A close up of a logo

AI-generated content may be incorrect.

**CAP463 – CAPTSONE PROJECT**

**GROCERY STORE CART SYSTEM PROJECT**

**BY**

**12200795 – ZUNEID KAEMBA**

**SECTION D2207SUBMITTED TO**

**HOD**

**SCHOOL OF COMPUTER APPLICATIONS,**

**LOVELY PROFESSIONAL UNIVERSITY,**

**JALANDHAR, PUNJAB.**

**📘 Table of Contents**

**1. Introduction**

* **1.1 Project Overview**
* **1.2 Technologies Used**
  + **Frontend: HTML, CSS, JavaScript, Bootstrap**
  + **Backend: Python Flask**
  + **Database: MySQL**
* **1.3 Technologies Used**
* **1.4 Why a 3-Tier Architecture**

**2. UI Mockup Design (Moqups.com)**

* **2.1 Purpose of Using Moqups**
* **2.2 Cart Page Wireframe**
* **2.3 Final Mockup Overview**

**3. System Overview**

* **3.1 Project Structure**
* **3.2 Frontend–Backend Flow**
* **3.3 MySQL Integration Overview**

**4. Frontend Development**

* **4.1 Cart Page (HTML & Bootstrap Layout)**
* **4.2 Styling (CSS)**
* **4.3 JavaScript Logic**
  + **Adding Items to Cart**
  + **Updating Quantity**
  + **Removing Items**
  + **Total Price Calculation**

**5. Backend Development (Flask)**

* **5.1 Flask Setup and Configuration**
* **5.2 API Routes for Cart Operations**
  + **Add to Cart**
  + **Update Cart**
  + **Remove from Cart**
  + **Fetch Cart Items**
* **5.3 Session Management**

**6. Database Design (MySQL)**

* **6.1 Cart-Related Tables**
  + **products**
  + **cart\_items**
* **6.2 Sample Schema and Queries**

**7. Testing**

* **7.1 Manual Testing Scenarios**
* **7.2 Common Bugs and Fixes**

**8. Conclusion**

* **8.1 Summary of the Cart Feature**
* **8.2 Possible Enhancements (e.g., User Login, Order Checkout)**

