |

**product**

| Attribute | Data Type | Constraints |
|---|---|---|
| _id | ObjectId | Primary Key, Not Null, Unique |
| product_name | String | Not Null |
| sku | Long | Primary Key, Not Null, Unique |
| unit | Integer | Not Null |
| min | Integer | Not Null |
| max | Integer | Not Null |
| price | Float | Not Null |
| status | String | Not Null |
| description | String | Not Null |
| created_by | String | Not Null |
| area_id | ObjectId | Not Null, Foreign Key (References area(area_id)) |
| image | String | Not Null |
| created_at | Date | Not Null |

**area**

| Attribute | Data Type | Constraints |
|---|---|---|
| _id | ObjectId | Primary Key, Not Null, Unique |
| area_name | String | Not Null, Unique |
| no_box | Integer | Not Null |
| status | String | Not Null |

**stock**

| Attribute | Data Type | Constraints |
|---|---|---|
| _id | ObjectId | Primary Key, Not Null, Unique |
| sku | Long | Not Null, Unique, Foreign Key (References products(sku)) |
| date | Date | Not Null |
| +qty | Integer | Not Null |
| -qty | Integer | Not Null |
| total_qty | Integer | Not Null |

**order**

| Attribute | Data Type | Constraints |
|---|---|---|
| _id | ObjectId | Primary Key, Not Null, Unique |
| user_id | ObjectId | Not Null, Foreign Key (References users(user_id)) |
| products | Array of Object | Not Null |
| total_amount | Float | Not Null |
| payment_type | String | Not Null |
| order_date | Date | Not Null |
| status | Array of Object | Not Null |
| otp | Integer | Not Null |

## transactions

| Attribute | Data Type | Constraints |
|---|---|---|
| _id | ObjectId | Primary Key, Not Null, Unique |
| order_id | ObjectId | Not Null, Foreign Key (References orders(order_id)) |
| amount | Float | Not Null |
| payment_type | String | Not Null |
| payment_status | String | Not Null |
| transaction_date | Date | Not Null |

## case_callections

| Attribute | Data Type | Constraints |
|---|---|---|
| _id | ObjectId | Primary Key, Not Null, Unique |
| order_id | ObjectId | Not Null, Foreign Key (References orders(order_id)) |
| amount | Float | Not Null |
| submission_date | Date | Not Null |
| supplier_id | ObjectId | Not Null, Foreign Key (References suppliers(supplier_id)) |
| status | String | Not Null |

## page_visibility

| Attribute | Data Type | Constraints |
|---|---|---|
| _id | ObjectId | Primary Key |
| page_name | String | Not Null, Unique, |
| roles | Object | Not Null |

## assign_tasks

| Attribute | Data Type | Constraints |
|---|---|---|
| _id | ObjectId | Primary Key, Not Null, Unique |
| order_id | ObjectId | Not Null, Foreign Key (References orders(order_id)) |
| employee_id | ObjectId | Not Null, Foreign Key (References users(user_id)) |
| assigned_date | Date | Not Null |
| status | String | Not Null |
| supplier_id | ObjectId | Not Null, Foreign Key (References suppliers(supplier_id)) |

# 5) Implementation

## 5.1) Description of Modules/Components

- **User Management:** Admins can create, manage, and assign roles to users, such as warehouse managers and employees, using the User Management module of the Warehouse Management System (WMS).

- **Inventory Management:** The module handles inventory tracking, management, and updates within a warehouse. It allows real-time stock monitoring, keeping track of product quantities and locations, as well as stock adjustments and low stock alerts.

- **Order Processing:** This module streamlines order handling, from order creation to fulfillment. This module automates order processing, so warehouse employees can get orders, allocate inventory, update order statuses, and prioritize order processing based on urgency and customer needs.

- **Picking and Packing:** A module that enables efficient picking and packing of orders. It guides warehouse staff through the most efficient routes based on inventory location and order priority.

- **Shipping and Dispatch:** Order fulfillment includes preparing shipments, generating shipping labels, and dispatching products to customers.

- **Reporting and Analytics:** A warehouse admin can generate detailed reports on inventory levels, order processing times, picking efficiency, and overall warehouse productivity with this module, which provides detailed reports and analytics to aid in data-driven decision-making.

## 5.2) Code Snippets

### 1) Mail Sent Code:

```
# Flask-Mail Configuration
app.config['MAIL_SERVER'] = 'smtp.gmail.com'
app.config['MAIL_PORT'] = 465
app.config['MAIL_USERNAME'] = 'shubhamlathiya2021@gmail.com'
app.config['MAIL_PASSWORD'] = 'tqerujnjzuvgdjho'
```

```
app.config['MAIL_USE_TLS'] = False
app.config['MAIL_USE_SSL'] = True

# Initialize the mail object
mail = Mail(app)

from email.mime.text import MIMEText
from flask_mail import Message, Mail
from flask import current_app

mail = Mail()

def send_email(subject, recipient_email, body_html):
  try:
    # Create the Flask-Mail message object
    msg = Message(subject=subject,
            sender=current_app.config['MAIL_USERNAME'],
            recipients=[recipient_email])

    # Attach the HTML body to the message
    msg.html = body_html

    # Send the email using Flask-Mail
    mail.send(msg)

    return True
  except Exception as e:
    print(f"Error sending email: {e}")
    return False
```

**2) Payment Code:**

```
const razorpayOptions = {
                key: 'rzp_test_51mQEdclvr946E', // Replace with your
Razorpay key
                amount: data.amount, // Amount in paise (multiply by 100
to convert INR to paise)
                currency: 'INR',
```

```
                name: 'Your Store Name',
                description: 'Order Payment',
                order_id: data.razorpay_order_id, // Razorpay order ID
from backend
                handler: function (response) {
                  // Step 3: Submit order data directly after payment
success
                  fetch('/client/submitOrder', {
                    method: 'POST',
                    headers: {
                      'Content-Type': 'application/json'
                    },
                    body: JSON.stringify({
                      products: orderList,  // Array of products in the
order
                      totalAmount: orderData.totalAmount,
                      paymentType: 'Razorpay', // Payment type is
Razorpay
                      razorpay_payment_id:
response.razorpay_payment_id,  // Razorpay Payment ID
                      razorpay_order_id: response.razorpay_order_id,  //
Razorpay Order ID
                      razorpay_signature: response.razorpay_signature  //
Razorpay Signature
                    })
                  })
                    .then(orderRes => orderRes.json())
                    .then(orderData => {
                      if (orderData.status === 'success') {
                        alert('Order placed successfully!');
                        window.location.href = orderData.url; // Redirect
to the success page
                      } else {
                        alert(orderData.message);  // Error handling if
order submission fails
                      }
                    })
```

```
            .catch(error => {
                console.error('Error submitting order:', error);
                alert('There was an error placing your order.');
            });
        },
        prefill: {
            name: 'Customer Name',
            email: 'customer@example.com',
            contact: '9999999999'
        },
        notes: {
            address: 'Customer Address'
        },
        theme: {
            color: '#F37254'
        }
    };

    // Open Razorpay payment gateway
    const razorpay = new Razorpay(razorpayOptions);
    razorpay.open();
```

**3) QR Code Scanner Code:**

```
import cv2
from pyzbar.pyzbar import decode
import os

# Beep sound parameters (only for Windows)
def play_beep():
    if os.name == 'nt':  # Only play beep on Windows
        import winsound
        frequency = 1000  # Set Frequency To 1000 Hertz
        duration = 500    # Set Duration To 500 ms (0.5 seconds)
        winsound.Beep(frequency, duration)
    else:
        # Alternative sound or print message on non-Windows systems
        print("\a")  # ASCII Bell character for simple beep sound
```

```
# Capture webcam
cap = cv2.VideoCapture(0)
if not cap.isOpened():
    print("Error: Could not open camera.")
    exit()

while cap.isOpened():
    success, frame = cap.read()
    if not success:
        print("Error: Could not read frame from camera.")
        break

    # Flip the image like a mirror image
    frame = cv2.flip(frame, 1)

    # Detect the barcode
    detectedBarcode = decode(frame)

    for barcode in detectedBarcode:
        # Decode the barcode data
        barcode_data = barcode.data.decode('utf-8')
        if barcode_data:
            print(f"{barcode_data}")
            # Display the barcode data on the frame
            cv2.putText(frame, barcode_data, (50, 50),
cv2.FONT_HERSHEY_COMPLEX, 2, (0, 255, 255), 2)

            # Play beep sound
            play_beep()

            # Save the frame containing the barcode as an image
            cv2.imwrite("code.png", frame)

            # Exit the program after detecting a barcode
            cap.release()
            cv2.destroyAllWindows()
```

```
            exit()

    # Show the video feed with the scanner window
    cv2.imshow('scanner', frame)

    # Press 'q' to quit the program manually
    if cv2.waitKey(1) == ord('q'):
        break

# Release the webcam and close windows
cap.release()
cv2.destroyAllWindows()
```

**4) Monitor Stock Level:**

```
def monitor_stock_levels():
    from app import mongo, app
    print("Scheduler started.")

    # Fetch all products from the products collection
    products = mongo.db.products.find({})

    for product in products:
        # Fetch the latest stock entry for this product
        stock_record = mongo.db.stock.find_one(
            {'sku': product['sku']}, sort=[('date', -1)]
        )

        if not stock_record:
            # If there's no stock record, continue to the next product
            continue

        current_stock_qty = stock_record['total_qty']
        min_stock_threshold = product.get('min', 10)  # Fetch min threshold or
default to 10
        notification_sent = product.get('notification_sent', False)  # Check if
notification has already been sent
        # Check if current stock is below the threshold
```

```
        if current_stock_qty < min_stock_threshold:
            if not notification_sent:
                # Send notification only if it hasn't been sent already
                send_notification(product, current_stock_qty, min_stock_threshold)
                # Update the product to mark that the notification has been sent
                mongo.db.products.update_one(
                    {'_id': ObjectId(product['_id'])},
                    {'$set': {'notification_sent': True}}
                )
                print(f"Low stock notification sent for {product['product_name']}
(SKU: {product['sku']}) (stock : {current_stock_qty}")
            else:
                print(f"Notification already sent for {product['product_name']}
(SKU: {product['sku']})")
        else:
            # If stock is replenished, reset the notification_sent field
            if notification_sent:
                mongo.db.products.update_one(
                    {'_id': ObjectId(product['_id'])},
                    {'$set': {'notification_sent': False}}
                )
                print(f"Stock replenished for {product['product_name']} (SKU:
{product['sku']}), notification flag reset.")
            else:
                print(f"Stock is sufficient for {product['product_name']} (SKU:
{product['sku']})")
    print("Scheduler stopped.")
```

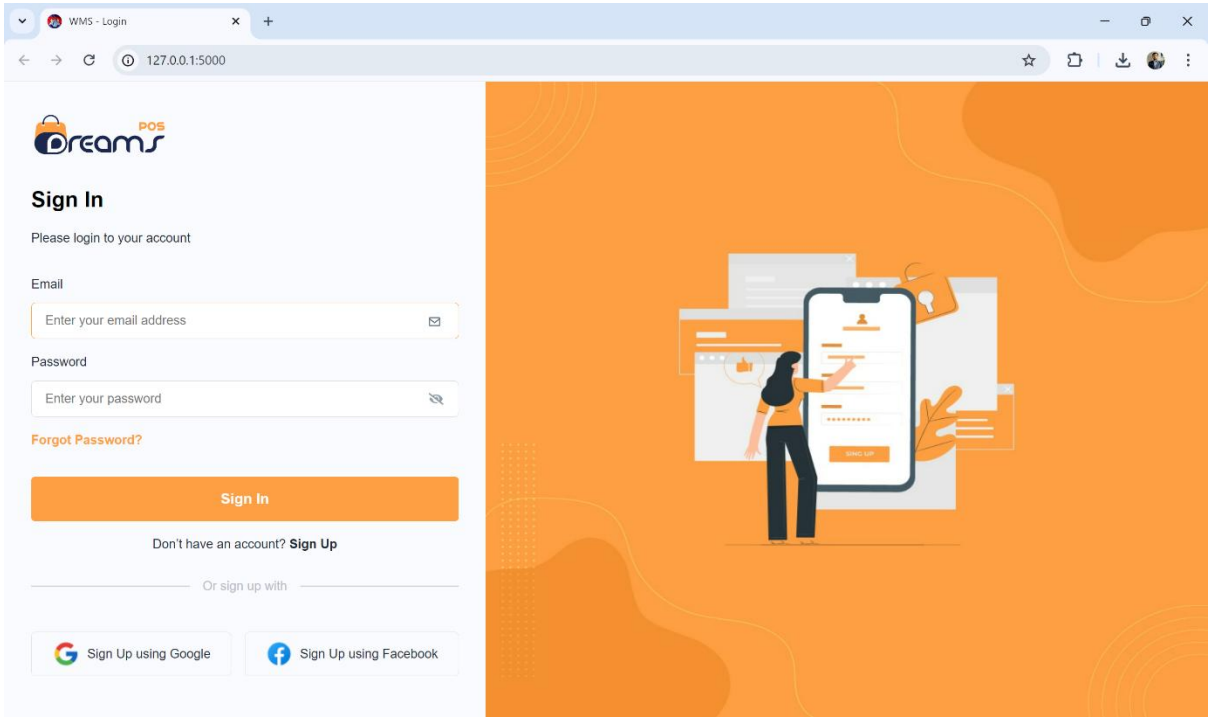
**5) Assign Orders for Employee:**

```
def assign_order_to_employee(order_id):
    # Fetch the queue of employees who have role "employee" and status
"true"
    employees = list(mongo.db.users.find(
        {'role': 'employee', 'status': 'true'}))  # Assuming the role and status are
in 'users' collection
    if not employees:
```

