RESTAURANT & INVENTORY MANAGEMENT SYSTEM

A MINI-PROJECT REPORT FOR CS23332 - DATABASE MANAGEMENT SYSTEM

Submitted by

Muhilan S	**241701033**
Dhanesh SK	**241701013**

in partial fulfillment of the award of the degree

Of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND DESIGN

RAJALAKSHMI ENGINEERING COLLEGE, CHENNAI

An Autonomous Institute

CHENNAI NOVEMBER 2025

BONAFIDE CERTIFICATE

Certified that this project "RESTAURANT MANAGEMENT SYSTEM" is the bonafide work of "**MUHILAN S, DHANESH SK**" who carried out the project work under my supervision.

SIGNATURE
MRS. D. KALPANA M.E
ASSISTANT PROFESSOR

Dept. of Computer Science and Design,
Rajalakshmi Engineering College
Chennai

This mini project report is submitted for the viva voce examination to be held on _____

INTERNAL EXAMINER **EXTERNAL EXAMINER**

ABSTRACT

This project presents a web-based Restaurant & Inventory Management System designed to eliminate inefficiencies in manual restaurant operations. Built using Streamlit and SQLite3, it integrates order-taking (POS) with real-time inventory control, automates stock management, and provides role-based access for Administrators, Managers, and Staff. The system includes modules for order processing, inventory tracking, user management, and analytics, successfully validating real-time, recipe-based inventory depletion. It serves as an effective proof-of-concept and a foundation for future production-grade applications.

ACKNOWLEDGEMENT

We express our sincere thanks to our beloved and honorable chairman **MR. S. MEGANATHAN and the chairperson DR. M.THANGAM MEGANATHAN** for their timely support and encouragement.

We are greatly indebted to our respected and honorable principal **Dr. S.N. MURUGESAN** for his able support and guidance.

No words of gratitude will suffice for the unquestioning support extended to us by our Head Of The Department **Dr. Mr.S.UMAMAHESHWARARAO** for being ever supporting force during our project work.

We also extend our sincere andOur sincere thanks to our family members, friends and other staff members of computer science engineering hearty thanks to our internal guide **MRS. D. KALPANA**, for her valuable guidance and motivation during the completion of this project.

.

1.MUHILAN S

2.DHANESH SK

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The restaurant industry operates on notoriously tight profit margins, where operational efficiency is not merely a goal but a prerequisite for survival.[1] Despite this, many establishments continue to rely on manual, paper-based systems for their most critical functions: order-taking, billing, and inventory management.[2] These traditional methods are not just dated; they are a source of systemic failure that actively erodes profitability, degrades the customer experience, and creates a high-stress environment for staff.

Scope Of The Work

This project documents the design, development, and implementation of a **Restaurant & Inventory Management System**. The system is a web-based application built using Python, Streamlit, and SQLite3, designed to replace these inefficient manual processes.[53] It provides a single, centralized platform to manage all core restaurant operations, from the moment a customer places an order to the final reconciliation of inventory. By digitally linking the front-of-house (FOH) sales with the back-of-house (BOH) inventory, this system aims to minimize human error, reduce food waste, and provide managers with the real-time data needed for effective decision-making.

Problem Statement

Manual restaurant systems are fundamentally characterized by high-friction processes that are slow, prone to human error, and a source of constant frustration. This inefficiency is not a minor inconvenience but has been identified as a direct "profit killer" and a primary driver of negative customer experiences. The analysis of this problem domain reveals that these failures can be categorized into four interconnected areas:

1. **Order and Billing Inaccuracies:** The reliance on handwritten orders is a primary source of costly errors. Studies indicate that nearly 60% of order mistakes originate from miscommunication between waitstaff and the kitchen, often due to illegible handwriting. This directly results in customers receiving the wrong dish, leading to food waste, customer disappointment, and the added cost of correcting the mistake. Manual billing is also highly susceptible to calculation errors or forgotten items.
2. **Service and Time Inefficiency:** The physical workflow of a manual system is inherently slow. The time spent by waitstaff writing down an order, walking it to the kitchen, and later manually calculating a bill creates significant service delays. This manifests for the customer as long wait times for a table, for food to arrive, and for the bill to be processed. This is compounded by the "tablet hell" of modern delivery apps, which forces staff to manually re-enter orders from multiple devices, wasting time and increasing the risk of errors.
3. **The Inventory Management Conundrum:** This is the most complex and financially damaging