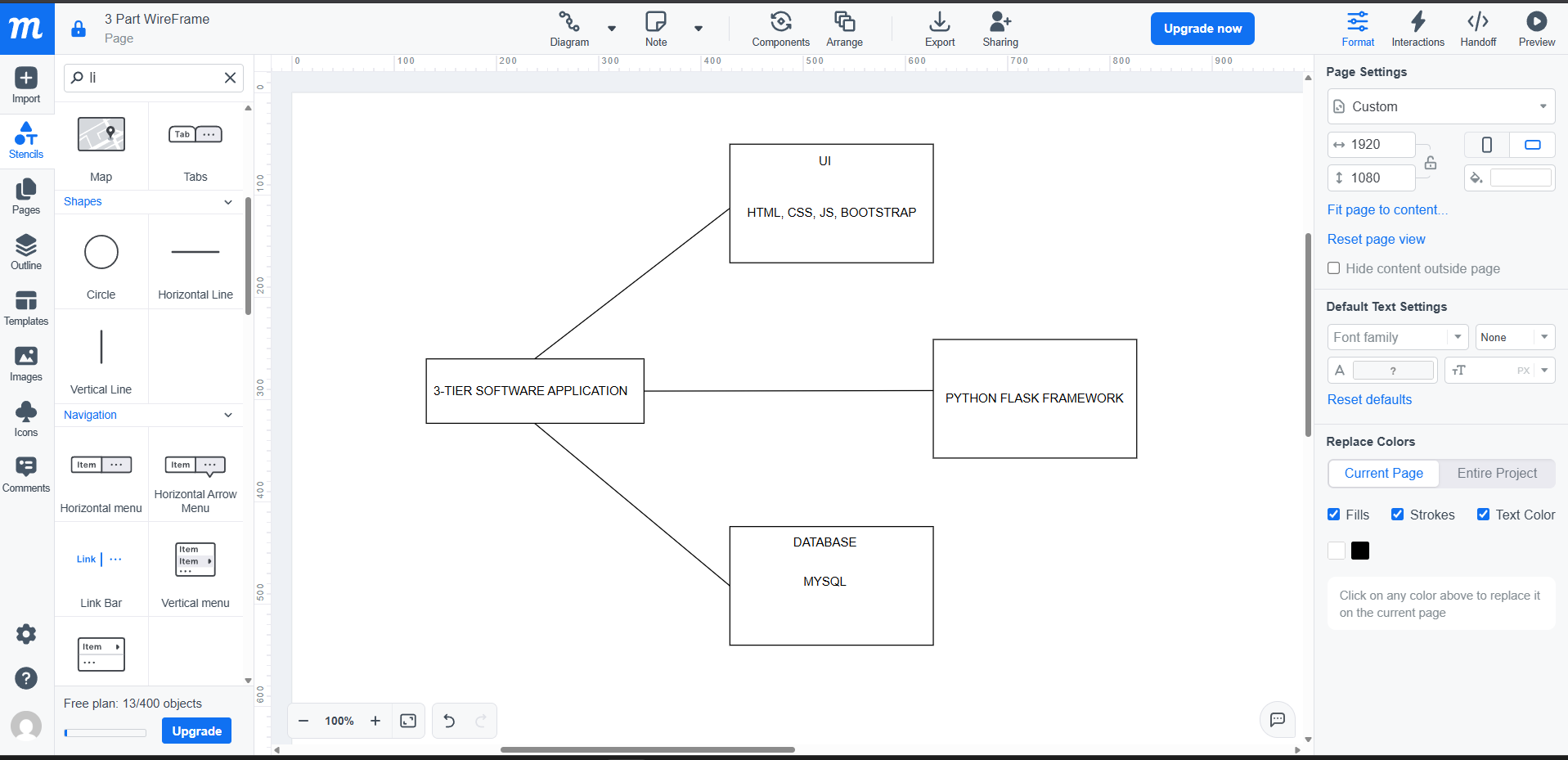
**1. Introduction**

**1.1 Project Overview**

This project is a simplified Grocery Store Cart System built as a web application. It focuses solely on the shopping cart functionality, allowing users to browse a selection of grocery products and manage their cart before purchase. The core features include adding products to the cart, updating quantities, removing items, and viewing the total cost dynamically. The system is designed as a full-stack application using modern web technologies and backend services.

This project simulates a real-world shopping experience and can serve as a building block for a larger e-commerce or inventory-based application. The cart system is dynamic, interactive, and responsive across devices thanks to the use of Bootstrap and JavaScript.

**THREE-TIER SYSTEM MOCKUP**

****

**1.2 Objectives**

* **Provide a user-friendly interface for grocery shopping.**
* **Enable real-time cart operations: add, update, and remove.**
* **Maintain session data to track cart contents.**
* **Connect to a MySQL database to fetch and display product data.**

**1.3 Technologies Used**

To implement this project, the following technologies were used:

**Frontend**

* + **HTML:** For structuring the web pages.
  + **CSS:** For custom styling.
  + **JavaScript:** For dynamic behavior and cart logic (add/remove/update).
  + **Bootstrap:** For responsive layout and UI components.

**Backend**

* + Python (Flask Framework): A lightweight web framework used to handle HTTP routes, API endpoints, and server-side logic.

**Database**

* + **MySQL:** Used to store product information and cart data.

These technologies together allow for seamless client-server interaction and data handling, creating a responsive and functional cart system.

**1.4 Why a 3-Tier Architecture?**

The system follows a clear separation of concerns through a 3-tier architecture:

* **Presentation Layer** (UI): HTML, CSS, JS, Bootstrap.
* **Application Layer** (Logic): Python Flask Framework.
* **Data Layer**: MySQL database.

**Chapter 2: UI Mockups (Moqups.com)**

**2.1 Purpose of Using Moqups**

Moqups.com was used to design the cart system layout before implementation. This tool helped create visual wireframes to map out UI components, allowing fast iteration on design ideas.

