mirafra
TECHNOLOGIES

CAPSTONE PROJECT ON
Advanced E-commerce Recommendation System

ABSTRACT

The Advanced E-commerce Recommendation System is a web-based platform designed to enhance the online shopping experience by allowing users to seamlessly switch between buyer and seller roles. Unlike traditional e-commerce systems that require separate accounts for buying and selling, this platform offers a unified login for both functionalities. Built using Flask for the backend and MySQL for database management, the system provides features such as user authentication, product management, a shopping cart, and order tracking. The system also integrates AI-driven recommendations to improve user engagement by suggesting personalized products based on browsing history and past purchases. This report outlines the development, implementation, and potential future enhancements of this e-commerce platform.

INTRODUCTION

E-commerce has revolutionized the way people buy and sell products, offering unparalleled convenience and accessibility. However, most traditional e-commerce platforms require users to create separate accounts for buying and selling, leading to inefficiencies and inconvenience. Our project addresses this issue by developing an integrated system where a single user account can act as both a buyer and a seller. This allows users to switch roles effortlessly, enhancing user experience and operational efficiency. The system supports product listing, cart management, order tracking, and an AI-powered recommendation engine to personalize the shopping experience. By leveraging Flask for backend processing and MySQL for database management, this platform provides a scalable and efficient solution for modern e-commerce needs.

OBJECTIVES

- **User Authentication -** Implement a secure authentication system allowing users to register and log in with a single account for both buying and selling.
- **Role Switching -** Develop a seamless "Switch User" feature to toggle between buyer and seller functionalities without re-logging.
- **Product Management -** Enable sellers to add, update, and delete their products, as well as track orders from buyers.
- **Buyer Experience -** Provide buyers with the ability to browse products, manage shopping carts, and place orders efficiently.
- **Order Management -** Allow buyers to track orders and sellers to manage incoming purchase requests.
- **Review Analysis** - Clicking on a product image suggests “Good” or “Not Recommended” based on past reviews.

FEATURES

- User Authentication - Secure login and registration with encrypted password storage.
- Dual-Role System - A single user can act as both a buyer and a seller.
- Product Management - Sellers can list, update, and remove products.
- Shopping Cart - Buyers can add products to their cart before purchasing.
- Order Management - Buyers can track their orders, and sellers can process received orders.
- Database Integration - MySQL for efficient data storage and retrieval.
- Product Recommendations – Personalized suggestions based on post reviews.