failure. Manual inventory tracking is a "delicate balance" [9] that is susceptible to errors.[16] This failure manifests in two destructive ways:
    - **Overstocking:** Lacking accurate, real-time data, managers over-order, tying up capital and leading to significant financial loss from spoilage.[8]
    - **Understocking:** A manager, relying on an inaccurate physical count, fails to reorder a key ingredient.[1] During a busy shift, the kitchen runs out, forcing staff to tell a customer, "We're out of that" [11], resulting in an immediate lost sale and a disappointed customer.[1]
4. **The Data Black Hole:** Running a restaurant with a manual system is "like driving blind".[5] There is no reliable mechanism for gathering sales insights [6], tracking real-time inventory levels [1], or monitoring food costs and waste.[12] This "lack of forecasting" [9] makes data-driven decision-making impossible.

Aim and Objectives of the Project

The primary aim of this project is to solve the problems outlined above by developing a robust and integrated digital system.[14] The overarching goal is to maximize profit by increasing operational efficiency and decreasing overheads, while simultaneously maintaining or enhancing customer satisfaction.[3]

To achieve this, the project has the following specific objectives:

- **Automate and Centralize Communication:** To design a system that eliminates paper-based communication between the front-of-house (FOH) and back-of-house (BOH), minimizing the probability of human errors.[53]
- **Streamline Sales and Billing:** To implement a digital Point of Sale (POS) system that provides efficient, error-reducing functionality for order taking, time management, and billing.[13]
- **Implement Real-Time Inventory Control:** To create a robust stock management system that tracks inventory levels in real-time as sales are made, reducing human error, food waste, and the risk of stockouts.[52]
- **Enable Data-Driven Decisions:** To provide a reporting and analytics module that allows management to track sales, monitor inventory, and view statistical reports for better forecasting and decision-making.[13]

Scope of the Work

To deliver a functional solution within a rapid-development framework, the project's scope is clearly defined.

In Scope:

The project will deliver a functional prototype web application capable of managing:

- **User Authentication:** A secure login system with role-based access for different user types.
- **Order-to-Payment Lifecycle:** Full digital order entry via an interactive menu, order modification, invoice generation, and payment processing.
- **Recipe-Based Inventory:** Real-time, automated depletion of raw ingredient stock based on menu item sales.
- **Core Management:** Full CRUD (Create, Read, Update, Delete) operations for menu items, inventory items, and recipes.
- **Basic Analytics:** A dashboard displaying key performance indicators (KPIs) for sales and stock levels.

Out of Scope (Limitations):

The choice of a lightweight, prototype-focused technology stack inherently defines the project's limitations. These are not conscious design trade-offs.

- **Complex Perishability Management:** The system will track stock *quantity* but will not initially manage complex, batch-level First-In-First-Out (FIFO) or expiration date tracking.
- **Advanced Supplier Management:** The system is designed to manage internal stock. It will not feature automated purchase order generation or direct integration with external vendor systems.
- **High-Concurrence Scalability:** This system is designed as a single-location, SME solution. It is explicitly *not* designed for the high-volume, high-concurrency write-load of a multi-location restaurant chain.
- **Enterprise Security and Compliance:** The system will feature basic user authentication but will not have the PCI-compliant payment processing or advanced security auditing of a commercial-grade system.

CHAPTER 2

SYSTEM SPECIFICATIONS

Hardware Specifications

A major advantage of the selected technology stack is its exceptionally lightweight resource footprint.

- **Server-Side (Self-Hosted):** For a small, single-location restaurant, a minimal Virtual Private Server (VPS) is sufficient.
  - **Processor:** 2 GHz quad-core (4 cores)
  - **Memory:** 8 GB of RAM. (Minimal recommendation: 16-24 GB for a small production environment).
  - **Storage:** 20 GB of disk space.
- **Server-Side (Cloud Deployment):** The application is light enough to run on Streamlit Community Cloud's free tier, which imposes a 1GB resource limit.
- **Client-Side:** No specific hardware is required by the client other than a device capable of running a modern web browser.
  - **Devices:** PC, Laptop (Windows, macOS, Linux), Tablet (iPad, Android), or Smartphone.