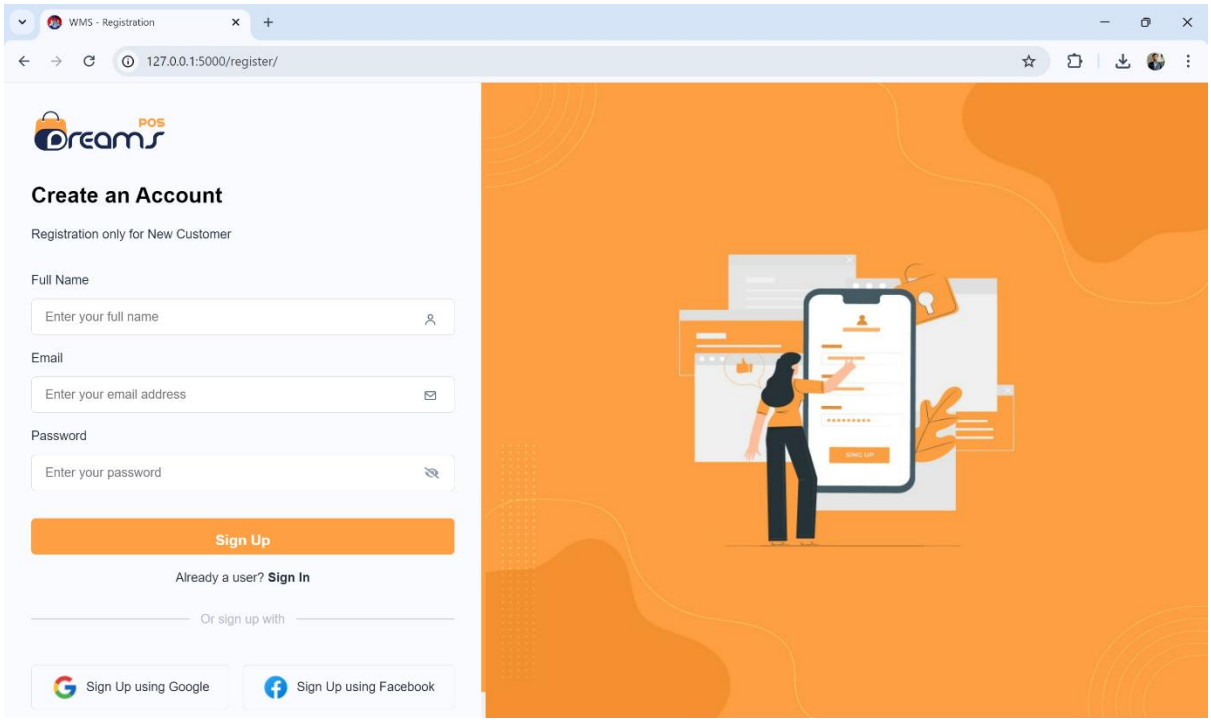
```
        return None

    # Fetch the current queue pointer (index of the last assigned employee)
    queue_data = mongo.db.queue.find_one({})  # Assuming the queue
information is stored in a 'queue' collection

    if not queue_data:
        # If no queue data exists, initialize it with the first employee
        queue_data = {
            'current_employee_index': 0
        }
        mongo.db.queue.insert_one(queue_data)
    # Get the current employee index
    current_employee_index = queue_data['current_employee_index']

    # Find the next employee in the queue
    next_employee_index = (current_employee_index + 1) % len(employees)
# Round-robin logic

    # Assign the next employee
    assigned_employee = employees[next_employee_index]

    # Update the queue pointer to the new employee
    mongo.db.queue.update_one({}, {'$set': {'current_employee_index':
next_employee_index}})
    # Create an assignment record in the 'assignments' collection
    assignment = {
        'order_id': ObjectId(order_id),
        'employee_id': assigned_employee['_id'],
        'assigned_date': datetime.now(),
        'status': 'Assigned'
    }
    mongo.db.assigned_tasks.insert_one(assignment)
    return assigned_employee
```

## 5.3) Screenshots of the Application/System

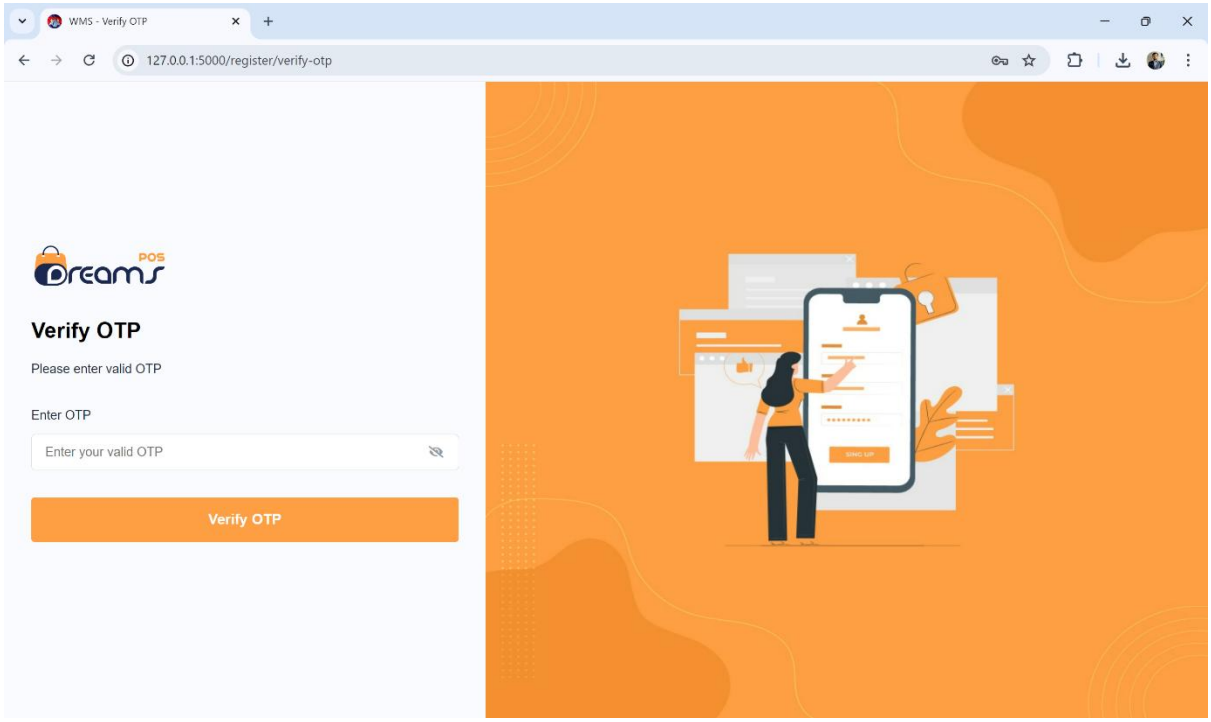
### 1) User:



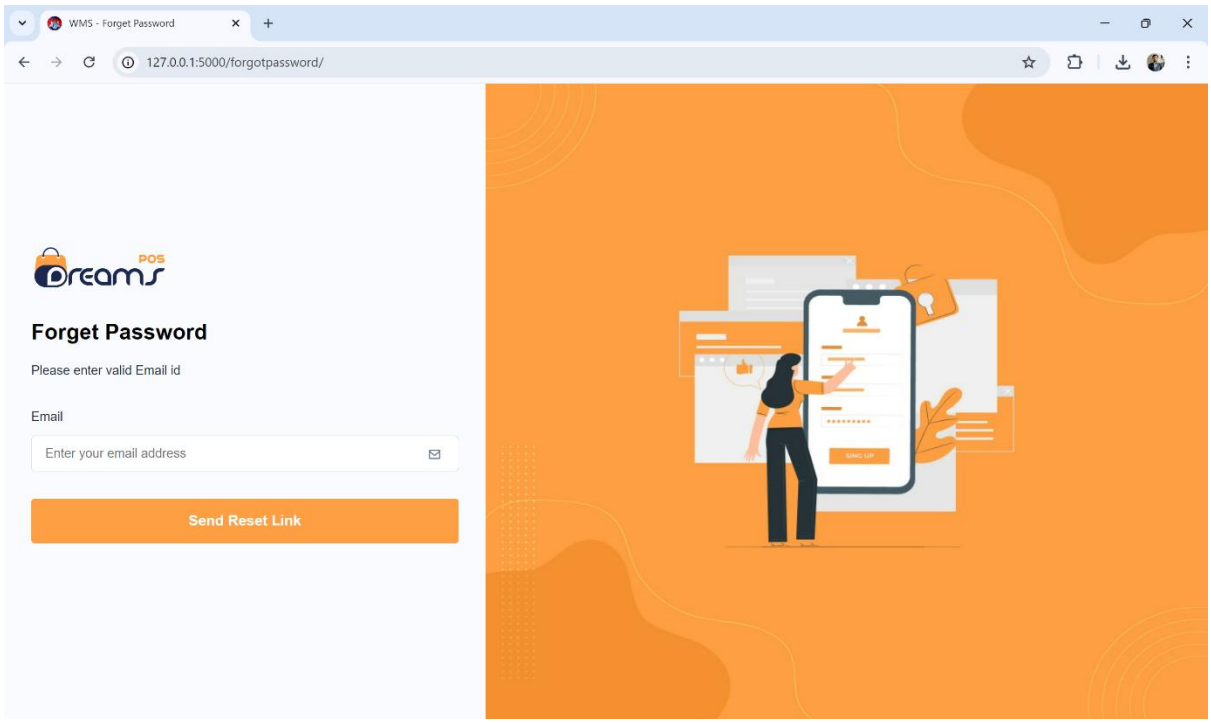

Sign In



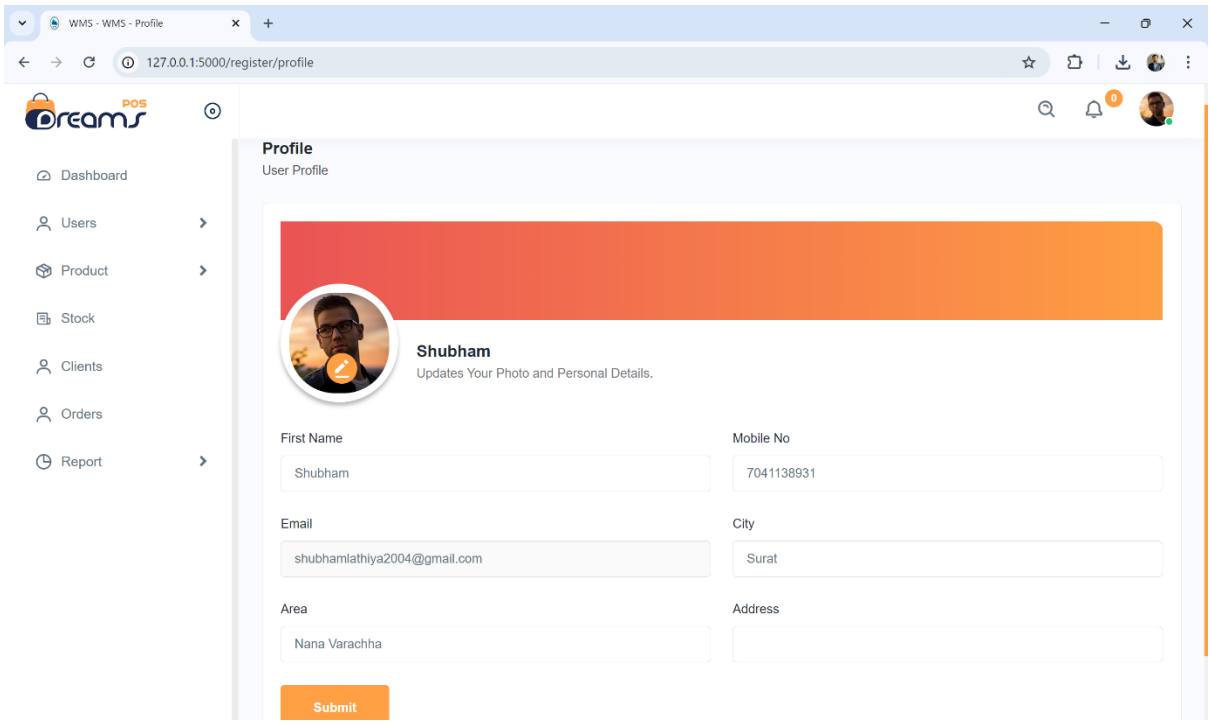

Sign Up




OTP Verify

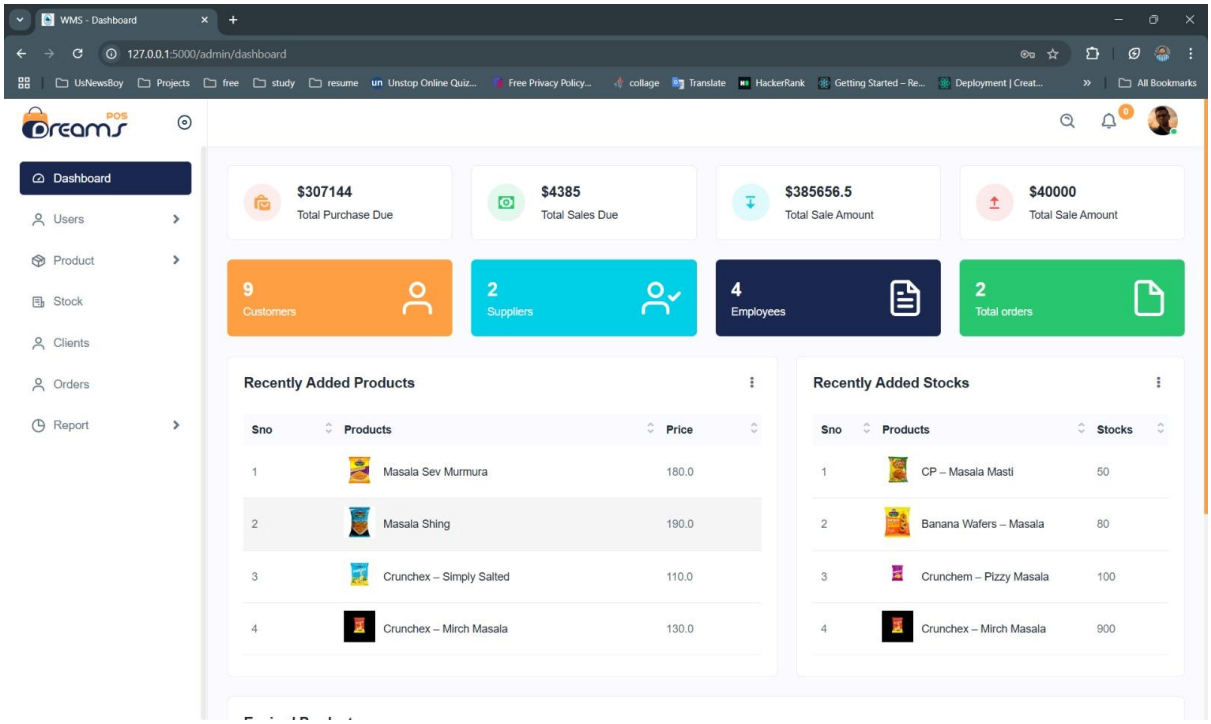



Forget Password

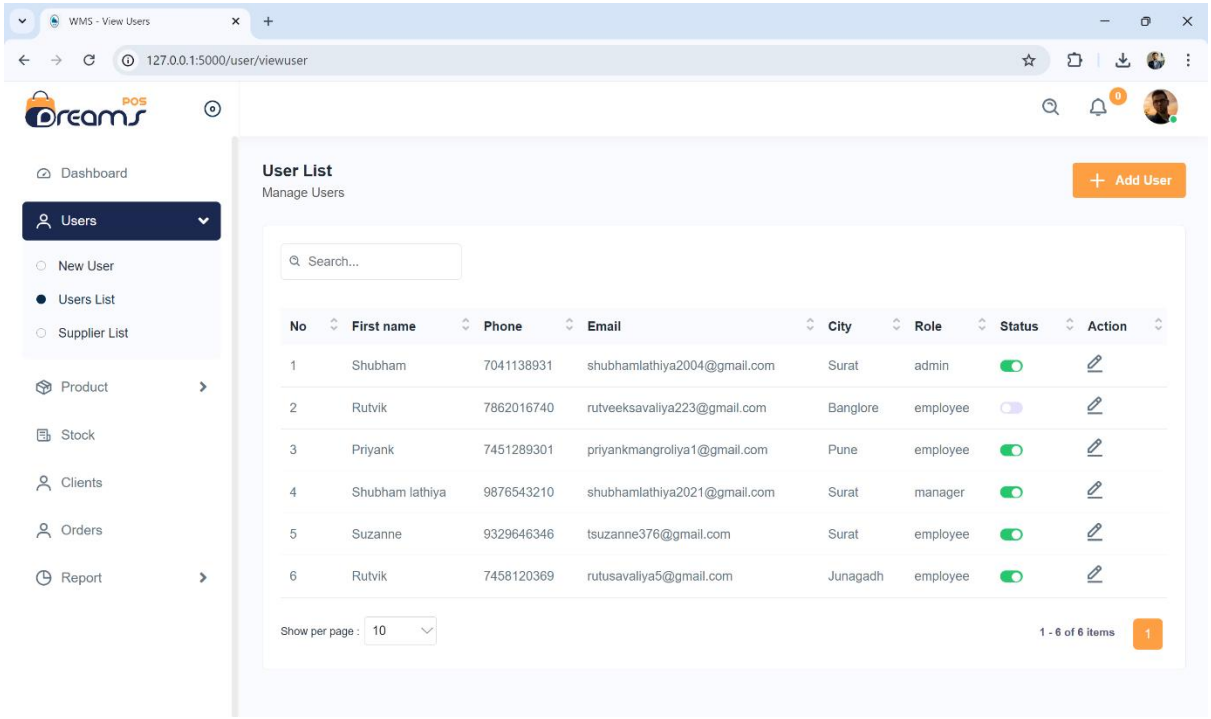



View Profile

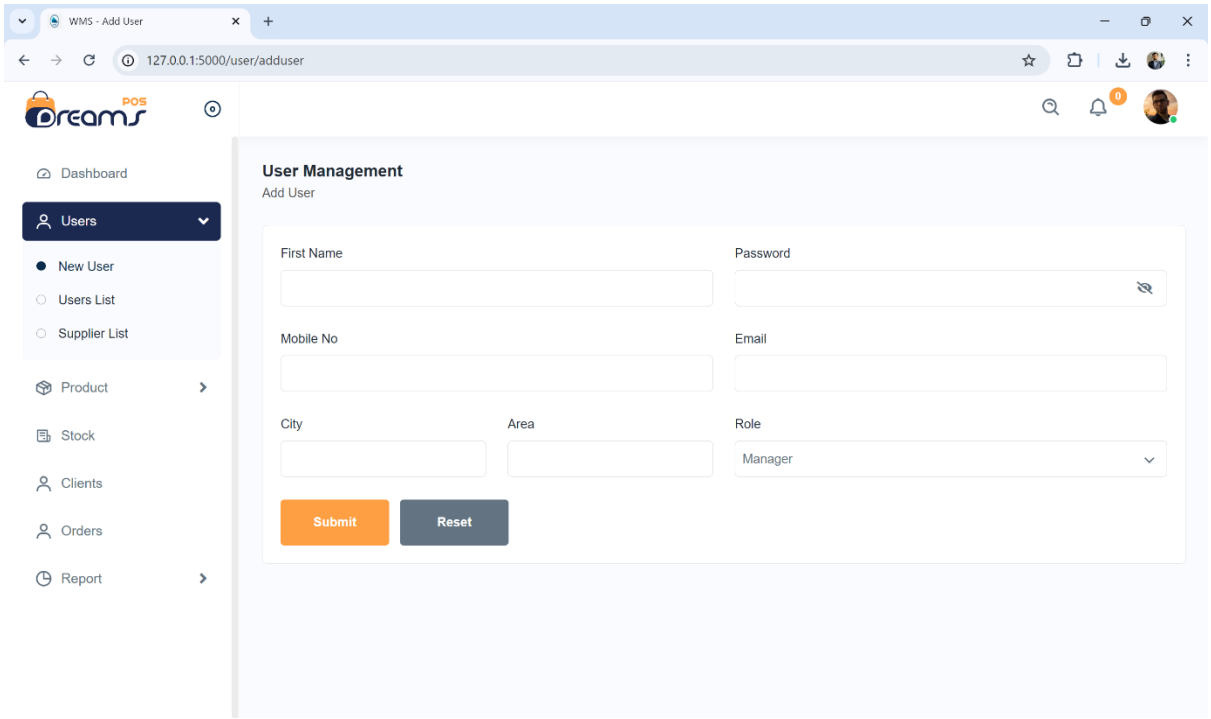
## 2) Admin and Manager:



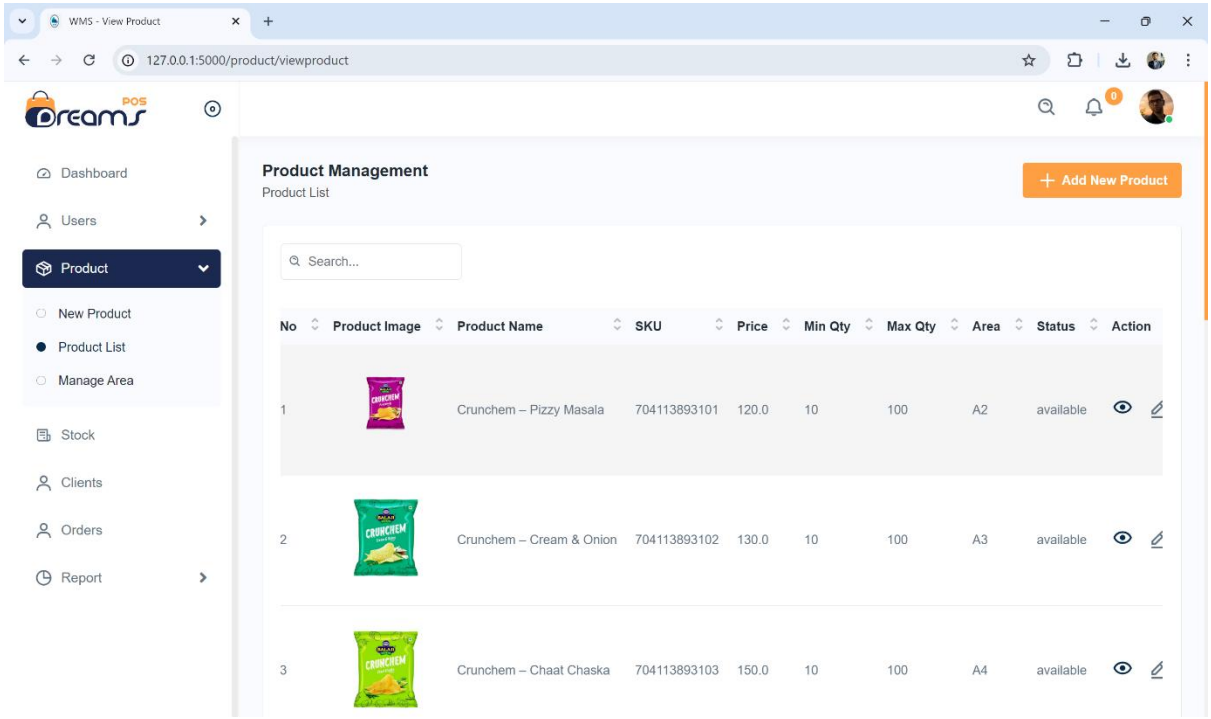

Admin Dashboard



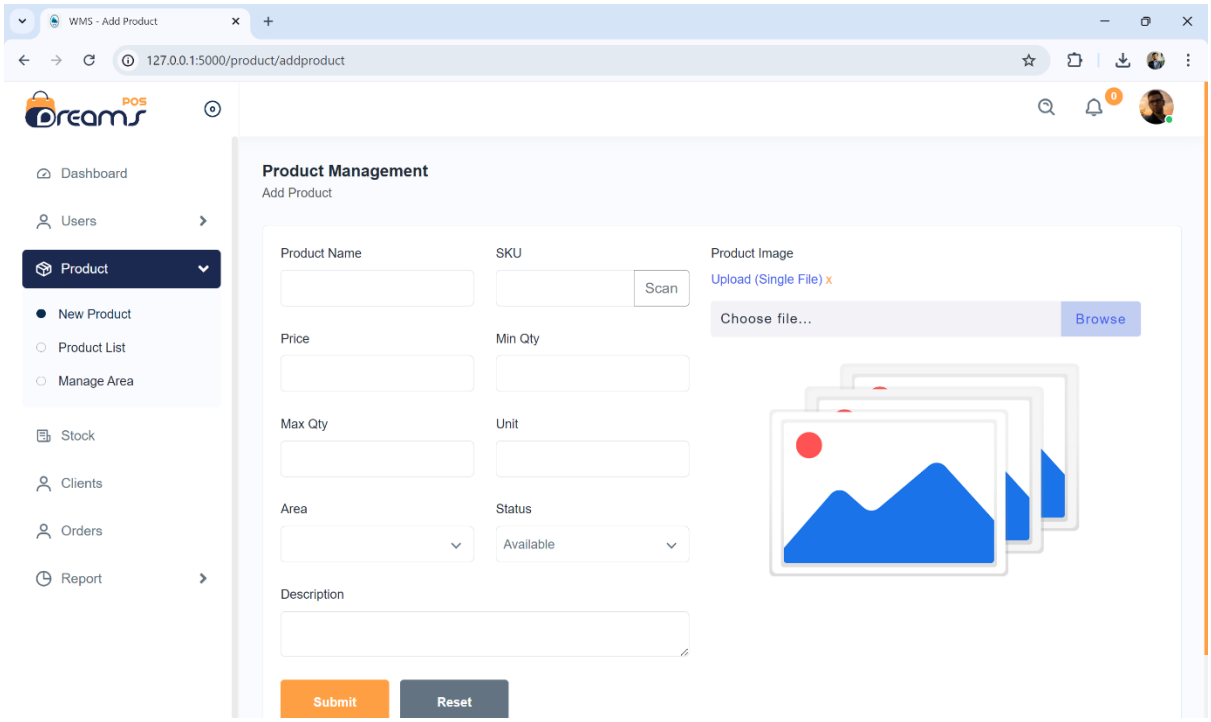

View Users



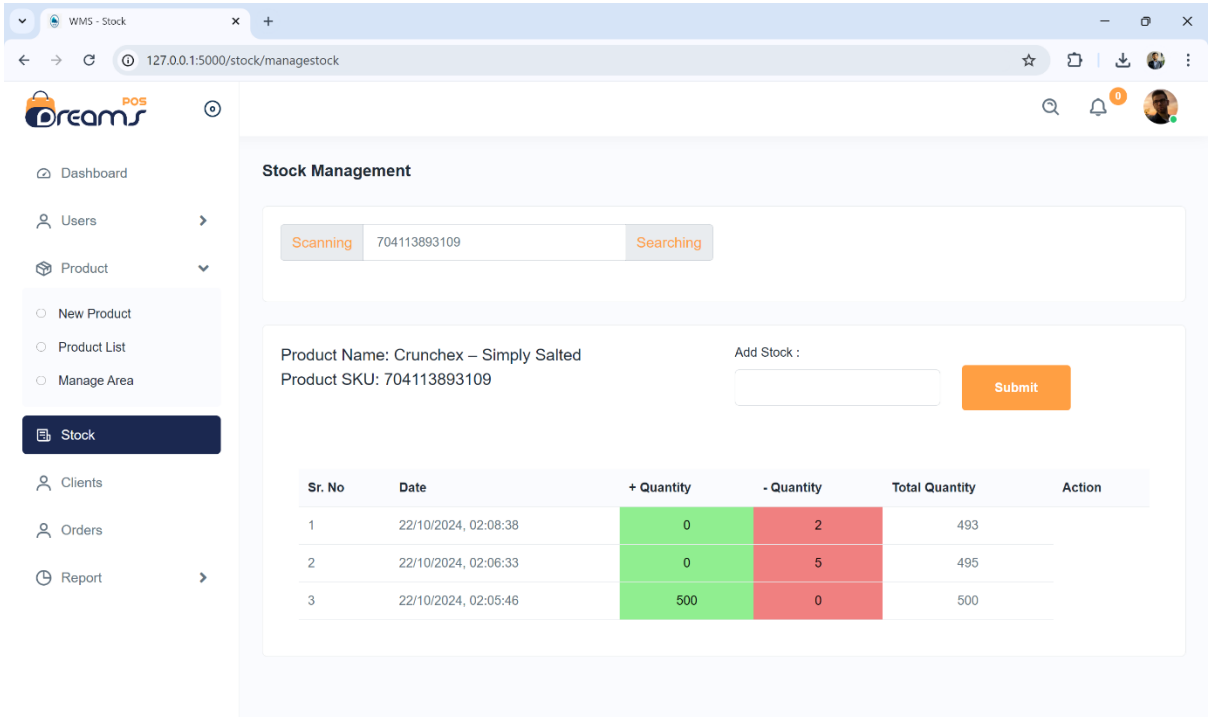

Manage User



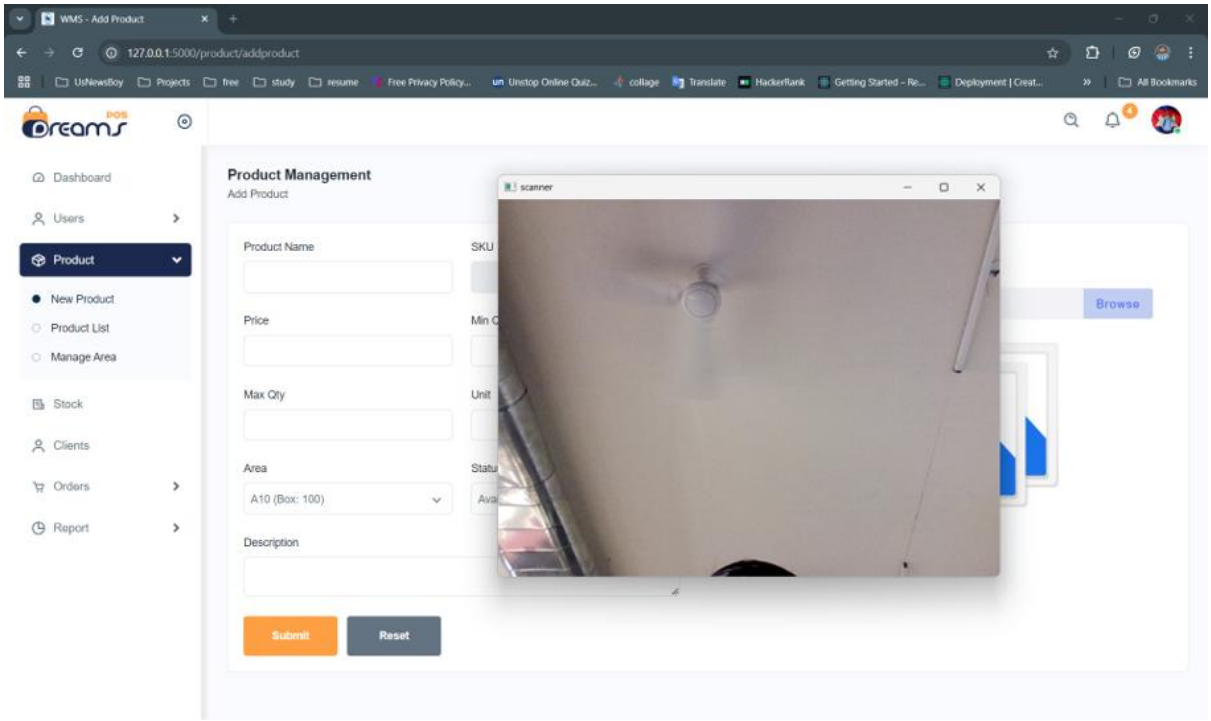

View Products



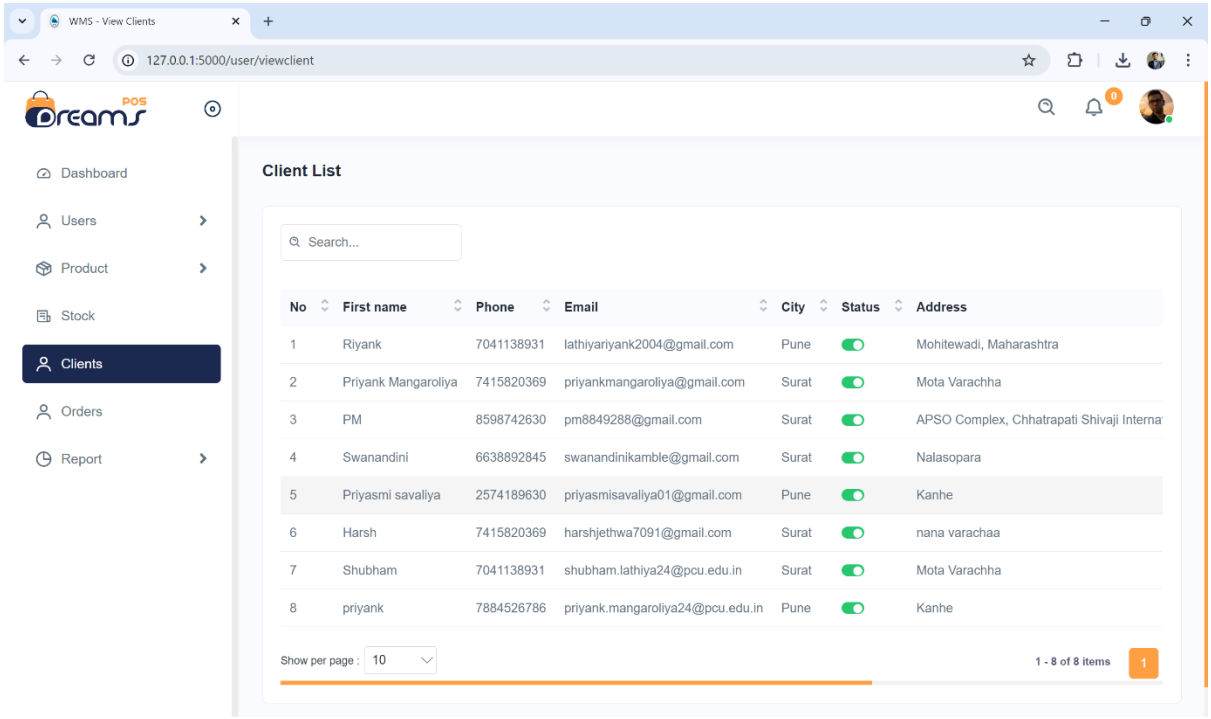

Manage Product