Before starting the actual development, Moqups.com was used to create a Visual representation of the cart interface. Moqups helped visualize the user experience and plan the layout of essential elements such as:

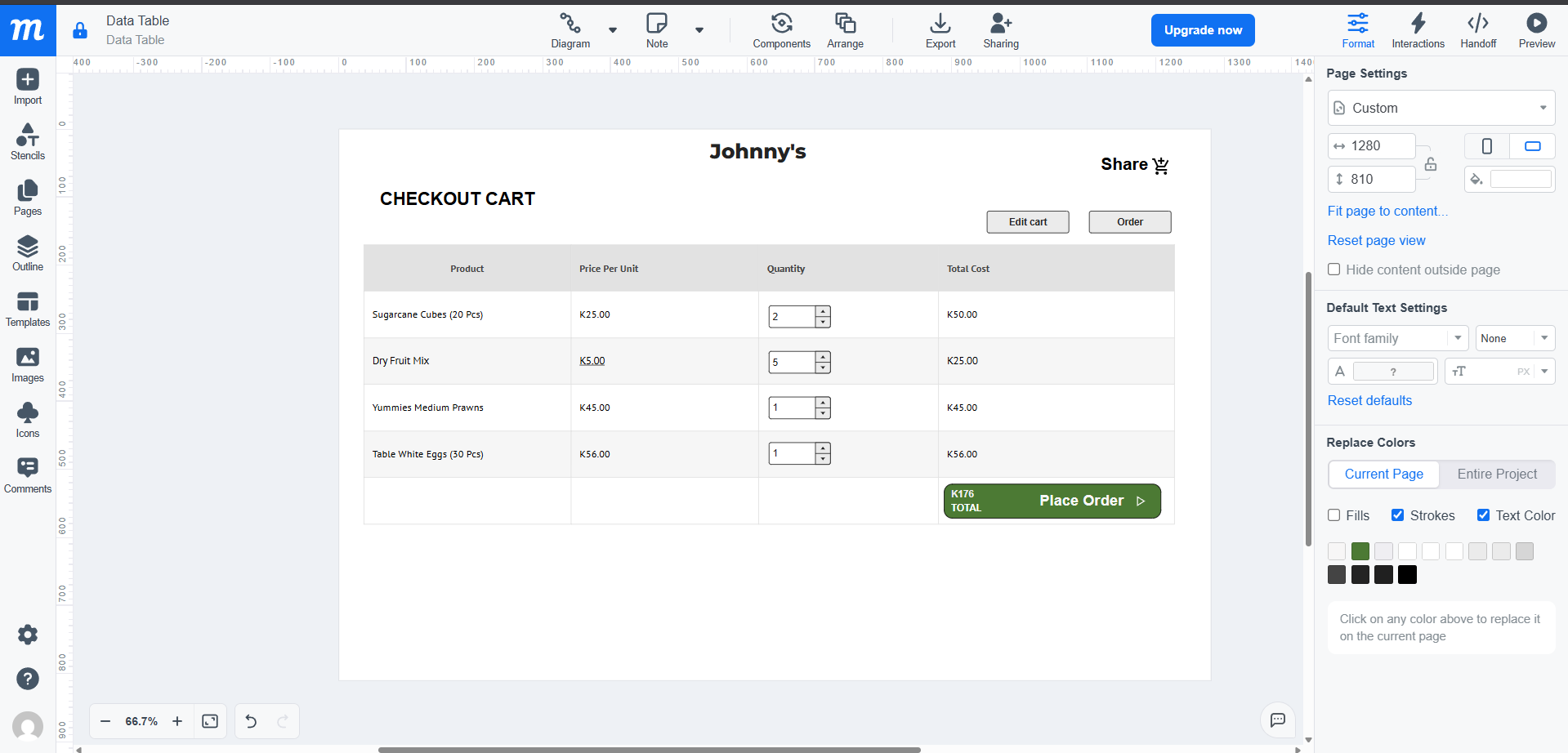
2.2 **Cart Page WireFrame**

* **Product listing cards**
* **“Add to Cart” buttons**
* **Cart icon or summary section**
* **Quantity selector**
* **Price display and total section**

Creating mockups on Moqups allowed for early feedback and iterative design changes without writing a single line of code. It ensured a clear vision of the final interface and streamlined frontend development.

**2.3 Final Mockup Overview**

The final design included a header with the shop name, a tabular display of cart items, and a summary section showing total price and a checkout button. This wireframe served as the blueprint during frontend development.