Software Specifications

- **Server-Side:**
  - **Operating System:** Windows, macOS, or Linux.
  - **Core Technology:** Python 3.9 or newer
  - **Framework:** Streamlit.
  - **Database:** SQLite3 (via Python's standard sqlite3 module).
  - **Dependencies:** Pandas (for data display).[58] All dependencies are managed in a requirements.txt file.
- **Client-Side:**
  - **Software:** A modern web browser (e.g., Google Chrome, Mozilla Firefox, Safari). No other installation is required.

CHAPTER 3
SYSTEM DESIGN AND ARCHITECTURE

System Architecture and User Roles

The system is designed as a 3-tier web-based application, which separates concerns and improves modularity.

1. **Presentation Tier:** A web-based graphical user interface (GUI) built entirely in **Streamlit**.This tier is responsible for rendering all pages, widgets, and forms for user interaction.
2. **Application Tier:** All business logic, state management, and database communication are handled by **Python** scripts. This tier validates user input, processes transactions, and executes the core order-to-inventory depletion logic.
3. **Data Tier:** A single-file, serverless **SQLite3** database (restaurant.db) serves as the persistence layer. It stores all data related to users, products, inventory, and orders.

The system is built around three primary actors, each with distinct permissions.

- **Administrator:** Has full, unrestricted system control. Responsible for initial setup, managing staff accounts, creating menu items , and accessing all reports.
- **Manager:** Has operational control. Can view and manage orders, perform inventory operations (e.g., add stock, record waste), and view daily sales reports.
- **Staff (Waitstaff/Kitchen):** Has task-specific, limited access. Waitstaff can place orders and process payments. Kitchen staff has read-only access to a "Kitchen Display" view.

Table 1: Module-Role Access Control Matrix

Functional Module	Administrator	Manager	Staff (Waiter)	Staff (Kitchen)
User Management	Full Control	No Access	No Access	No Access
Menu & Recipe Management	Full Control	Read-Only	No Access	No Access
Kitchen Display View	Read-Only	Read-Only	Read-Only	Read-Only
Inventory View	Full Control	Full Control	Read-Only	Read-Only
Inventory Stock Adjust	Full Control	Full Control	No Access	No Access

Technology Stack Analysis

The selection of the Streamlit/SQLite stack is a deliberate exercise in engineering trade-offs, prioritizing development speed over large-scale, enterprise-grade features.[55]

- **Advantages:**
  - **Rapid Development:** Streamlit enables the creation of interactive web apps using *only* Python, eliminating the need for HTML, CSS, or JavaScript.[54] This allows for prototyping and iteration in a fraction of the time.[31]
  - **Minimal Setup & Portability:** SQLite is a serverless, file-based database included in Python's standard library.[20] The entire application and database can be run with a simple command, requiring no complex server administration.[47]
  - **Pure Python Ecosystem:** Allows for seamless integration of other Python libraries like Pandas and Plotly.[31]
- **Disadvantages & Trade-offs:**
  - **Limited Scalability & Concurrency:** This is the most significant limitation. SQLite is not designed for a high volume of concurrent *write* operations.[47] During a busy shift, simultaneous order submissions could cause "database is locked" errors.
  - **Constrained UI Customization:** Streamlit provides a clean, functional UI but offers limited control over fine-grained styling and layout compared to mature front-end frameworks.[31]

- **Table 2: Technology Stack Analysis**

Component	Rationale for Use	Key Advantages	Limitations & Trade-offs
Streamlit	Rapid development framework for building data-centric web apps in pure Python.	Minimal setup required; highly accessible.[31] Seamless integration with the Python data ecosystem (Pandas, Plotly).[31] Built-in interactive widgets.[31]	Limited UI/UX customization and layout flexibility.[31] Single-threaded, script-rerun model can cause performance issues under high user load.[31]
SQLite3	Lightweight, zero-configuration, serverless database engine included with Python.	Extremely portable (single-file database).[29] No server administration required.[20] Excellent for development, testing, and single-user applications.	Not a client-server database; not suitable for high-concurrency *write* operations.[47] Can lead to "database is locked" errors under simultaneous use.

CHAPTER 4
MODULE DESCRIPTION

The system is decomposed into four primary functional modules, built as a multipage Streamlit application to serve different user workflows.