FLOWCHART

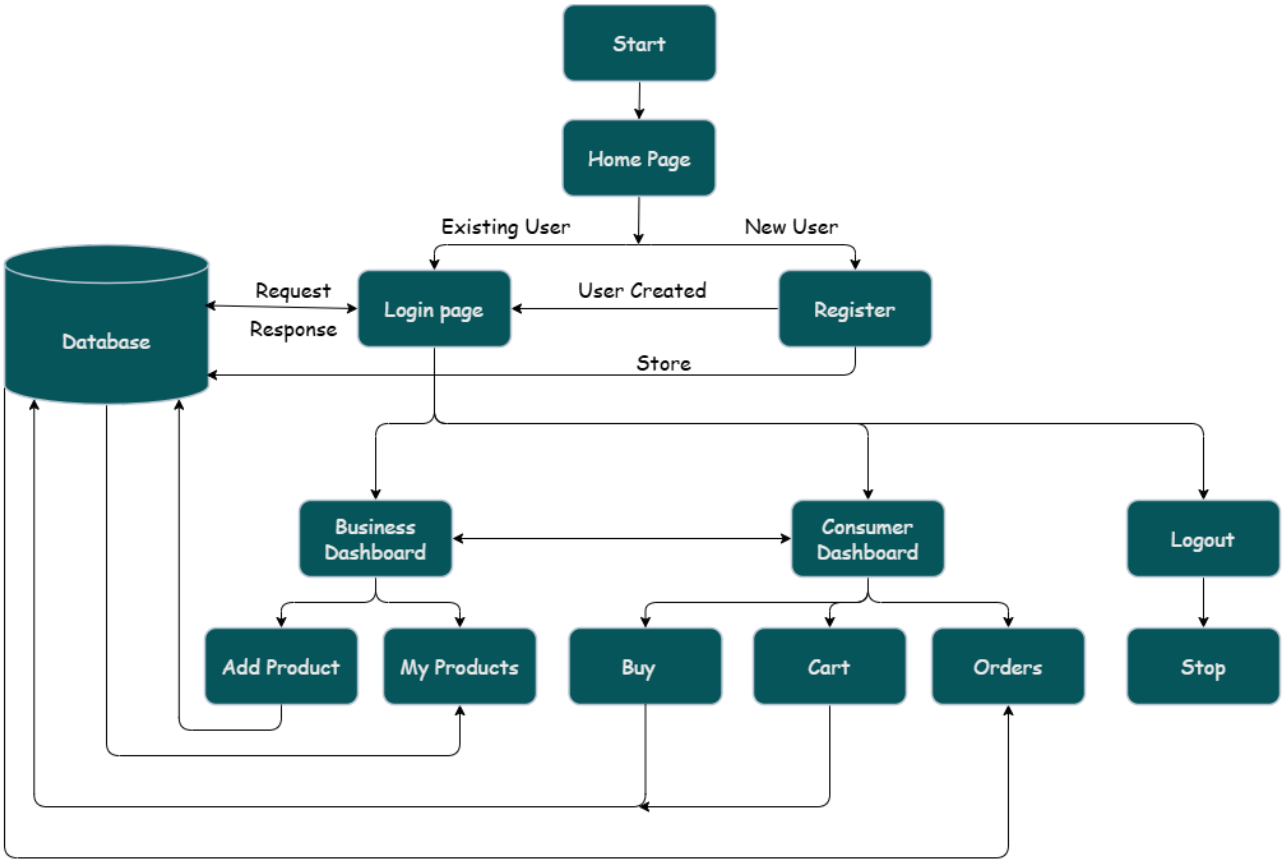

Fig 1: Flowchart

ER DIAGRAM

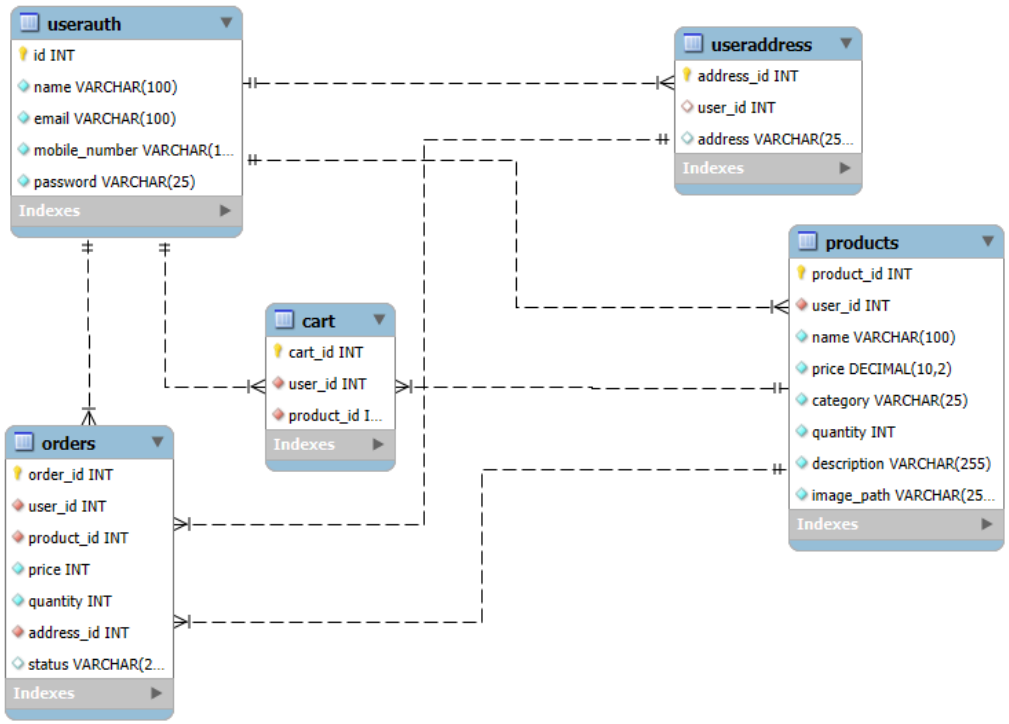

Fig 2: ER Diagram

CODE IMPLEMENTATION

```
import os
from flask import Flask, render_template, request, redirect, flash, url_for, jsonify
from flask_mysqldb import MySQL
from flask_login import LoginManager, UserMixin, login_user, logout_user,
login_required, current_user
import csv
import json
from database import UserAuth, Address, Product, Cart, Order  # Import database models

# Initialize Flask app
app = Flask(__name__)

# Secret key for session management (should be kept secure)
app.secret_key = 'your_secret_key'

# Define upload folder path
UPLOAD_FOLDER = 'static/uploads/'
if not os.path.exists(UPLOAD_FOLDER):
    os.makedirs(UPLOAD_FOLDER)  # Ensure the folder exists before using it
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER

# MySQL Configuration
app.config['MYSQL_HOST'] = 'localhost'  # Database host (localhost for local
development)
app.config['MYSQL_USER'] = 'root'  # MySQL username
app.config['MYSQL_PASSWORD'] = 'Anu@441461'  # MySQL password (should not be hardcoded
in production)
app.config['MYSQL_DB'] = 'anuroop'  # Database name

# Initialize MySQL with Flask app
mysql = MySQL(app)

# Initialize Flask-Login manager
login_manager = LoginManager()
login_manager.init_app(app)  # Attach login manager to the Flask app
login_manager.login_view = 'login'  # Set default login route

# Define User class for Flask-Login
class User(UserMixin):
    def __init__(self, id, name, email):
        self.id = id  # Unique user ID
        self.name = name  # User's name
        self.email = email  # User's email

# User loader function for Flask-Login
@login_manager.user_loader
def load_user(user_id):
```

```
    """
    Load user from the database by ID for session management.
    Returns a User object if found, otherwise None.
    """
    user_data = UserAuth.get_user_by_id(user_id)
    if user_data:
        return User(user_data['id'], user_data['name'], user_data['email'])
    return None

# Home route - Displays available products
@app.route('/')
def home():
    """
    Fetches all available products (where quantity > 0) from the database
    and renders them on the home page.
    """
    cur = mysql.connection.cursor()  # Create a cursor object
    cur.execute('SELECT * FROM products WHERE quantity > 0')  # Query products with
stock available
    products = cur.fetchall()  # Fetch all results
    return render_template('home.html', products=products)  # Render home page with
products data

# Home route - Renders the index page
@app.route('/index')
def index():
    """
    Renders the Home page.
    """
    return render_template('index.html')

# Route for user registration
@app.route('/register', methods=['POST', 'GET'])
def register():
    """
    Handles user registration.
    - GET request: Displays the registration form.
    - POST request: Processes the form data, checks if the user exists, and registers
them if not.
    """
    if request.method == 'POST':
        # Retrieve form data
        name = request.form['name']
        email = request.form['email']
        mobile_number = request.form['mobile_number']
        password = request.form['password']

        # Check if user already exists in the database
        if UserAuth.user_exists(email, mobile_number):
```

```
            flash("User with this email already exists. Please log in.", "warning")
            return redirect(url_for('register'))  # Stay on registration page if user
exists

        # Register new user in the database
        success = UserAuth.register_user(name, email, mobile_number, password)

        if success:
            flash("Registration successful! Please log in.", "success")
            return redirect(url_for('login'))  # Redirect to login page on successful
registration
        else:
            flash("Registration failed. Try again.", "danger")
            return redirect(url_for('register'))  # Stay on registration page if
registration fails

    return render_template('register.html')  # Render registration form for GET
requests

# Route for user login
@app.route('/login', methods=['POST', 'GET'])
def login():
    """
    Handles user login.
    - GET request: Displays the login form.
    - POST request: Authenticates the user, logs them in, and redirects based on
account type.
    """
    if request.method == 'POST':
        # Retrieve form data
        email = request.form['email']
        password = request.form['password']
        account_type = request.form['account_type']  # Determines if login is for
consumer or business

        # Validate user credentials
        success, account = UserAuth.login_user(email, password)

        if success:
            # Create a User object for session management
            user = User(account['id'], account['name'], account['email'])
            login_user(user)  # Log in the user using Flask-Login
            flash("Login successful!", "success")

            # Redirect user based on account type
            if account_type == 'consumer':
                return redirect(url_for('consumer'))  # Redirect to consumer dashboard
            else:
                return redirect(url_for('business'))  # Redirect to business dashboard
```

```
        else:
            flash("Invalid credentials. Please try again.", "danger")
            return redirect(url_for('index'))  # Redirect to Home page on failed login

    return render_template('index.html')  # Render login page for GET requests



# Route for adding an address (general use)
@app.route('/add_address', methods=['POST', 'GET'])
@login_required  # Ensures only logged-in users can access this route
def add_address():
    """
    Allows a logged-in user to add a new address.
    - GET request: Displays the address form.
    - POST request: Processes form data and adds a new address to the database.
    """
    if request.method == "POST":
        try:
            # Attempt to add the address using form data
            success = Address.add_address(
                u_id=current_user.id,  # Gets the current logged-in user's ID
                street_no=request.form['street_no'],
                address_line1=request.form['address_line1'],
                address_line2=request.form.get('address_line2'),  # Optional field
                city=request.form['city'],
                region=request.form.get('region'),  # Optional field
                postal_code=request.form['postal_code'],
                country=request.form['country']
            )
            if success:
                flash("Address added successfully!", "success")  # Success message
            else:
                flash("Failed to add address. Please try again.", "danger")  # Failure
message
        except Exception as e:
            flash(f"An error occurred: {str(e)}", "danger")  # Handle unexpected errors