****

**Chapter 3: System Overview**

**3.1 Project Structure**

**Grocery app/**

1. **Backend/**
   1. **Products\_dao.py**

from sql\_connection import get\_sql\_connection  
  
def get\_all\_products(connection):  
 cursor = connection.cursor()  
 query = """  
 SELECT p.product\_id, p.name, p.uom\_id, p.price\_per\_unit, u.uom\_name  
 FROM products p  
 INNER JOIN uom u ON p.uom\_id = u.uom\_id  
 """  
 cursor.execute(query)  
 result = cursor.fetchall()  
 response = [{  
 'product\_id': row[0],  
 'name': row[1],  
 'uom\_id': row[2],  
 'price\_per\_unit': float(row[3]),  
 'uom\_name': row[4]  
 } for row in result]  
 return response  
  
def get\_product\_by\_id(connection, product\_id):  
 cursor = connection.cursor()  
 query = """  
 SELECT product\_id, name, uom\_id, price\_per\_unit  
 FROM products  
 WHERE product\_id = %s  
 """  
 cursor.execute(query, (product\_id,))  
 result = cursor.fetchone()  
 if result:  
 return {  
 'product\_id': result[0],  
 'name': result[1],  
 'uom\_id': result[2],  
 'price\_per\_unit': float(result[3])  
 }  
 return None  
  
def insert\_new\_product(connection, product):  
 cursor = connection.cursor()  
 query = "INSERT INTO products (name, uom\_id, price\_per\_unit) VALUES (%s, %s, %s)"  
 data = (product['product\_name'], product['uom\_id'], product['price\_per\_unit'])  
 cursor.execute(query, data)  
 connection.commit()  
 return cursor.lastrowid  
  
def update\_product(connection, product\_id, product):  
 cursor = connection.cursor()  
 query = """  
 UPDATE products  
 SET name = %s, uom\_id = %s, price\_per\_unit = %s  
 WHERE product\_id = %s  
 """  
 data = (product['product\_name'], product['uom\_id'], product['price\_per\_unit'], product\_id)  
 cursor.execute(query, data)  
 connection.commit()  
  
def delete\_product(connection, product\_id):  
 cursor = connection.cursor()  
 cursor.execute("DELETE FROM products WHERE product\_id = %s", (product\_id,))  
 connection.commit()

* 1. **Sql\_connection.py**

import mysql.connector  
  
def get\_sql\_connection():  
 return mysql.connector.connect(  
 user='root',  
 password='JohnnyDxng0!',  
 host='127.0.0.1',  
 database='grocery\_store'  
 )