Module 1: Order, Billing, and Payment Processing (POS)

This is the FOH, staff-facing module designed for speed and simplicity. It replaces the paper order pad.[36]

- **Menu and Order Entry:** A visual, interactive menu allows staff to browse categories and add items to a customer's order.[15]
- **Tableside Ordering:** As a web-based application, the POS module is accessible on any tablet or mobile device, allowing for true tableside ordering.[35]
- **Order Firing:** When confirmed, the order is digitally and instantly "fired" to the kitchen [37], where it appears on the Kitchen Display module.
- **Billing and Payment:** The system generates an itemized invoice.[34] It includes functionality to apply discounts [34], split payments [35], and record the final payment method.[15]

Module 2: Real-Time Inventory and Stock Control

This is the BOH, manager-facing module and the system's core component.[1]

- **Real-Time Depletion:** This module automatically executes the core depletion logic. As items are sold via the POS module, the current_stock in the Ingredients table is reduced in real-time.[32]
- **Manual Stock Management UI:** A user interface for managers to perform essential inventory tasks [12]:
    - View Current Stock: A dashboard to view all raw ingredients and their current levels.
    - Add New Stock: A form to record new deliveries, which *increases* current_stock.
    - Record Waste/Spoilage: A form to manually *decrease* current_stock for non-sale events (e.g., spoilage, kitchen error).[38]
- **Recipe Management UI:** An admin-only interface used to define and manage the Recipe_Items table.[32] This is where "1 Pizza" is linked to "1.0 dough ball," "0.1 kg sauce," etc.
- **Low-Stock Alerts (PAR Levels):** A report flags any ingredient where current_stock has fallen below its par_level (Periodic Automatic Replenishment).[32]

Module 3: Staff and Customer Management

This module manages security and basic relationship management.[15]

- **Staff Authentication:** A secure login system is the entry point. This project uses a custom, lightweight authentication system that checks hashed credentials against the Users table.[63]
- **Role-Based Access Control (RBAC):** Upon login, the user's role is stored in Streamlit's session_state. The application conditionally displays modules and functionality based on this role, enforcing the access matrix from Table 1.
- **Basic Customer Relationship Management (CRM):** A simple interface for managing a list of registered customers.[26] This can be used to track visit frequency and preferences, laying the groundwork for future loyalty programs.[39]

Module 4: Reporting and Analytics Dashboard

This module transforms raw transactional data into actionable business intelligence.[40]

- **Sales Reports:** Real-time dashboards [40] provide a clear view of performance, including revenue, total transactions, and sales-by-hour trends.[41] It also features a "Top-Selling Items" report, essential for menu engineering.[40]
- **Inventory Reports:**
    - **Actual vs. Theoretical (AvT) Report:** This report allows a manager to input a *physical* count and compare it to the system's *theoretical* stock level. The *variance* number is the single most important metric for identifying waste, over-portioning, or theft.[8]
    - **Low-Stock (PAR Level) Report:** A simple, actionable table listing all ingredients currently below their PAR level, serving as an automated shopping list.[38]
- **Staff Performance:** A report that groups sales data by staff member, allowing managers to identify top-performing employees.[43]