        return redirect(url_for('add_address'))  # Reload the form after submission

    return render_template('add_address.html')  # Render address form for GET requests



# Route for adding a user-specific address (might be for a different user role or use
case)
@app.route('/add_address_u', methods=['POST', 'GET'])
@login_required  # Ensures only logged-in users can access this route
def add_address_u():
    """
```

```
    Allows a logged-in user to add an address (specific to a user role or different use
case).
    - GET request: Displays the user-specific address form.
    - POST request: Processes form data and adds the address to the database.
    """
    if request.method == "POST":
        try:
            # Attempt to add the address using form data
            success = Address.add_address(
                u_id=current_user.id,  # Gets the current logged-in user's ID
                street_no=request.form['street_no'],
                address_line1=request.form['address_line1'],
                address_line2=request.form.get('address_line2'),  # Optional field
                city=request.form['city'],
                region=request.form.get('region'),  # Optional field
                postal_code=request.form['postal_code'],
                country=request.form['country']
            )
            if success:
                flash("Address added successfully!", "success")  # Success message
            else:
                flash("Failed to add address. Please try again.", "danger")  # Failure
message
        except Exception as e:
            flash(f"An error occurred: {str(e)}", "danger")  # Handle unexpected errors

        return redirect(url_for('add_address_u'))  # Reload the form after submission

    return render_template('add_address_u.html')  # Render user-specific address form
for GET requests


# Route for business dashboard (for sellers)
@app.route('/business')
@login_required  # Ensures only logged-in users can access this route
def business():
    """
    Displays the business dashboard, where sellers can view orders placed for their
products.
    Fetches order details, including buyer information and delivery address.
    """
    try:
        # Create a database cursor to execute queries
        cursor = mysql.connection.cursor()

        # Fetch orders where the logged-in user is the seller
        cursor.execute("""
            SELECT
                o.order_id,
```

```
                u.name as uname,  # Buyer's name
                p.name,  # Product name
                u.mobile_number,  # Buyer's contact number
                o.price,  # Price of the order
                o.quantity,  # Quantity ordered
                a.address,  # Shipping address
                o.status  # Order status (e.g., pending, shipped)
            FROM orders o
            JOIN products p ON o.product_id = p.product_id  # Link orders to products
            JOIN userAuth u ON o.user_id = u.id  # Link orders to buyers
            JOIN userAddress a ON o.address_id = a.address_id  # Link orders to
shipping address
            WHERE p.user_id = %s  # Fetch only orders belonging to the logged-in seller
            ORDER BY o.order_id DESC  # Display most recent orders first
        """, (current_user.id,))

        # Fetch all orders and format them into a list of dictionaries
        orders = cursor.fetchall()
        column_names = [desc[0] for desc in cursor.description]  # Extract column names
        orders = [dict(zip(column_names, row)) for row in orders]  # Convert query
result to dictionary format

        # Close the cursor after fetching data
        cursor.close()

        # Render the business dashboard with fetched order details
        return render_template('Business.html', name=current_user.id, orders=orders)

    except Exception as e:
        print(f"Error fetching orders: {e}")  # Log error for debugging
        return render_template('error.html', message="Unable to fetch business
orders.")  # Display error page


# Route for consumer dashboard (for buyers)
@app.route('/consumer')
@login_required  # Ensures only logged-in users can access this route
def consumer():
    """
    Displays the consumer dashboard, where buyers can browse available products.
    Filters out products added by the logged-in user (since they shouldn't see their
own products).
    """
    cursor = mysql.connection.cursor()

    # Fetch all available products except those added by the logged-in user
    cursor.execute("SELECT * FROM products WHERE user_id != %s AND quantity > 0",
(current_user.id,))
    products = cursor.fetchall()
```

```
    # Close the cursor after fetching data
    cursor.close()

    # Render the consumer dashboard with available products
    return render_template('consumer.html', products=products, name=current_user.name)


# Route for uploading a new product (for sellers)
@app.route('/upload', methods=['POST', 'GET'])
@login_required  # Ensures only logged-in users can access this route
def upload():
    """
    Allows sellers to upload new products with an image, description, price, and other
details.
    """
    if request.method == "POST":
        # Check if an image file is selected
        if 'image' not in request.files or request.files['image'].filename == '':
            flash("No file selected!", "warning")  # Show warning message if no file is
chosen
            return redirect(url_for('upload'))  # Redirect back to the upload page

        # Retrieve the uploaded file
        file = request.files['image']
        filename = file.filename  # Get the file name

        # Define file storage path
        file_path = os.path.join(app.config['UPLOAD_FOLDER'], filename)
        file.save(file_path)  # Save the file in the specified directory

        # Define database path for accessing the image
        db_file_path = f'static/uploads/{filename}'

        # Insert product details into the database
        cursor = mysql.connection.cursor()
        cursor.execute(
            """
            INSERT INTO products (user_id, name, price, category, quantity,
description, image_path)
            VALUES (%s, %s, %s, %s, %s, %s, %s)
            """,
            (current_user.id,  # Seller ID
             request.form['name'],  # Product name
             request.form['price'],  # Price
             request.form['category'],  # Category
             request.form['quantity'],  # Available stock
             request.form['desc'],  # Product description
             db_file_path)  # Image file path
```

```
        )

        # Commit the transaction and close the cursor
        mysql.connection.commit()
        cursor.close()

        flash("Product uploaded successfully!", "success")  # Success message
        return redirect(url_for('upload'))  # Redirect back to the upload page

    # Render the product upload form
    return render_template('upload.html')