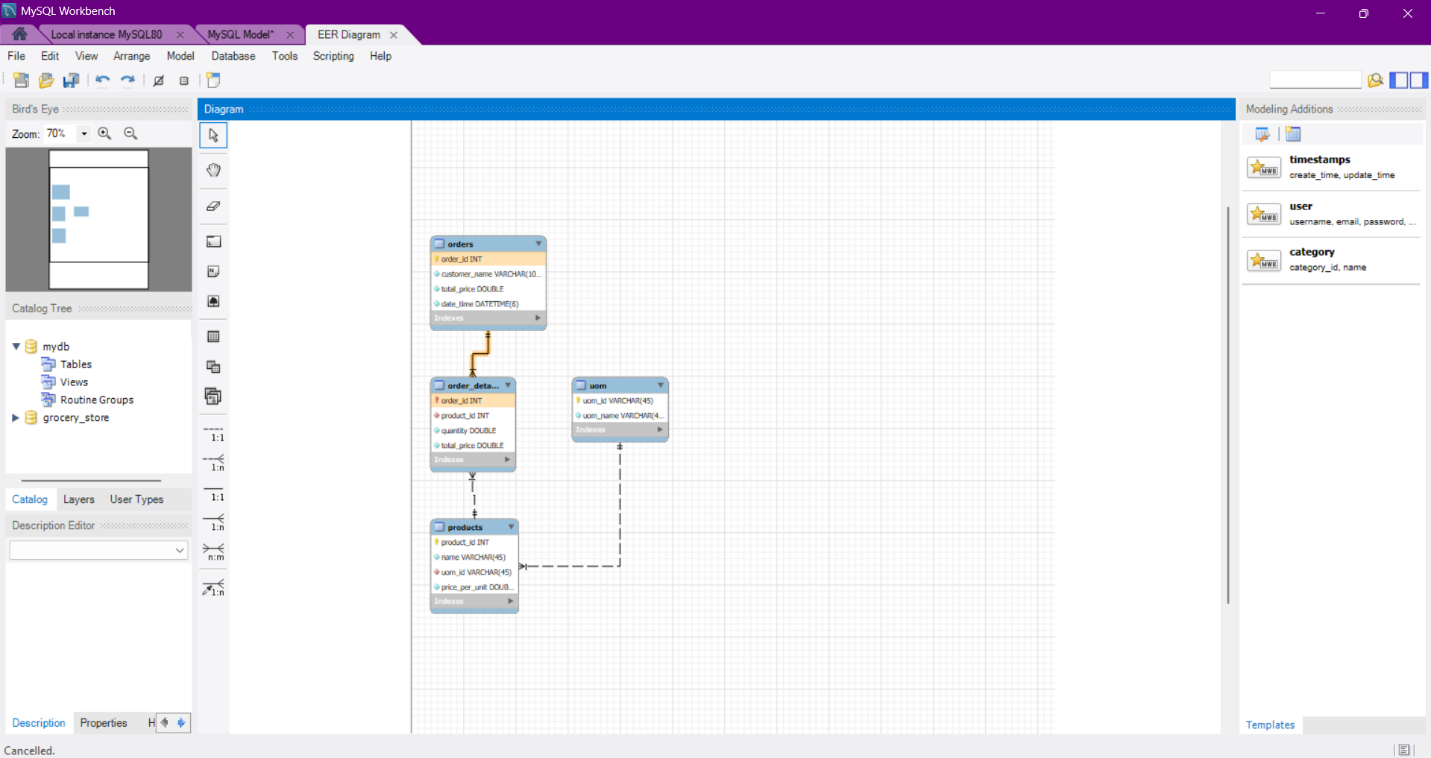
* 1. **Flaskserver.py**

from flask import Flask, request, jsonify  
import products\_dao  
from sql\_connection import get\_sql\_connection  
app = Flask(\_\_name\_\_)  
  
connection = get\_sql\_connection()  
@app.route('/getProducts', methods=['GET'])  
def get\_products():  
 conn = get\_db\_connection()  
 cur = conn.cursor()  
 cur.execute("""  
 SELECT p.product\_id, p.name, p.price\_per\_unit, p.uom\_id, u.uom\_name  
 FROM products p  
 JOIN units u ON p.uom\_id = u.uom\_id  
 """)  
 products = cur.fetchall()  
 conn.close()  
 return jsonify([dict(row) for row in products])  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("Starting Flask Server")  
 app.run(port=5000, debug=True)

* 1. **App.py**

from flask import Flask, request, jsonify  
from flask\_cors import CORS  
from sql\_connection import get\_sql\_connection  
import products\_dao  
  
app = Flask(\_\_name\_\_)  
CORS(app)  
  
@app.route('/getProducts', methods=['GET'])  
def get\_products():  
 conn = get\_sql\_connection()  
 products = products\_dao.get\_all\_products(conn)  
 conn.close()  
 return jsonify(products)  
  
@app.route('/getProduct/<int:product\_id>', methods=['GET'])  
def get\_product(product\_id):  
 conn = get\_sql\_connection()  
 product = products\_dao.get\_product\_by\_id(conn, product\_id)  
 conn.close()  
 if product:  
 return jsonify(product)  
 return jsonify({'error': 'Product not found'}), 404  
  
@app.route('/insertProduct', methods=['POST'])  
def insert\_product():  
 data = request.get\_json()  
 conn = get\_sql\_connection()  
 product\_id = products\_dao.insert\_new\_product(conn, data)  
 conn.close()  
 return jsonify({'product\_id': product\_id})  
  
@app.route('/updateProduct/<int:product\_id>', methods=['PUT'])  
def update\_product(product\_id):  
 data = request.get\_json()  
 conn = get\_sql\_connection()  
 products\_dao.update\_product(conn, product\_id, data)  
 conn.close()  
 return jsonify({'status': 'success'})  
  
@app.route('/deleteProduct', methods=['POST'])  
def delete\_product():  
 data = request.get\_json()  
 conn = get\_sql\_connection()  
 products\_dao.delete\_product(conn, data['product\_id'])  
 conn.close()  
 return jsonify({'status': 'success'})  
  