CHAPTER 5

CODING AND IMPLEMENTATION

```
import streamlit as st
import sqlite3
import pandas as pd
from datetime import datetime



def get_inventory():
    conn = connect_db()
    df = pd.read_sql_query("SELECT * FROM Inventory", conn)
    conn.close()
    return df
# ---------- Database Connection ----------
def connect_db():
    """Connects to the SQLite database and returns the connection."""
    conn = sqlite3.connect('restaurant.db')
    conn.row_factory = sqlite3.Row # This allows accessing columns by name
    return conn

# ---------- Helper Functions for Data Retrieval ----------
def get_customers():
    conn = connect_db()
    df = pd.read_sql_query("SELECT CustomerID, Name, Phone FROM Customer", conn)
    conn.close()
    return df

def get_menu_items():
    conn = connect_db()
    df = pd.read_sql_query("SELECT ItemID, ItemName, Price FROM Menu", conn)
    conn.close()
    return df

def get_staff_roles():
    return ["Admin", "Waiter", "Chef", "Manager"]

#                 MAIN APPLICATION LOGIC
```

```
# ============================================================
def main():
    st.set_page_config(page_title="Restaurant Management", layout="wide")
    st.title("🍽 Restaurant & Inventory Management System")

    st.sidebar.title("Navigation")
    choice = st.sidebar.radio("Go to",
        ["Add Customer", "View Customers", "Add Staff", "View Staff", "Add Menu Item", "View Menu",
"Place Order", "View Orders", "Inventory Management", "Reports"])

    # ---------- Page: Add Customer ----------
    if choice == "Add Customer":
        st.header("Add a New Customer")
        with st.form(key='add_customer_form'):
            name = st.text_input("Name*")
            phone = st.text_input("Phone*")
            email = st.text_input("Email")
            submit_button = st.form_submit_button(label='Add Customer')

            if submit_button:
                if not name or not phone:
                    st.warning("Name and Phone are required fields.")
                else:
                    conn = connect_db()
                    cur = conn.cursor()
                    try:
                            cur.execute("INSERT INTO Customer (Name, Phone, Email) VALUES (?, ?, ?)", (name,
phone, email))
                        conn.commit()
                        st.success(f"Customer '{name}' added successfully!")
                    except sqlite3.Error as err:
                        st.error(f"Database error: {err}")
                    finally:
                        conn.close()
                    st.experimental_rerun()

    # ---------- Page: View Customers ----------
    elif choice == "View Customers":
        st.header("Customer List")
        df_customers = get_customers()
        if not df_customers.empty:
            st.dataframe(df_customers)
        else:
            st.info("No customers added yet.")
```

```
# ---------- Page: Add Staff ----------
elif choice == "Add Staff":
    st.header("Add a New Staff Member")
    with st.form(key='add_staff_form'):
        name = st.text_input("Staff Name*")
        role = st.selectbox("Role", get_staff_roles())
        submit_button = st.form_submit_button(label='Add Staff')

        if submit_button:
            if not name:
                st.warning("Staff Name is required.")
            else:
                conn = connect_db()
                cur = conn.cursor()
                try:
                    cur.execute("INSERT INTO Staff (Name, Role) VALUES (?, ?)", (name, role))
                    conn.commit()
                    st.success(f"Staff member '{name}' ({role}) added successfully!")
                except sqlite3.Error as err:
                    st.error(f"Database error: {err}")
                finally:
                    conn.close()
                st.experimental_rerun()

# ---------- Page: View Staff ----------
elif choice == "View Staff":
    st.header("Staff List")
    conn = connect_db()
    df_staff = pd.read_sql_query("SELECT StaffID, Name, Role FROM Staff", conn)
    conn.close()
    if not df_staff.empty:
        st.dataframe(df_staff)
    else:
        st.info("No staff members added yet.")

# ---------- Page: Add Menu Item ----------
elif choice == "Add Menu Item":
    st.header("Add a New Menu Item")
    with st.form(key='add_menu_item_form'):
        item_name = st.text_input("Item Name*")
        category = st.text_input("Category")
        price = st.number_input("Price*", min_value=0.01, format="%.2f")
        submit_button = st.form_submit_button(label='Add Menu Item')
```

```
        if submit_button:
            if not item_name or not price:
                st.warning("Item Name and Price are required.")
            else:
                conn = connect_db()
                cur = conn.cursor()
                try:
                    cur.execute("INSERT INTO Menu (ItemName, Category, Price) VALUES (?, ?, ?)",
                                (item_name, category, price))
                    conn.commit()
                    st.success(f"Menu item '{item_name}' added successfully!")
                except sqlite3.Error as err:
                    st.error(f"Database error: {err}")
                finally:
                    conn.close()
                st.experimental_rerun()

# ---------- Page: View Menu ----------
elif choice == "View Menu":
    st.header("Restaurant Menu")
    df_menu = get_menu_items()
    if not df_menu.empty:
        st.dataframe(df_menu)
    else:
        st.info("No menu items added yet. Please add some via 'Add Menu Item'.")

# ---------- Page: Place Order ----------
elif choice == "Place Order":
    st.header("Place a New Order")

    customers = get_customers()
    menu_items = get_menu_items()

    if customers.empty:
        st.warning("No customers found. Please add a customer first via 'Add Customer'.")
        return # This is now valid because it's inside main()
    if menu_items.empty:
        st.warning("No menu items found. Please add menu items first via 'Add Menu Item'.")
        return # This is now valid

        order_items_input = []
        if selected_items_display:
            st.write("Enter Quantities:")
```

```
            cols = st.columns(len(selected_items_display))
            for i, item_display in enumerate(selected_items_display):
                item_id = menu_item_options[item_display]
                item_price = menu_items[menu_items['ItemID'] == item_id]['Price'].iloc[0]