# Route to view all products uploaded by the current user (seller)
@app.route('/view_products')
@login_required  # Ensures only logged-in users can access this route
def view_products():
    """
    Fetches and displays all products uploaded by the logged-in user.
    """
    cursor = mysql.connection.cursor()

    # Fetch all products added by the logged-in user (seller)
    cursor.execute("SELECT * FROM products WHERE user_id=%s", (current_user.id,))
    products = cursor.fetchall()

    # Close the cursor after fetching data
    cursor.close()

    # Render the view_products page with the seller's product listings
    return render_template('view_products.html', products=products)


# Route to add a product to the shopping cart
@app.route('/add_cart/<int:product_id>', methods=['GET'])
@login_required  # Ensures only logged-in users can access this route
def add_cart(product_id):
    """
    Adds a selected product to the shopping cart for the logged-in user.
    Redirects the user to the cart page after adding the product.
    """
    user_id = current_user.id  # Get the logged-in user's ID

    # Call the Cart model's method to add the product to the cart
    Cart.add_to_cart(user_id, product_id)

    # Redirect to the cart page to view the updated cart
    return redirect(url_for('cart'))
```

```
# Route to display the user's shopping cart
@app.route('/cart')
@login_required  # Ensures only logged-in users can access this route
def cart():
    """
    Displays the shopping cart for the logged-in user, including the selected products.
    Also fetches user addresses for order placement.
    """
    cursor = mysql.connection.cursor()

    # Fetch all saved addresses of the logged-in user
    addresses = Address.get_user_addresses(current_user.id)

    # Fetch all products in the cart for the logged-in user
    cursor.execute("""
        SELECT p.*, c.* FROM cart c
        JOIN products p ON c.product_id = p.product_id
        WHERE c.user_id = %s
    """, (current_user.id,))

    products = cursor.fetchall()  # Retrieve all cart items

    # Close the cursor after fetching data
    cursor.close()

    # Render the cart template with cart products and saved addresses
    return render_template('cart.html', products=products, address=addresses)


# Route to remove an item from the shopping cart
@app.route('/remove_from_cart/<int:cart_id>', methods=['POST', 'GET'])
@login_required  # Ensures only logged-in users can access this route
def remove_from_cart(cart_id):
    """
    Removes a specific product from the shopping cart.
    Redirects back to the cart page after deletion.
    """
    user_id = current_user.id  # Get the logged-in user's ID

    # Create a database cursor to execute queries
    conn = mysql.connection.cursor()

    # Delete the item from the cart based on cart_id
    conn.execute("DELETE FROM cart WHERE cart_id = %s", (cart_id,))

    # Commit the transaction to save changes
    conn.connection.commit()
```

```
    # Close the database connection
    conn.close()

    # Display a success message
    flash('Item removed from cart', 'success')

    # Redirect to the cart page after removing the item
    return redirect(url_for('cart'))


# Route to delete a product uploaded by the seller
@app.route('/delete/<int:product_id>', methods=['GET'])
@login_required  # Ensures only logged-in users can access this route
def delete(product_id):
    """
    Deletes a product uploaded by the logged-in seller.
    Redirects the seller back to the view_products page after deletion.
    """
    flash("Record Has Been Deleted Successfully", "success")

    # Create a database cursor to execute queries
    cur = mysql.connection.cursor()

    # Delete the selected product from the database
    cur.execute("DELETE FROM products WHERE product_id=%s", (product_id,))

    # Commit the transaction to save changes
    mysql.connection.commit()

    # Close the cursor
    cur.close()

    # Redirect to the view_products page after deletion
    return redirect(url_for('view_products'))


# Route to log out the user
@app.route('/logout')
@login_required  # Ensures only logged-in users can access this route
def logout():
    """
    Logs out the currently logged-in user and redirects them to the home page.
    """
    logout_user()  # Log out the user

    flash("You have been logged out.", "info")  # Display a logout message

    # Redirect to the home page
    return redirect(url_for('home'))
```

```
# Route to update product quantity after a purchase
@app.route('/update_quantity/<int:product_id>/<int:quantity>', methods=['GET', 'POST'])
def update_quantity(product_id, quantity):
    """
    Deducts the purchased quantity from the product's available stock.
    Redirects the user to the all_order page after updating the quantity.
    """
    print(f"Updating product {product_id} quantity by reducing {quantity}")

    # Create a cursor to execute database queries
    cur = mysql.connection.cursor()

    # Update the product quantity by subtracting the purchased quantity
    cur.execute('UPDATE products SET quantity = quantity - %s WHERE product_id = %s',
(quantity, product_id,))

    # Commit the transaction to save the changes
    mysql.connection.commit()

    # Close the cursor to free resources
    cur.close()

    # Redirect to the orders page after updating the quantity
    return redirect(url_for('all_order'))


# Route to handle product purchase
@app.route('/buy/<int:product_id>', methods=['GET', 'POST'])
@login_required  # Ensures only logged-in users can access this route
def buy(product_id):
    """
    Handles the purchasing process for a product.
    - Fetches product details and available user addresses.
    - Ensures the user has an address before proceeding.
    - Handles order placement and updates the product quantity.
    """

    # Fetch product details by product_id
    product = Product.get_product_by_id(product_id)

    # Fetch all saved addresses of the logged-in user
    addresses = Address.get_user_addresses(current_user.id)

    # If no addresses are available, prompt the user to add one before ordering
    if not addresses:
        flash("Please add an address before placing an order.", "warning")
```

```
        return redirect(url_for('add_address_u'))  # Redirect to the address addition
page

    # Process form submission (order placement)
    if request.method == 'POST':
        selected_address = request.form['address']  # Get the selected address ID
        quantity = int(request.form['quantity'])  # Get the quantity entered by the
user
        print(f"Received quantity: {quantity}")

        # Create a database cursor to execute queries
        conn = mysql.connection.cursor()

        # Insert the new order into the orders table
        conn.execute('''INSERT INTO orders (product_id, user_id, quantity, price,
address_id)
                            VALUES (%s, %s, %s, %s, %s)''',