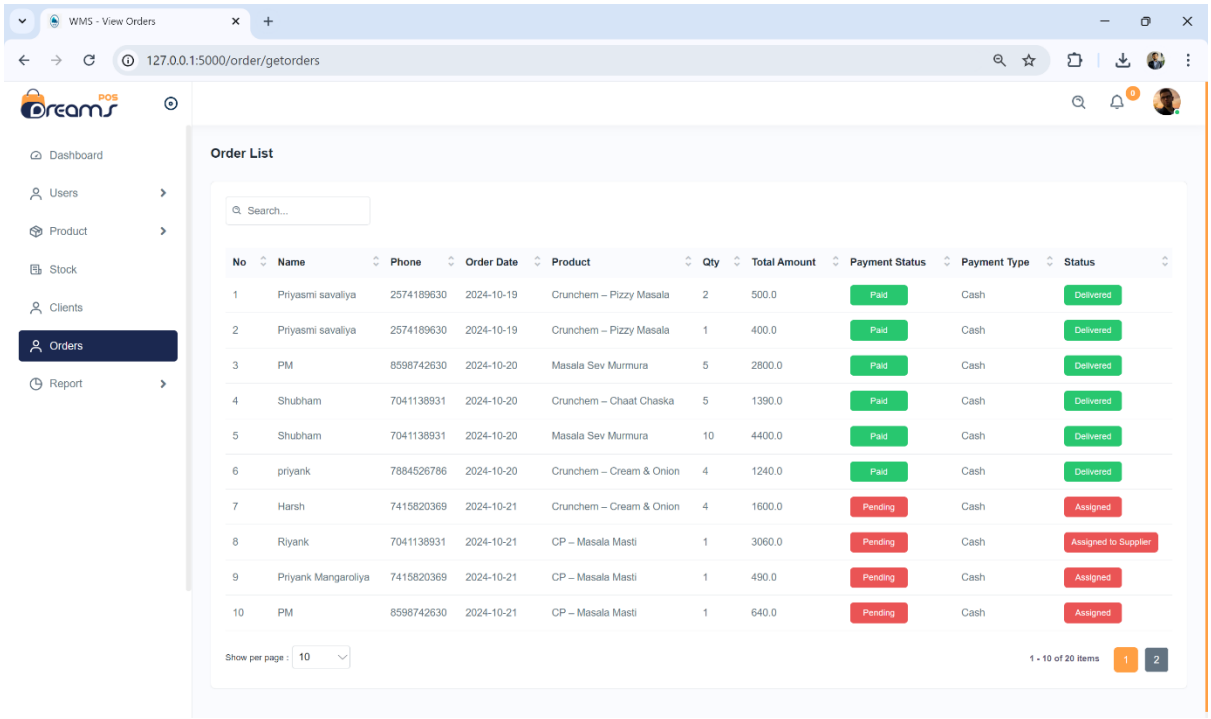

Manage Stock



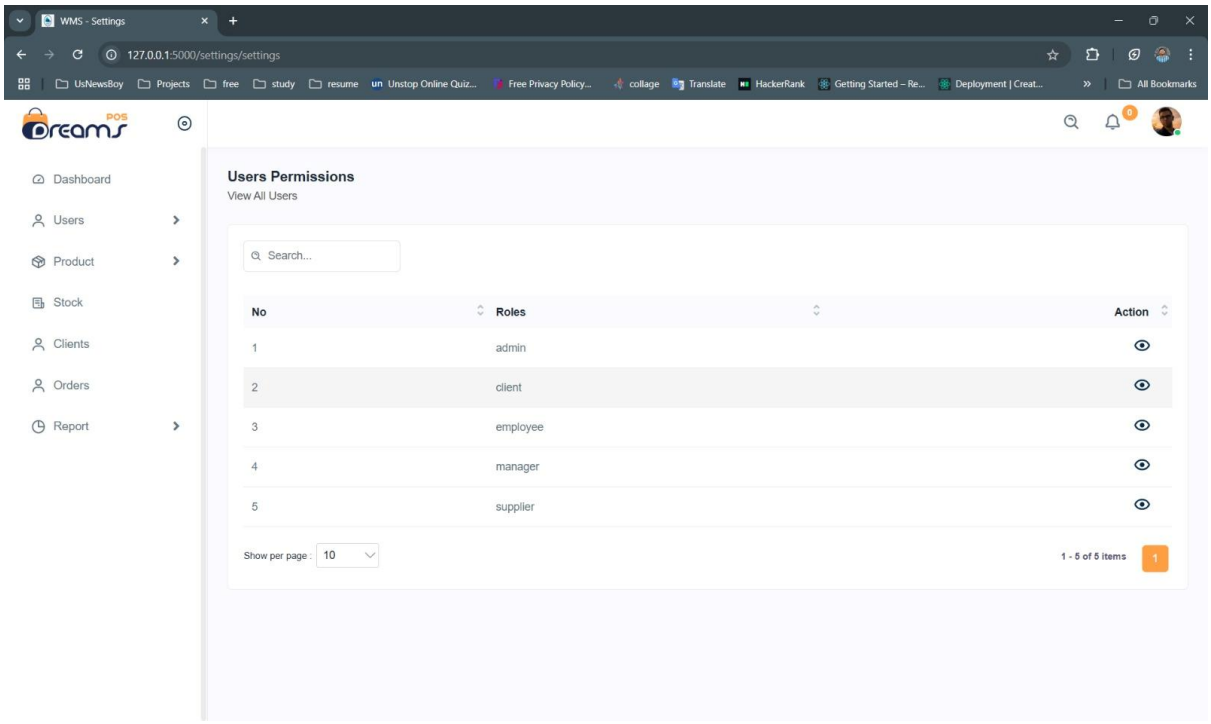

Scan QR Code



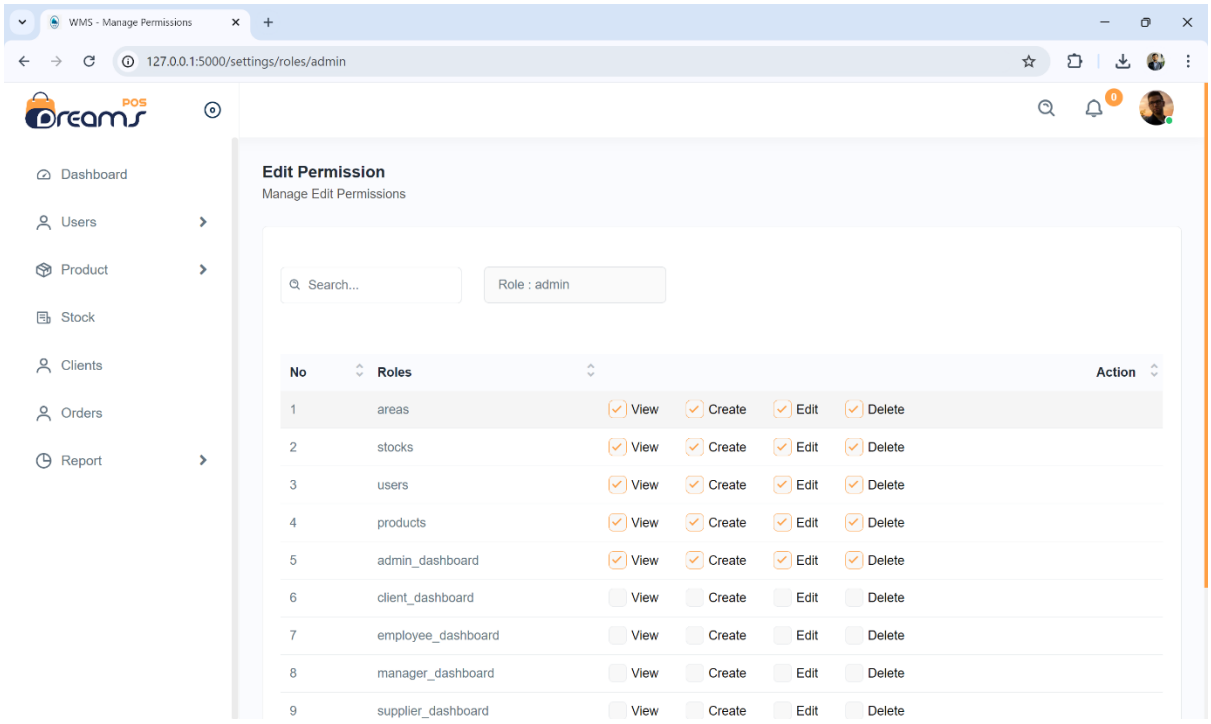

View Clients




View Orders

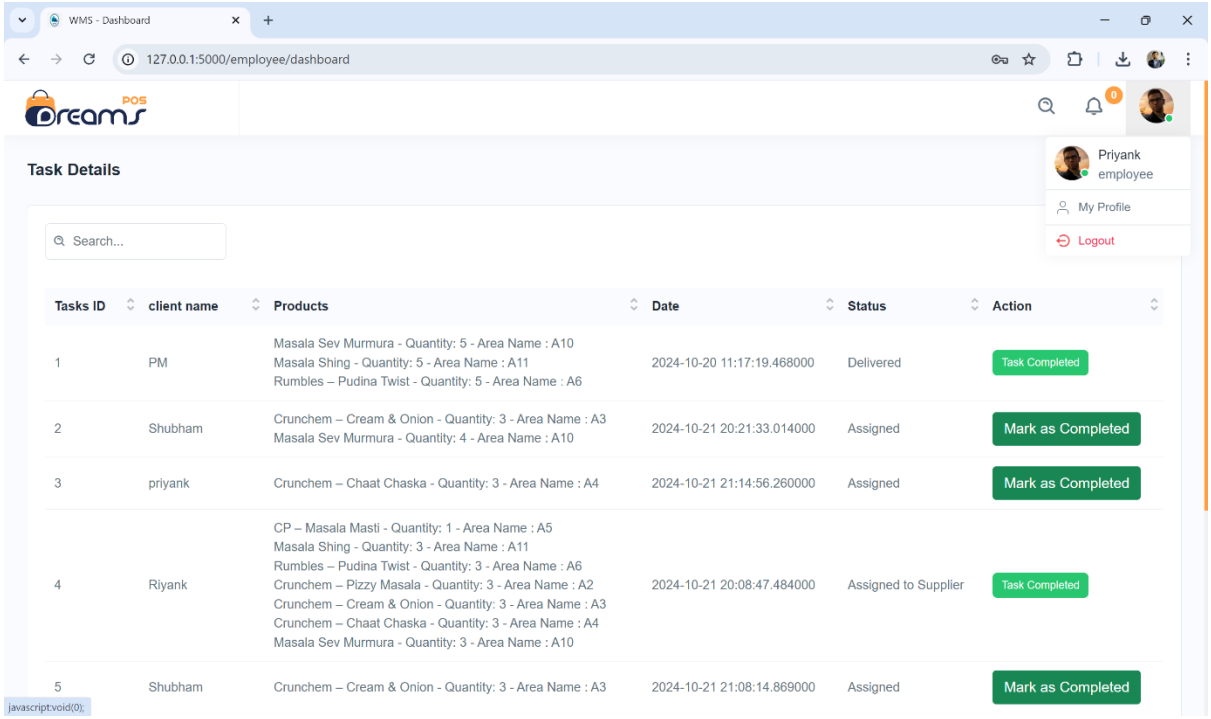



Manage Role Permission



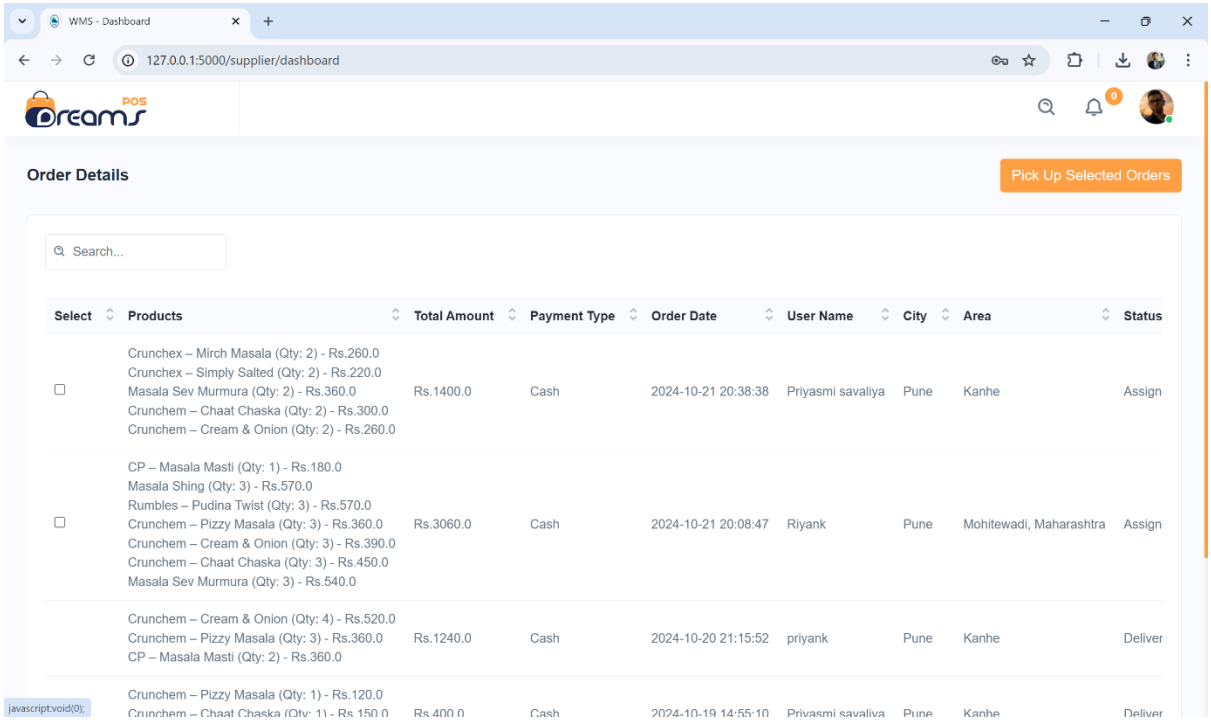