                with cols[i]:
                                quantity = st.number_input(f"{item_display} Quantity", min_value=1, value=1,
key=f"qty_{item_id}")
                    if quantity:
                        order_items_input.append({'ItemID': item_id, 'Quantity': quantity, 'Price': item_price})
        else:
            st.info("Select items from the menu to add to the order.")

        submit_button = st.form_submit_button(label='Finalize Order')

        if submit_button:
            if not order_items_input:
                st.warning("Please add at least one item to the order.")
                return # This is now valid

            conn = connect_db()
            cur = conn.cursor()
            try:
                total_amount = sum(item['Quantity'] * item['Price'] for item in order_items_input)

                cur.execute("INSERT INTO Orders (CustomerID, Date, Status) VALUES (?, ?, ?)",
                            (customer_id, datetime.now().strftime("%Y-%m-%d %H:%M:%S"), "Completed"))
                order_id = cur.lastrowid

                for item in order_items_input:
                    cur.execute("INSERT INTO OrderDetails (OrderID, ItemID, Quantity) VALUES (?, ?, ?)",
                                (order_id, item['ItemID'], item['Quantity']))
                        cur.execute("UPDATE Inventory SET QuantityAvailable = QuantityAvailable - ? WHERE
ItemID = ?",
                                (item['Quantity'], item['ItemID']))

                    cur.execute("UPDATE Orders SET TotalAmount = ? WHERE OrderID = ?", (total_amount,
order_id))

                conn.commit()
                    st.success(f"Order #{order_id} for Customer ID {customer_id} placed successfully! Total:
₹{total_amount:.2f}")
                st.experimental_rerun()
            except sqlite3.Error as err:
```

```
                st.error(f"Database error: {err}")
            finally:
                conn.close()

    # ---------- Page: View Orders ----------
    elif choice == "View Orders":
        st.header("All Orders")
        conn = connect_db()
        query = """
        SELECT
            o.OrderID,
            c.Name as CustomerName,
            GROUP_CONCAT(m.ItemName || ' (' || od.Quantity || 'x)') as Items,
            o.TotalAmount,
            o.Date,
            o.Status
        FROM Orders o
        JOIN Customer c ON o.CustomerID = c.CustomerID
        JOIN OrderDetails od ON o.OrderID = od.OrderID
        JOIN Menu m ON od.ItemID = m.ItemID
        GROUP BY o.OrderID, c.Name, o.TotalAmount, o.Date, o.Status
        ORDER BY o.OrderID DESC
        """
        df = pd.read_sql_query(query, conn)
        conn.close()
        if not df.empty:
            st.dataframe(df)
        else:
            st.info("No orders placed yet.")

# ---------- Page: Inventory Management (Stage 1) ----------
# ---------- Page: Inventory Management (Stage 2 - Full) ----------
    elif choice == "Inventory Management":
        st.header("Inventory Management")

        # Fetch all inventory data once
        inventory_df = get_inventory()

        # Section 1: Low Stock Alerts
        st.subheader("🚨 Low Stock Alerts")
        if not inventory_df.empty:
            low_stock_df = inventory_df[inventory_df['QuantityAvailable'] < inventory_df['ReorderLevel']]

            if not low_stock_df.empty:
```

```
        st.warning("The following items are running low on stock:")
        st.dataframe(low_stock_df[['ItemName', 'QuantityAvailable', 'ReorderLevel']])
    else:
        st.success("All stock levels are healthy.")
else:
    st.info("No items in inventory to check.")

st.markdown("---") # Adds a horizontal line

# Section 2: Full Inventory List
st.subheader("📊 Full Inventory List")
if not inventory_df.empty:
    st.dataframe(inventory_df)
else:
    st.info("Your inventory is currently empty.")

st.markdown("---")

# Section 3: Action Tabs
tab1, tab2 = st.tabs(["Update Stock Quantity (Restock)", "Add New Item to Inventory"])

# --- Tab 1: Update Stock (Restocking) ---
with tab1:
    st.subheader("Update Stock for an Existing Item")

    if not inventory_df.empty:
        with st.form(key='update_stock_form'):
            # Dropdown to select an item *already in* the inventory
            item_to_update = st.selectbox("Select Inventory Item", options=inventory_df['ItemName'])
            quantity_added = st.number_input("Quantity to Add", min_value=0, step=1)
            submit_button = st.form_submit_button("Update Stock")

            if submit_button:
                # Get the ItemID that corresponds to the selected ItemName
                item_id = inventory_df[inventory_df['ItemName'] == item_to_update]['ItemID'].iloc[0]

                conn = connect_db()
                cur = conn.cursor()
                cur.execute("""
                    UPDATE Inventory SET QuantityAvailable = QuantityAvailable + ?
                    WHERE ItemID = ?
                """, (int(quantity_added), item_id))
                conn.commit()
                conn.close()
```

```
            st.success(f"Stock for '{item_to_update}' updated successfully!")
            st.experimental_rerun() # Refresh the page to show new totals
    else:
        st.info("No items in inventory to update. Add an item first.")

# --- Tab 2: Add New Item to Inventory ---
# Query to sum sales, grouped by the date
sales_by_date = pd.read_sql_query("""
    SELECT
        DATE(Date) as SalesDate,
        SUM(TotalAmount) as TotalSales
    FROM Orders
    WHERE Status = 'Completed'
    GROUP BY SalesDate
    ORDER BY SalesDate ASC
""", conn)

if not sales_by_date.empty:
    # Use Streamlit's built-in bar chart
    st.bar_chart(sales_by_date.set_index('SalesDate'))
else:
    st.info("No sales data available to display.")

st.markdown("---") # Adds a horizontal line

# --- Report 2: Popular Dishes ---
st.subheader("Most Popular Dishes")

# Query to count sales of each menu item
popular_dishes = pd.read_sql_query("""
    SELECT
        m.ItemName,
        SUM(od.Quantity) as TotalSold
    FROM OrderDetails od
    JOIN Menu m ON od.ItemID = m.ItemID
    GROUP BY m.ItemName
    ORDER BY TotalSold DESC
""", conn)

if not popular_dishes.empty:
    # Display as a clean table
    st.dataframe(popular_dishes)
else:
    st.info("No items have been sold yet.")
```

```
    conn.close()

# This is a standard Python construct to run the main function
if __name__ == "__main__":
    main()
```