                       (product_id, current_user.id, quantity, product[3] * quantity,
selected_address))

        # Commit the transaction to save the order
        mysql.connection.commit()

        # Close the database connection
        conn.close()

        # Notify the user that the order was placed successfully
        flash("Order placed successfully!", "success")

        # Redirect to update the product quantity after purchase
        return redirect(url_for('update_quantity', product_id=product_id,
quantity=quantity))

    # Render the buy page with product details and available addresses
    return render_template('buy.html', product=product, addresses=addresses)


# Route to handle the checkout process
@app.route('/checkout', methods=['POST'])
def checkout():
    try:
        # Extract cart data (JSON format) from the form submitted by the user
        cart_data = request.form.get("cart_data")
        cart_items = json.loads(cart_data)  # Convert the JSON string into a Python
object (list of items)

        for item in cart_items:
            cart_id = item["product_id"]  # Get product_id from the cart item
            quantity = item["quantity"]   # Get quantity of the product
```

```
            subtotal = item["subtotal"]   # Get subtotal (price * quantity)
            address_id = item["address_id"]  # Get address ID for delivery

            # Create a database cursor and fetch the cart item details
            conn = mysql.connection.cursor()
            conn.execute('''SELECT * FROM cart WHERE cart_id = %s''', (cart_id,))
            cart_details = conn.fetchall()  # Fetch the details of the product in the
cart
            mysql.connection.commit()
            conn.close()

            # Extract product_id from cart details
            product_id = cart_details[0][2]
            print(product_id)  # Debugging output to verify product_id

            # Create a new order entry in the orders table
            conn = mysql.connection.cursor()
            conn.execute('''INSERT INTO orders (product_id, user_id, quantity, price,
address_id)
                            VALUES (%s, %s, %s, %s, %s)''',
                        (product_id, current_user.id, quantity, subtotal, address_id))
            mysql.connection.commit()
            conn.close()

            # Remove the product from the cart after the order is placed
            conn = mysql.connection.cursor()
            conn.execute('''DELETE FROM cart WHERE product_id = %s''', (product_id,))
            mysql.connection.commit()
            conn.close()

            # Update the product's quantity after the order is placed
            cur = mysql.connection.cursor()
            cur.execute('UPDATE products SET quantity = quantity - %s WHERE product_id
= %s', (quantity, product_id,))
            mysql.connection.commit()
            cur.close()

            # Flash a success message
            flash("Order placed successfully!", "success")

        # Redirect to the page showing all orders placed
        return redirect(url_for("all_order"))

    except Exception as e:
        # In case of any errors, return an error message
        return jsonify({"error": str(e)}), 500


# Route to add a new order, called by frontend via JSON
```

```
@app.route('/add_order', methods=['POST'])
@login_required
def add_order():
    try:
        # Parse JSON data from the frontend request
        data = request.get_json()

        # Get the current user_id from Flask-Login
        user_id = current_user.id

        # Extract order details from the received data
        product_id = data.get('product_id')
        quantity = data.get('quantity')
        total_price = data.get('total_price')
        address_id = data.get('address_id')

        # Add the new order to the database using a method from the Order class
        success = Order.add_order(user_id, product_id, total_price, quantity,
address_id)

        if success:
            flash("Order placed successfully!", "success")  # Optional: Flash message
            return jsonify({"success": True, "redirect_url": url_for('all_order')})
        else:
            return jsonify({"success": False, "message": "Failed to place order. Please
try again."}), 500

    except Exception as e:
        # In case of any errors, return an error message
        return jsonify({"success": False, "message": str(e)}), 500


# Route to view all orders placed by the current user
@app.route('/all_order')
@login_required
def all_order():
    try:
        # Create a cursor to interact with the database
        cursor = mysql.connection.cursor()

        # Fetch the current logged-in user's orders
        cursor.execute("SELECT * FROM orders WHERE user_id = %s ORDER BY order_id
DESC", (current_user.id,))
        orders = cursor.fetchall()

        # Close the cursor after fetching the orders
        cursor.close()

        # Fetch the name of the products and addresses related to the orders
```

```
        cur1 = mysql.connection.cursor()
        cur1.execute('SELECT name FROM products WHERE product_id = %s',
(orders[0][2],))
        product_name = cur1.fetchall()
        cur1.close()

        cur2 = mysql.connection.cursor()
        cur2.execute('SELECT address FROM userAddress WHERE address_id = %s',
(orders[0][5],))
        address_d = cur2.fetchall()
        cur2.close()

        # Render the template to display all orders placed by the user
        return render_template('view_all_order.html', orders=orders,
product_name=product_name, address_d=address_d)

    except Exception as e:
        # If there is an error fetching orders, display an error message
        print(f"Error: {e}")
        return render_template('error.html', message="Unable to fetch orders.")


# Route to mark an order as delivered by the business (for sellers)
@app.route('/mark_delivered/<int:order_id>', methods=['POST'])
def mark_delivered(order_id):
    try:
        # Create a cursor to update the order's status to "Delivered"
        conn = mysql.connection.cursor()
        conn.execute('UPDATE orders SET status = %s WHERE order_id = %s', ("Delivered",
order_id))
        mysql.connection.commit()  # Commit the changes
        conn.close()  # Close the cursor

        # Redirect to the business page after marking the order as delivered
        return redirect(url_for('business'))

    except Exception as e:
        # If there is an error while updating the order status, display an error
message
        print(f"Error updating order status: {e}")
        return render_template('error.html', message="Unable to update order status.")


# Path to store reviews
REVIEWS_FILE = "reviews.csv"

# Ensure CSV file has headers
if not os.path.exists(REVIEWS_FILE):
    with open(REVIEWS_FILE, 'w', newline='') as file:
```

```
        writer = csv.writer(file)
        writer.writerow(["Product ID", "User ID", "Rating", "Review"])