Manage Permission

# 3) Employee:

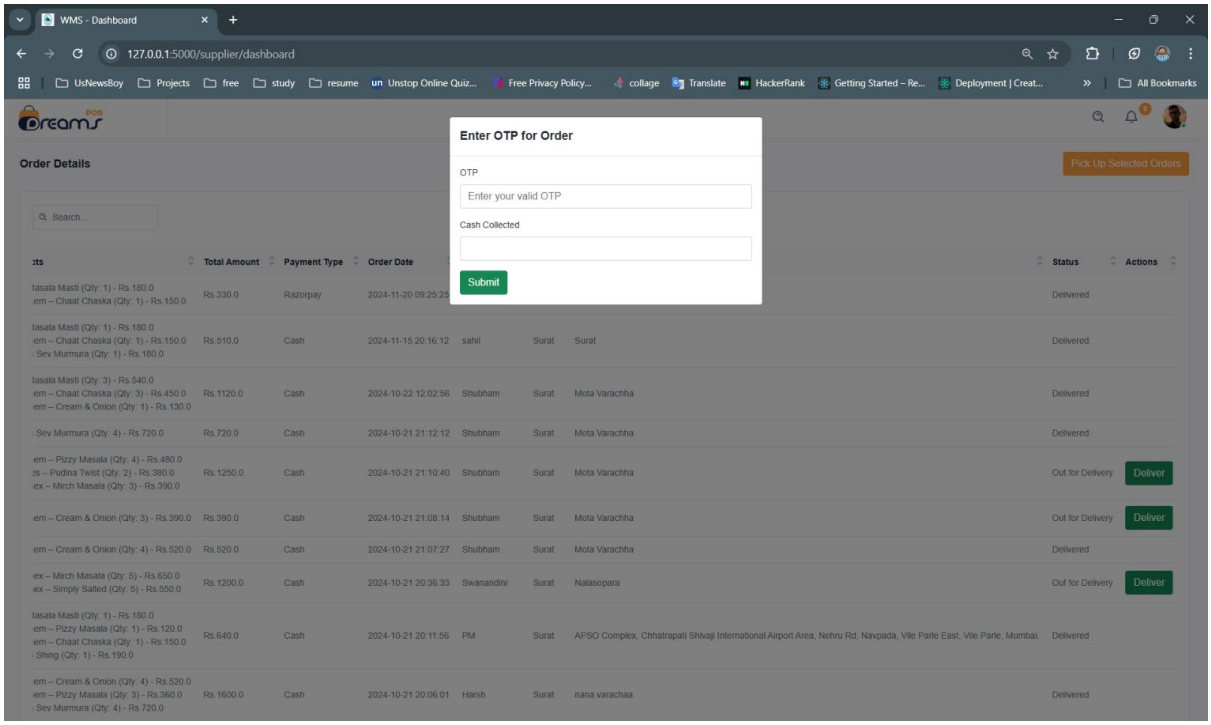



Employee Dashboard

# 4) Supplier:

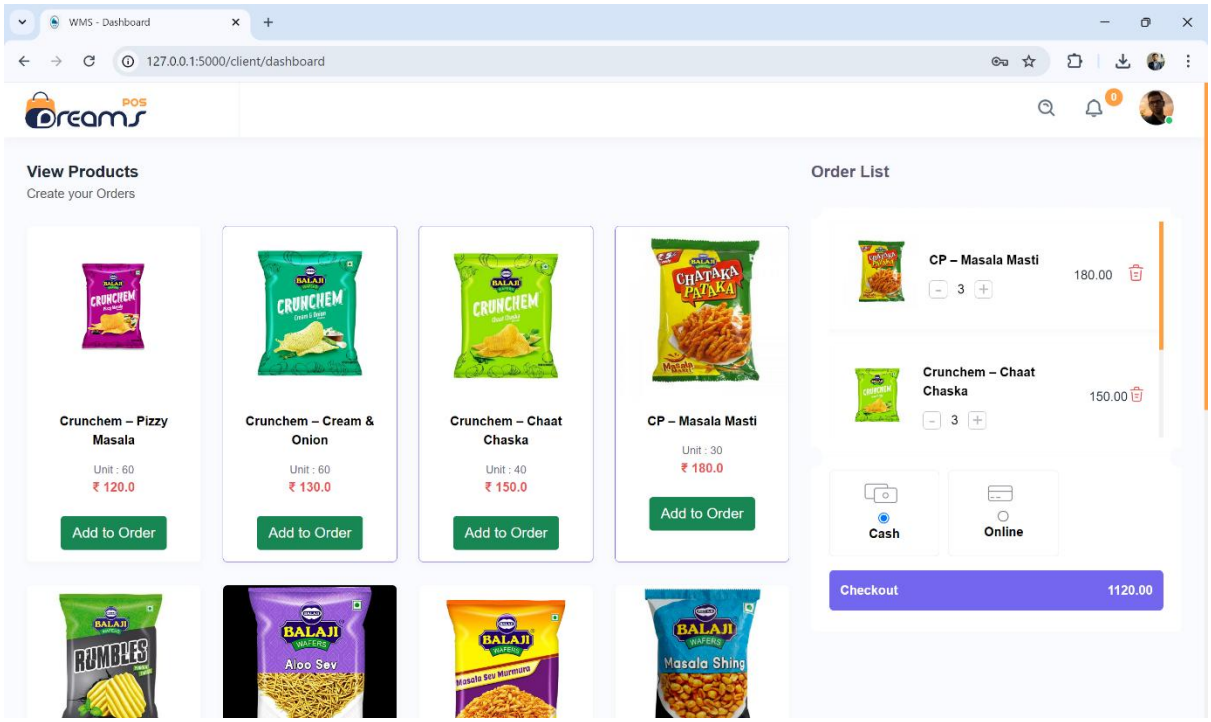



Supplier Dashboard

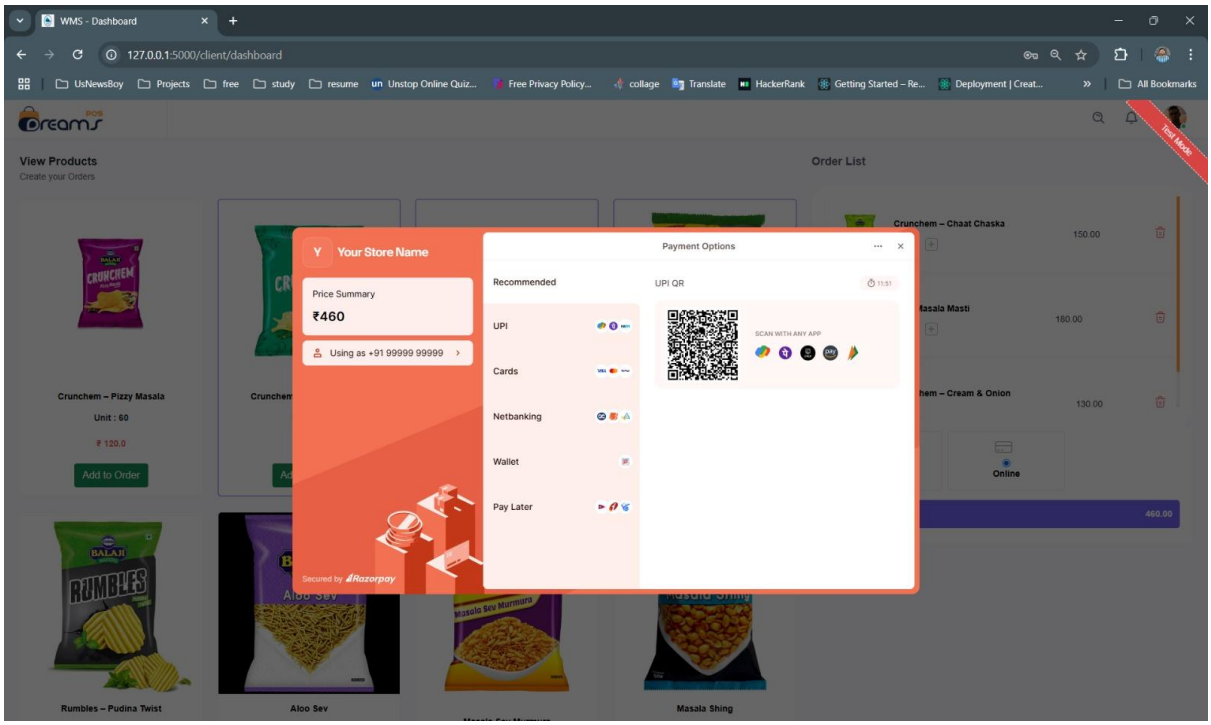



Verify OTP

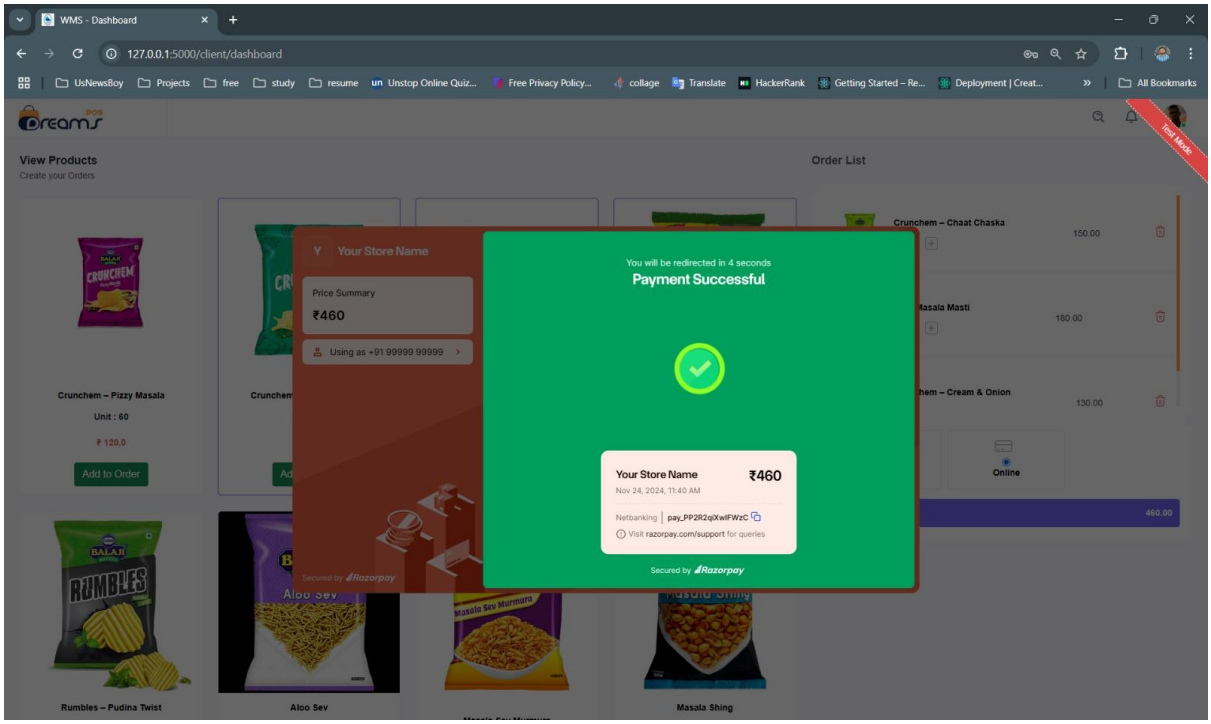
## 5) Client:



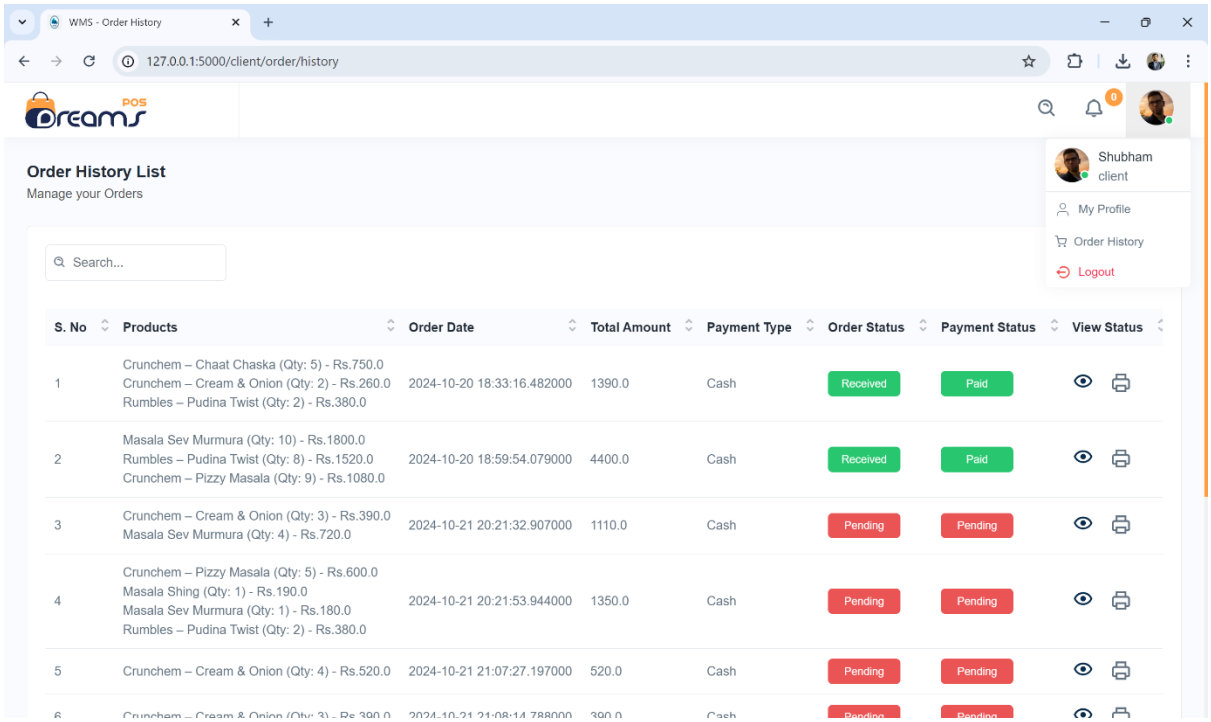

Client Dashboard



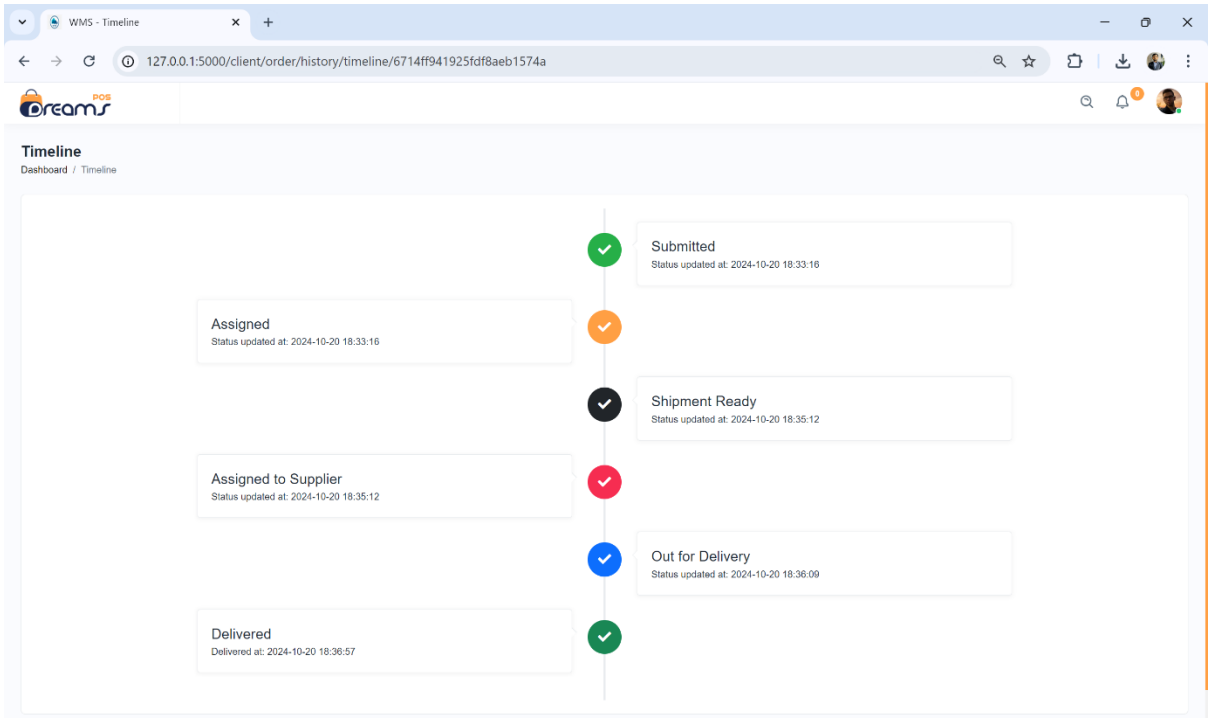

Payment



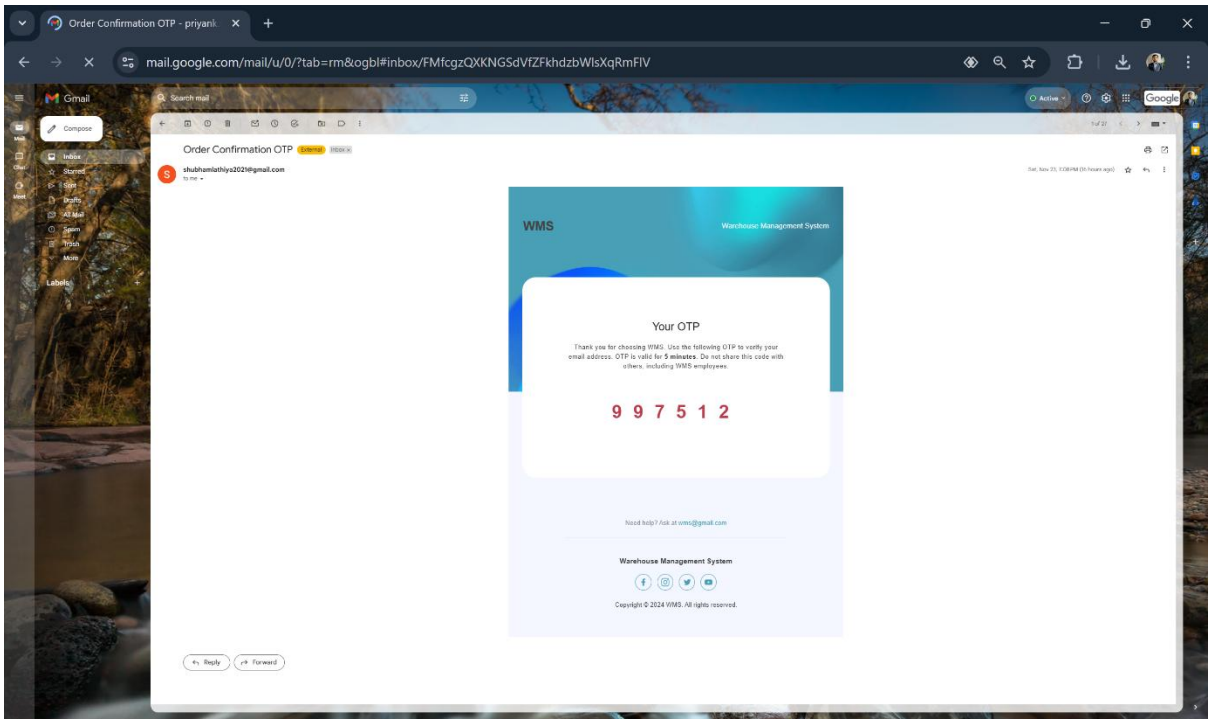

Payment Success



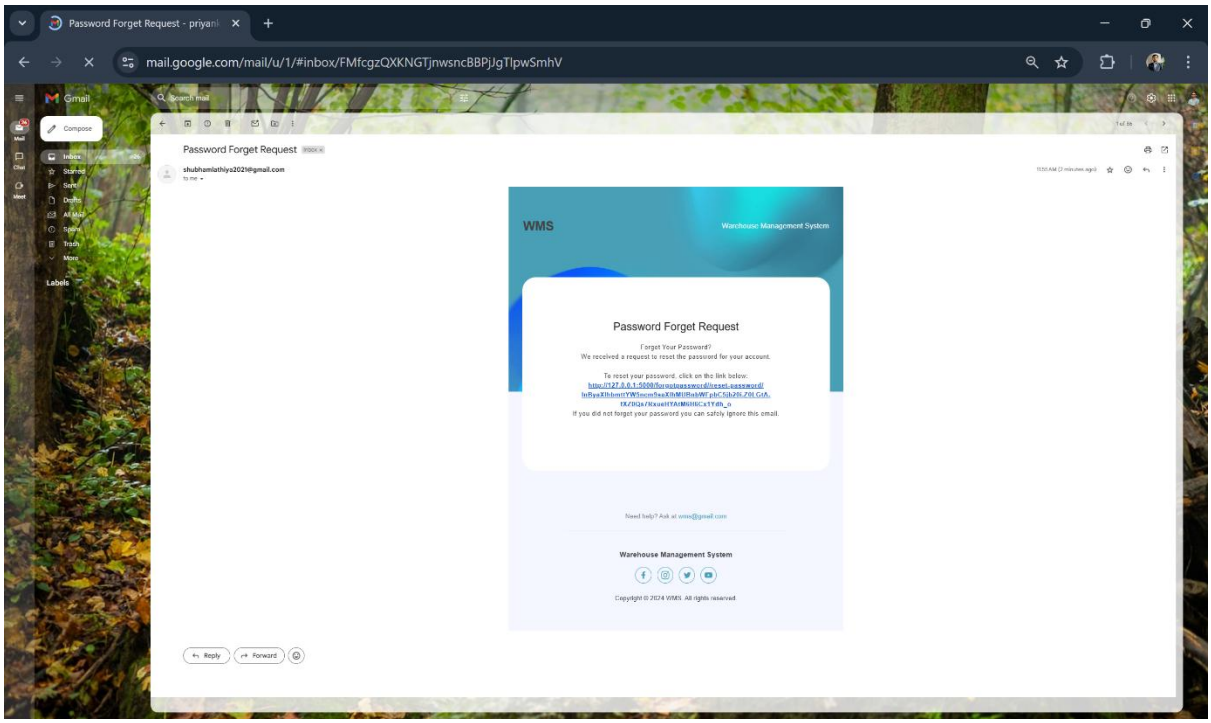

View Order History




View Timeline




OTP Mail




Reset Password Mail

# 6) Testing

## 6.1) Test Cases and Testing Methodology

1) **Unit Testing:** Each module, such as inventory management, order processing, and user authentication, is tested independently to ensure they function correctly.

2) **Integration Testing:** Focuses on the interaction between modules like order placement by clients, updates by suppliers, and dashboard updates for managers.

3) **Functional Testing:** Ensures that the system meets all functional requirements, including user role-specific functionalities.

4) **Performance Testing:** Tests the system's responsiveness, speed, and scalability under varying loads, including high data volumes and simultaneous user interactions.

5) **Security Testing:** Verifies encryption, role-based access control, and protection against vulnerabilities like data breaches.

6) **Usability Testing:** Assesses the system's user interface and ease of use for all roles, ensuring it is intuitive and accessible.

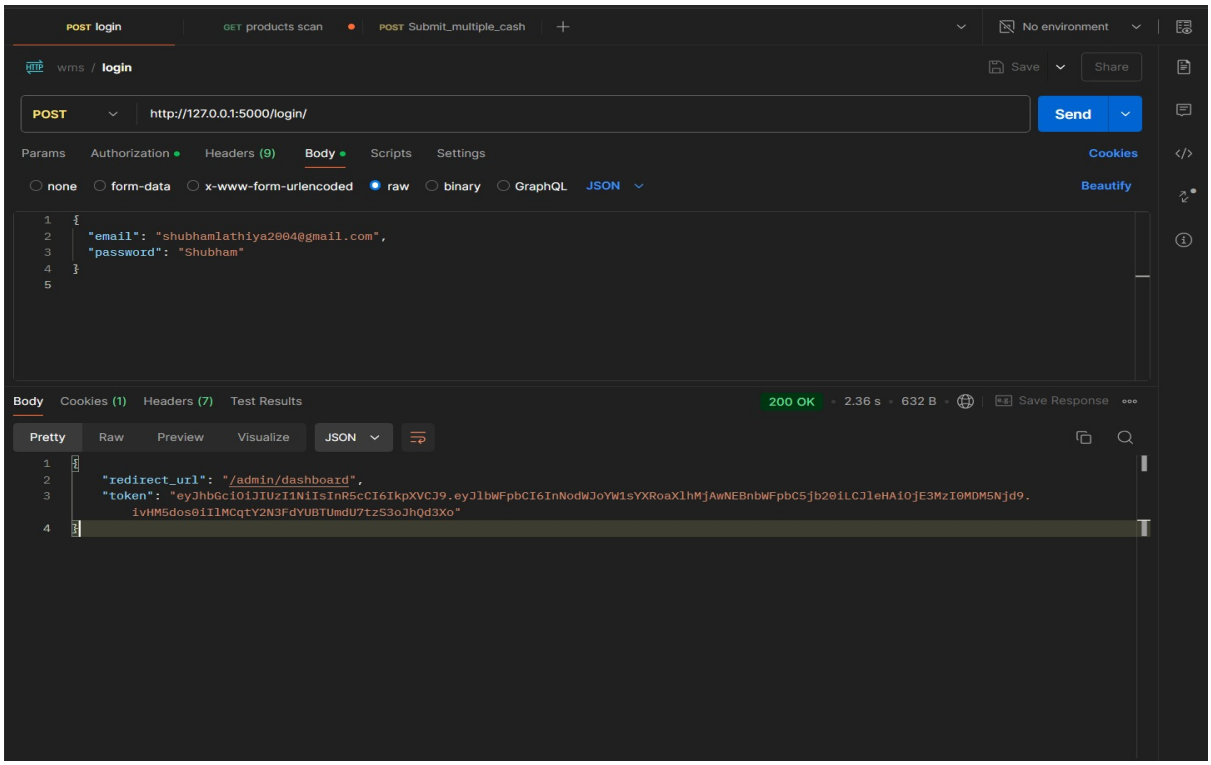
7) **Regression Testing:** Re-tests existing functionality after updates or bug fixes to ensure no new issues are introduced.

8) **Acceptance Testing:** Final validation by end-users to ensure the system meets business needs and is ready for deployment.