@app.route('/getUOMs', methods=['GET'])  
def get\_uoms():  
 conn = get\_sql\_connection()  
 cursor = conn.cursor()  
 cursor.execute("SELECT uom\_id, uom\_name FROM uom")  
 uoms = cursor.fetchall()  
 conn.close()  
 return jsonify([{'uom\_id': u[0], 'uom\_name': u[1]} for u in uoms])  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("Starting Flask Server...")  
 app.run(port=5000, debug=True)

1. **User\_Interface/**
   1. Index.html
   2. Styles.css
   3. Script.js
2. **Database.sql/**



**3.2 Frontend–Backend Flow**

1. User interacts with the cart via HTML/JS.
2. JS sends AJAX requests to Flask backend routes.
3. Flask handles request, fetches data from MySQL.
4. Response sent back to frontend to update UI.

**3.3 MySQL Integration**

The database includes two main tables:

* products: stores item name, ID, price, unit.
* uom: unit of measurement reference.

**Chapter 4: Frontend Development**

**4.1 HTML + Bootstrap Layout**

* Tables used for listing cart items.
* Bootstrap grid is used to align elements responsively.

**4.2 CSS Styling**

* Custom styles for table headers, buttons, and price highlights.

**4.3 JavaScript Functionality**

* **Add to Cart**: Listens for a button click, adds item.
* **Update Quantity**: Spinner updates trigger recalculation.
* **Remove Item**: Deletes item row from DOM and backend.
* **Calculate Total**: Recomputes total price on every change.

function updateTotal() {

let total = 0;

document.querySelectorAll('.item-row').forEach(row => {

const price = parseFloat(row.querySelector('.price').textContent);

const qty = parseInt(row.querySelector('.qty').value);

total += price \* qty;

});

document.getElementById('total-cost').textContent = 'K' + total;

}

(Include screenshot of cart page here)

**Chapter 5: Backend Development (Flask)**

**5.1 Flask Setup**

* Created a Flask app with app.py
* Configured CORS, debug mode, and MySQL connector

**5.2 API Routes**

@app.route('/cart/add', methods=['POST'])

def add\_to\_cart():

data = request.json

# Insert logic to add to session or DB

* /cart/add: Adds a product
* /cart/update: Updates quantity
* /cart/remove: Removes a product
* /cart/items: Fetches all cart items

**5.3 Session Management**

* Used Flask sessions to keep cart items persistent during a session

session['cart'] = [{'product\_id': 1, 'qty': 2}]

**Chapter 6: Database Design**

**6.1 Tables**

* **products**:
  + product\_id (PK)
  + name
  + uom\_id (FK)
  + price\_per\_unit
* **uom**:
  + uom\_id (PK)
  + uom\_name

**6.2 Sample SQL Query**

SELECT p.name, p.price\_per\_unit, u.uom\_name

FROM products p

JOIN uom u ON p.uom\_id = u.uom\_id;

(Include Workbench screenshot and ER diagram here)

**Chapter 7: Testing**

**7.1 Manual Testing**

* **Add product**: PASS
* **Update quantity**: PASS
* **Remove item**: PASS
* **Database fetch**: PASS

**7.2 Debugging**

* Fixed "Table 'grocery\_store.oum' doesn't exist" by correcting the table name to uom
* Used terminal logs and MySQL Workbench to test queries

**Chapter 8: Conclusion**

**8.1 Summary**

This project successfully delivered a web-based cart system simulating real-world shopping functionality. It demonstrated the use of a 3-tier architecture and integration of frontend, backend, and database layers effectively.

**8.2 Future Improvements**

* Add User Login and Authentication
* Implement full order placement and payment system
* Enable user profile and order history features

**Chapter 9: Appendix**

**9.1 Installing MySQL Connector in PyCharm**

* Open PyCharm settings > Project Interpreter
* Search and install mysql-connector-python

**9.2 MySQL Workbench Setup**

* Created schema grocery\_store
* Executed table creation SQLs

**9.3 Sample Data**

INSERT INTO products (name, uom\_id, price\_per\_unit)

VALUES ('Toothpaste', 1, 30.00);