# Route to handle product reviews by users
@app.route('/review/<int:product_id>/<int:user_id>', methods=['GET', 'POST'])
def review(product_id, user_id):
    # If the request method is POST, store the review
    if request.method == 'POST':
        review_text = request.form['review']  # Get the review text from the form
        rating = request.form['rating']  # Get the rating from the form

        # Store review in a CSV file
        with open(REVIEWS_FILE, 'a', newline='') as file:
            writer = csv.writer(file)
            # Write the product_id, user_id, rating, and review text to the CSV
            writer.writerow([product_id, user_id, rating, review_text])

        # Redirect to the all_order page after storing the review
        return redirect(url_for('all_order'))

    # If the request method is GET, render the review page
    return render_template('review.html', product_id=product_id, user_id=user_id)


# Define the CSV file for contact queries
CONTACT_FILE = "contact.csv"

# Ensure the CSV file has headers if it does not exist
if not os.path.exists(CONTACT_FILE):
    with open(CONTACT_FILE, 'w', newline='') as file:
        writer = csv.writer(file)
        # Write the header for the contact CSV file
        writer.writerow(["NAME", "EMAIL ID", "QUERY"])

# Route to handle the contact form submissions
@app.route('/contact', methods=['GET', 'POST'])
def contact():
    # If the request method is POST, store the contact query
    if request.method == 'POST':
        name = request.form['name']  # Get the user's name from the form
        email = request.form['email']  # Get the user's email from the form
        query = request.form['query']  # Get the user's query from the form

        # Store the contact query in the CSV file
        with open(CONTACT_FILE, 'a', newline='') as file:
            writer = csv.writer(file)
            # Write the name, email, and query to the CSV file
            writer.writerow([name, email, query])
```

```
        # Redirect to the contact page after submitting the form
        return redirect(url_for('contact'))

    # If the request method is GET, render the contact form
    return render_template('contact_us.html')

# Route for search functionality with filters
@app.route('/search_filter', methods=['GET'])
@login_required  # Only authenticated users can access this route
def search_filter():
    try:
        # Get the search query, price filter, and category filter from the request
parameters
        search_query = request.args.get('query', '').strip().lower()  # Ensure query is
lowercase
        filter_option = request.args.get('filter', 'default')  # Default filter option
        category_filter = request.args.get('category', 'all').lower()  # Default
category is 'all'

        conn = mysql.connection.cursor()

        # Base SQL query: Exclude products of the current user and those with zero
quantity
        query = "SELECT * FROM products WHERE user_id != %s AND quantity > 0 AND
LOWER(name) LIKE %s"
        params = (current_user.id, f"%{search_query}%")

        # Apply category filter if specified
        if category_filter != "all":
            query += " AND LOWER(category) = %s"
            params += (category_filter,)

        # Apply sorting based on the price filter
        if filter_option == "low-to-high":
            query += " ORDER BY price ASC"  # Sort by price in ascending order
        elif filter_option == "high-to-low":
            query += " ORDER BY price DESC"  # Sort by price in descending order

        # Execute the SQL query with the provided parameters
        conn.execute(query, params)
        products = conn.fetchall()  # Fetch all the matching products
        conn.close()

        # Convert the product data into a list of dictionaries for JSON response
        product_list = []
        for product in products:
            product_list.append({
                "product_id": product[0],
                "user_id": product[1],
```

```
                "name": product[2],
                "price": product[3],
                "category": product[4],
                "quantity": product[5],
                "description": product[6],
                "image_path": product[7]
            })

        # Return the product list as a JSON response
        return jsonify(product_list)

    except Exception as e:
        # Return an error response if something goes wrong
        return jsonify({"error": str(e)}), 500


# Route to handle product update
@app.route('/update_product/<int:product_id>', methods=['POST', 'GET'])
def update_p(product_id):
    # Create a cursor to interact with the database
    conn = mysql.connection.cursor()

    # Fetch the product details based on the product_id
    conn.execute('SELECT * FROM products WHERE product_id = %s', (product_id,))
    product = conn.fetchone()  # Fetch one product record
    conn.close()  # Close the cursor after retrieving product details

    # If the request method is POST (form submission), update the product
    if request.method == 'POST':
        # Get the file from the form submission (if provided)
        file = request.files.get('image')

        # If a new image file is uploaded, save it to the designated folder
        if file and file.filename != '':
            filename = file.filename  # Get the uploaded file's filename
            file_path = os.path.join(app.config['UPLOAD_FOLDER'], filename)  # Define
the path to save the file
            file.save(file_path)  # Save the file
            db_file_path = f'static/uploads/{filename}'  # Store relative path in the
database
        else:
            db_file_path = product[7]  # If no new image is uploaded, retain the
existing image

        # Update the product details in the database with the new data
        conn = mysql.connection.cursor()
        conn.execute('''
            UPDATE products
```

```
            SET name = %s, price = %s, category = %s, quantity = %s, description = %s,
image_path = %s
            WHERE user_id = %s AND product_id = %s
        ''', (request.form['name'], request.form['price'], request.form['category'],
request.form['quantity'],
              request.form['desc'], db_file_path, current_user.id, product_id))

        # Commit the transaction to save the changes to the database
        mysql.connection.commit()
        conn.close()  # Close the cursor after committing the changes

        # Redirect to the 'view_products' page to show the updated product
        return redirect(url_for('view_products'))

    # If the request method is GET, render the product update page
    return render_template('update_product.html', product=product)

if __name__ == '__main__':
    # UserAuth.create_user_table()
    # Address.create_address_table()
    # Product.create_product_table()
    # Cart.create_cart_table()
    # Order.create_order_table()
    app.run(debug=True)
```