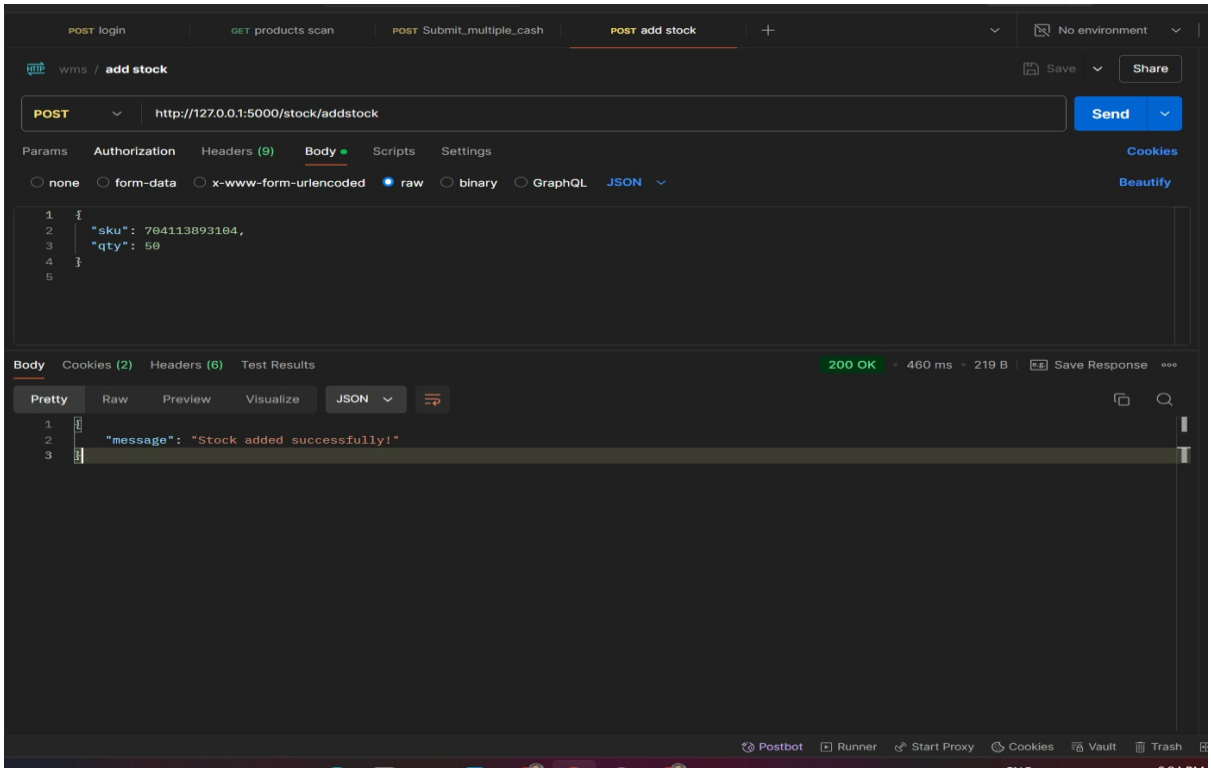
## 6.2) Results of Testing

### 1) Automatic Testing:

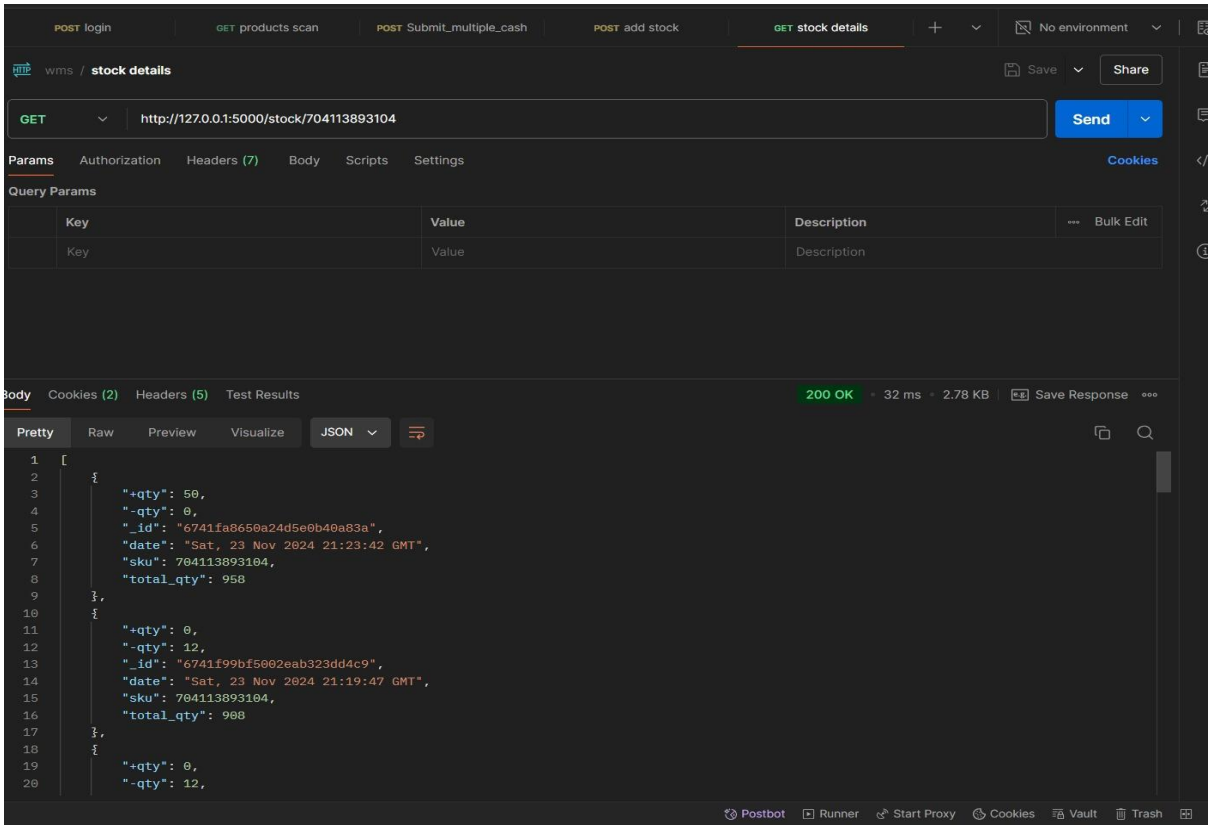
Login:



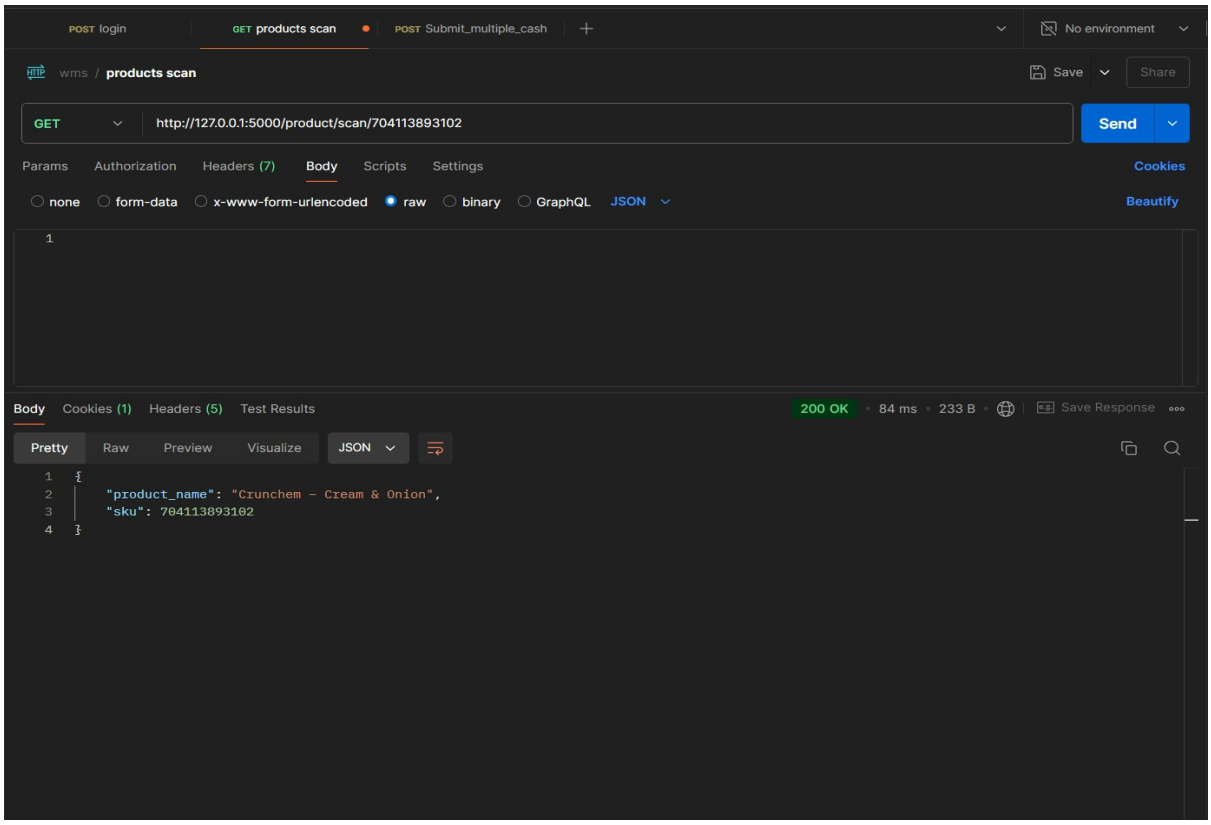

Add Stock:



## Stock View:




## Scan QR:



**2) Manual Testing:**

| Test Case ID | Test Case Name | Input Data | Expected Output | Actual Output | Test Status |
|---|---|---|---|---|---|
| TC_01 | Retrieve stock by valid SKU | SKU: 1001 (valid SKU) | Stock details for SKU: 1001 with fields total_qty, +qty, -qty, and date. Example: { "sku": 1001, "total_qty": 100, "+qty": 50, "-qty": 0, "date": "2023-11-23" } | Stock details for SKU: 1001 returned as expected. | Pass |
| TC_02 | Retrieve stock for non-existent SKU | SKU: 9999 (non-existent SKU) | Error message indicating "No stock found" for SKU 9999. Example: { "error": "No stock entries found" }. | Error message No stock entries found returned correctly. | Pass |
| TC_03 | Test unauthorized access (no JWT token) | SKU: 1001, No JWT token | Error message indicating unauthorized access. Example: { "error": "Authentication required" }. | Error message Authentication required returned. | Pass |
| TC_04 | Retrieve stock for SKU with multiple stock entries | SKU: 1001 (Multiple entries) | Multiple stock entries for SKU 1001, each showing total_qty, +qty, -qty, and date. Example: [ { | Multiple stock entries returned as expected. | Pass |

| | | | | | |
|---|---|---|---|---|---|
| | | | "sku": 1001, "total_qty": 100, "+qty": 50, "-qty": 0, "date": "2023-11-23" }, { "sku": 1001, "total_qty": 200, "+qty": 100, "-qty": 0, "date": "2023-11-24" } ] | | |
| TC_05 | Test stock retrieval with invalid SKU format | SKU: 'ABC' (invalid format) | Error message indicating invalid SKU format. Example: { "error": "Invalid SKU format" }. | Error message Invalid SKU format returned correctly. | Pass |
| TC_06 | Test stock retrieval with missing SKU parameter | SKU: (empty) | Error message indicating SKU is required. Example: { "error": "SKU is required" }. | Error message SKU is required returned correctly. | Pass |

## 6.3) Bug Fixes/Issues Resolved

### 1) User Interface:

**Issue:** Some UI elements were not responsive on smaller screen sizes.
**Fix:** Updated CSS and JavaScript for better responsiveness and compatibility.

### 2) Order Processing

**Issue:** Order status was not synchronizing correctly between the dashboard and client view.
**Fix:** Optimized database queries and improved data synchronization logic.

# 7) Results and Discussion

## 7.1) Outcomes of the Project

- **Admin:** Manage users, settings, integrations, and the whole system with real-time visibility, improved decision-making, and enhanced security.

- **Manager:** The Warehouse Manager ensures efficient warehouse performance by monitoring inventory, order processing, and workflows with real-time data access.

- **Employee:** Automates daily tasks like inventory updates, picking, and packing. Makes their tasks easier and more efficient by scanning barcodes and generating picking lists.

- **Client:** The system allows clients to place orders and track them in real time. They get order tracking at every step, order confirmation, picking, packing, and shipping.

- **Supplier:** The system enables suppliers to manage their orders efficiently by providing real-time order status updates, delivery tracking, and payment processing.

## 7.2) Comparison with existing solutions

- The proposed Warehouse Management System (WMS) offers several advantages over existing solutions. Unlike legacy systems, it provides real-time inventory tracking, scalability, and user-friendly interfaces. Cloud-based WMS solutions are more flexible but often come with high costs and security concerns, which this system addresses through cost-effective, open-source technologies and robust security features like AES-256 encryption.

- ERP-integrated WMS solutions offer unified operations but may be complex to implement, whereas our system focuses on simplicity and ease of use for quick adoption. Additionally, this WMS is more customizable and adaptable compared to many industry-specific solutions, offering a flexible platform without high implementation costs.

## 7.3) Challenges Faced

- During the development of the Warehouse Management System (WMS), several challenges were encountered. One major challenge was ensuring seamless real-time inventory tracking and accurate order processing, which required robust backend architecture and efficient database management. Integrating security features like role-based access control posed additional complexity to ensure data protection and prevent unauthorized access.

- The development also faced challenges in creating an intuitive user interface that could be easily adopted by warehouse staff with minimal training. Furthermore, scalability and ensuring the system could handle increased data loads as the warehouse grows required careful planning. Addressing these challenges required a balance between performance, security, and user experience to create a system that is both functional and efficient.

# 8) Conclusion and Future Scope

## 8.1) Summary of the work

- This project focuses on designing and developing a Warehouse Management System (WMS) to streamline warehouse operations by addressing common challenges like inaccurate inventory tracking, inefficient space utilization, and slow order processing.

- The WMS employs an Agile development methodology and a technology stack comprising Python (Flask) for backend, MongoDB for scalable storage, and a responsive frontend built with HTML, CSS, and JavaScript. Core functionalities include real-time inventory tracking, automated order management, role-based user access, and analytics reporting. Key user roles—admin, manager, employee, supplier, and client—have distinct functionalities, such as managing stock, processing orders, tracking deliveries, and viewing reports. Non-functional requirements like security, email notifications, and 99.9% availability are prioritized.

## 8.2) Limitations

- **No Multi-Warehouse Support:** Features like multi-warehouse management are excluded.

- **No Returns Processing:** The system does not include return or replacement functionality.

- **Limited Integrations:** Advanced integrations with external systems, such as ERP platforms, are not supported.

## 8.3) Future Enhancements

- **Multi-Warehouse Management:** Extend functionality to support multiple warehouses.

- **Returns Processing:** Add features for handling product returns and replacements.

- **Advanced Integrations:** Enable integration with ERP systems and third-party logistics providers.
- **AI and IoT Integration:** Incorporate AI for demand forecasting and IoT for real-time inventory tracking.
- **Mobile App:** Develop a mobile application for on-the-go access and warehouse task management.
- **Customizable Dashboards:** Allow users to personalize dashboard views and reports.

# References

1) https://ieeexplore.ieee.org/document/278253
2) https://dspmuranchi.ac.in/pdf/Blog/srs_template-ieee.pdf
3) https://www.zoho.com/in/inventory/
4) https://www.kladana.com/product-tour/inventory-management/