RESULT

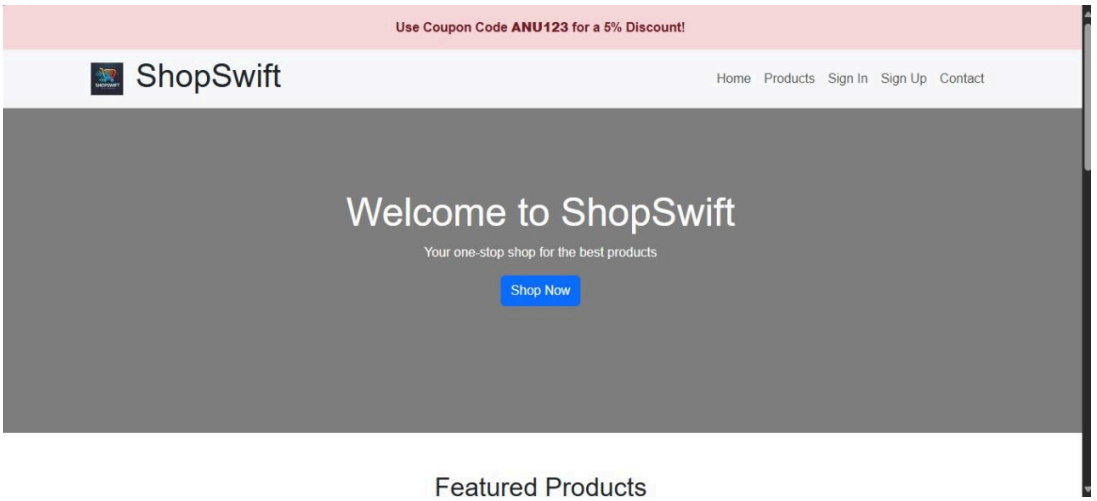

Fig 3: Home page

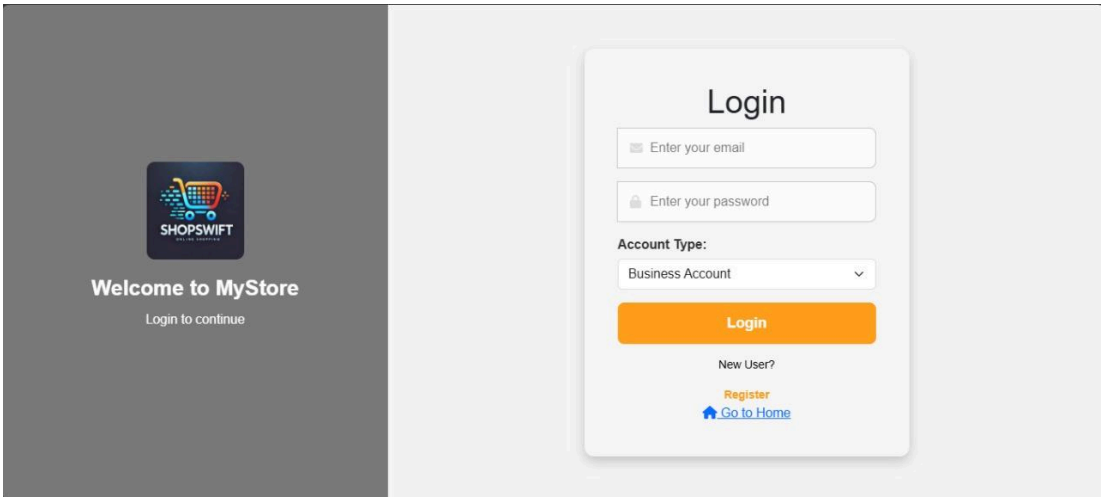

Fig 4: Login page

Fig 5: Business dashboard

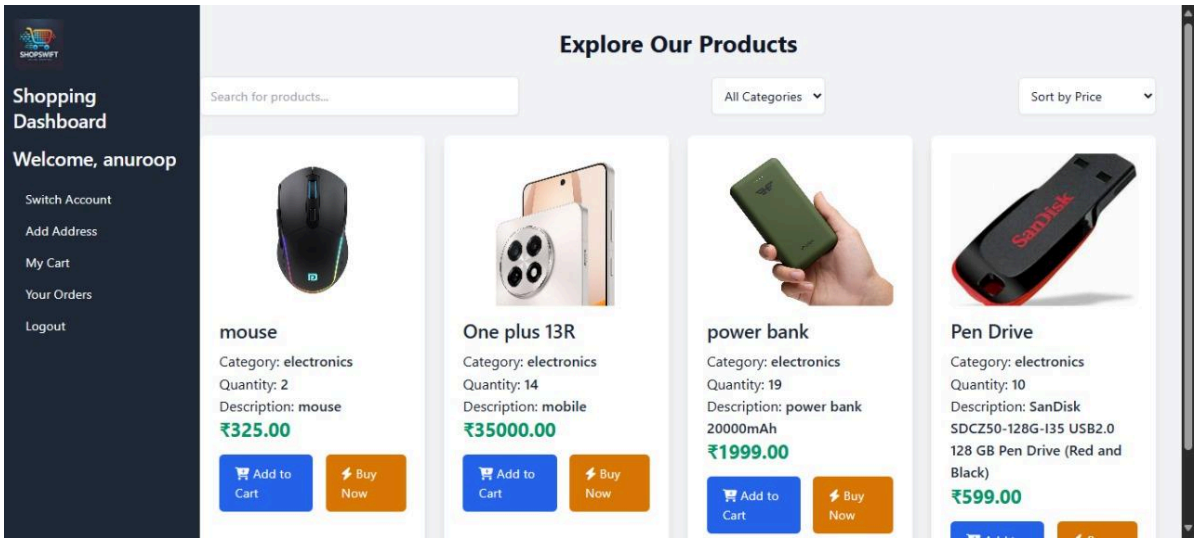

Fig 6: Shopping dashboard page

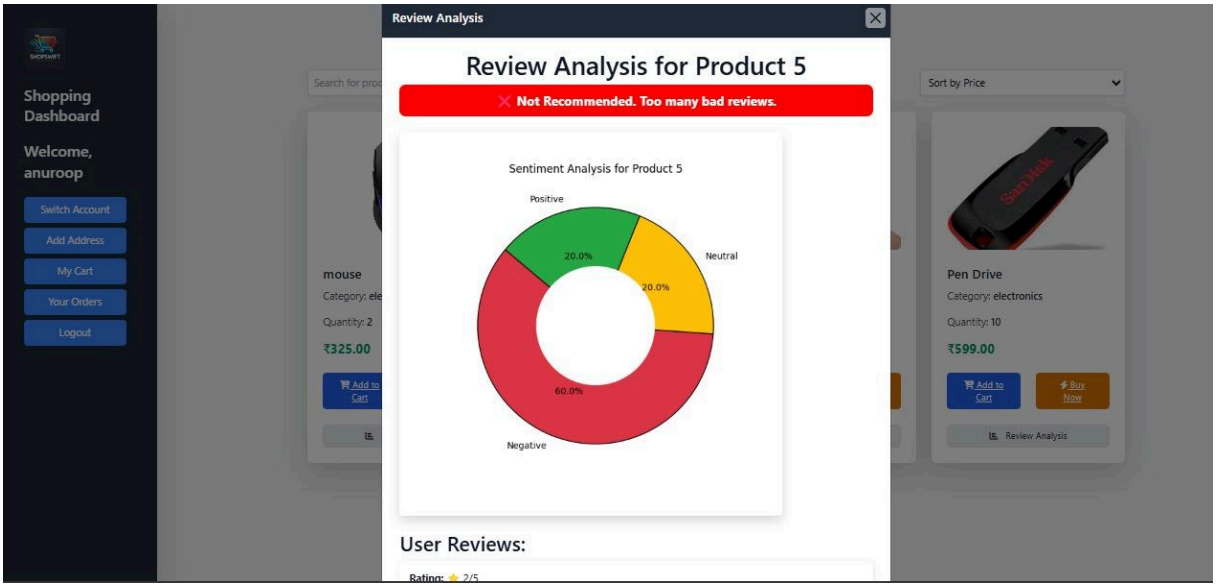

Fig 7: Review analysis page

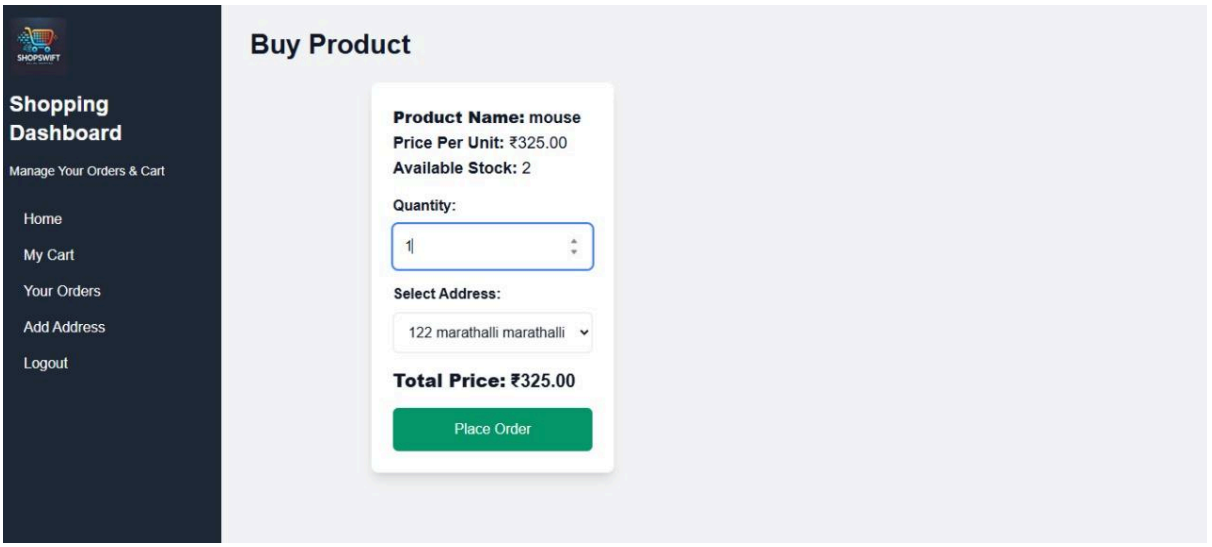

Fig 8: Buy section

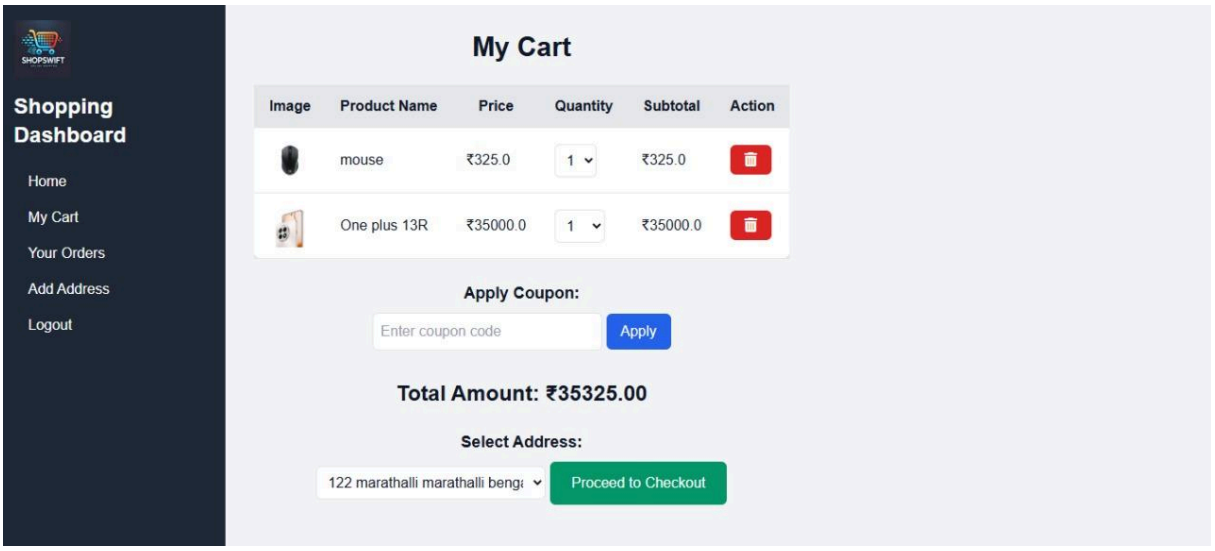

Fig 9: Cart page

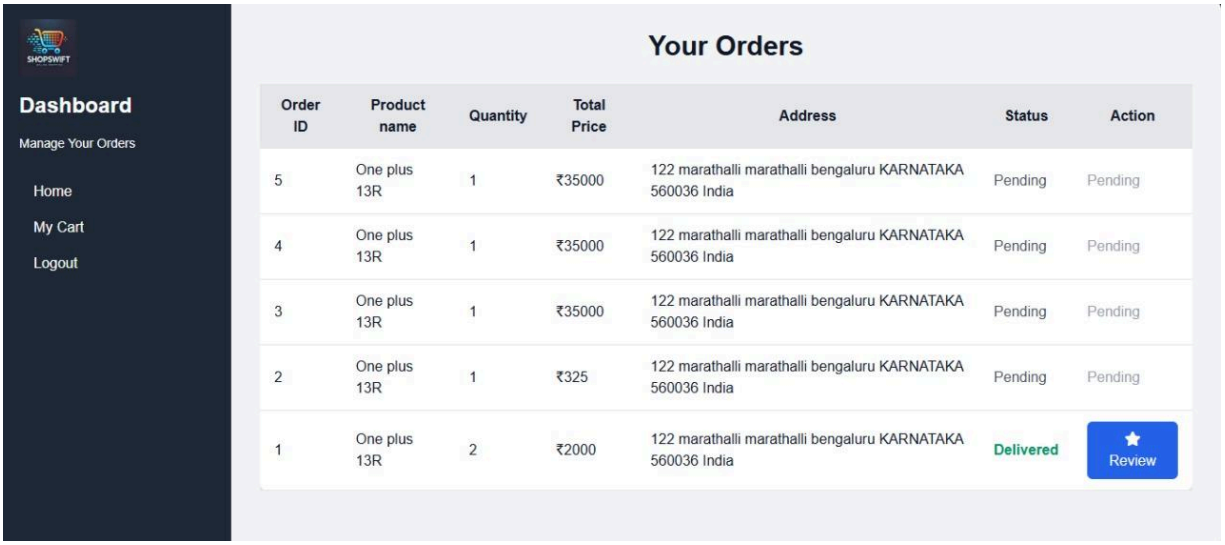

Fig 10: Consumer orders page

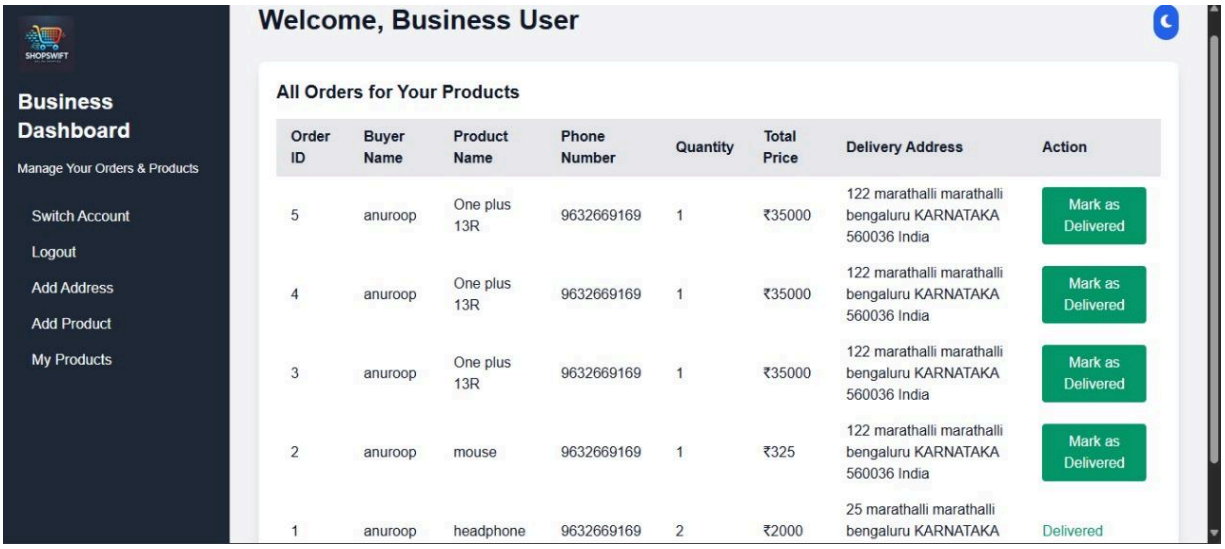

Fig 11: Placed order page from consumer

ADVANTAGES

- **Unified Account System -** Users do not need separate accounts for buying and selling, simplifying the overall experience.
- **Improved User Experience -** The seamless role-switching feature enhances usability and efficiency.
- **Scalable Architecture -** The system is designed to handle large numbers of products
- **Structured Database Design -** The relational database ensures optimal performance and data consistency.

FUTURE ENHANCEMENTS

Integration with Payment Gateways

- Secure Online Payments: Integrate Razorpay, Stripe, or PayPal for online transactions.
- Cash on Delivery (COD): Offer flexible payment options for users.

AI-Based Recommendation System

- Personalized Shopping Experience – AI will recommend products based on user preferences, past purchases, and browsing behaviour.
- Smart & Real-Time Suggestions – AI will analyze real-time data to suggest trending, relevant, and best-matching products instantly.

CONCLUSION

The Advanced E-commerce Recommendation System successfully delivers an innovative approach to online shopping by integrating both buying and selling functionalities within a single user account. The use of Flask and MySQL ensures a robust backend, while AI-driven recommendations enhance the shopping experience. The system addresses key inefficiencies in traditional e-commerce platforms and provides a scalable, secure, and user-friendly solution. Future enhancements, including payment integration and AI-powered analytics, will further solidify the platform's position as a next-generation e-commerce solution.