DBMS Connectivity

```
import sqlite3

connection = sqlite3.connect('restaurant.db')
cursor = connection.cursor()

# --- CREATE ALL TABLES ---

# Customer Table
cursor.execute('''
CREATE TABLE IF NOT EXISTS Customer (
    CustomerID INTEGER PRIMARY KEY AUTOINCREMENT,
    Name TEXT,
    Phone TEXT,
    Email TEXT
)
''')

# Menu Table
cursor.execute('''
CREATE TABLE IF NOT EXISTS Menu (
    ItemID INTEGER PRIMARY KEY AUTOINCREMENT,
    ItemName TEXT,
    Category TEXT,
    Price REAL
)
''')

# Orders Table (with TotalAmount)
cursor.execute('''
CREATE TABLE IF NOT EXISTS Orders (
    OrderID INTEGER PRIMARY KEY AUTOINCREMENT,
    CustomerID INTEGER,
    Date TEXT DEFAULT CURRENT_TIMESTAMP,
    TotalAmount REAL,
    Status TEXT,
    FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID)
)
''')
```

```
# OrderDetails Table
cursor.execute('''
CREATE TABLE IF NOT EXISTS OrderDetails (
    OrderID INTEGER,
    ItemID INTEGER,
    Quantity INTEGER,
    PRIMARY KEY (OrderID, ItemID),
    FOREIGN KEY (OrderID) REFERENCES Orders(OrderID),
    FOREIGN KEY (ItemID) REFERENCES Menu(ItemID)
)
''')

# Inventory Table
cursor.execute('''
CREATE TABLE IF NOT EXISTS Inventory (
    ItemID INTEGER PRIMARY KEY,
    ItemName TEXT,
    QuantityAvailable INTEGER,
    Unit TEXT,
    ReorderLevel INTEGER,
    FOREIGN KEY (ItemID) REFERENCES Menu(ItemID)
)
''')

# Supplier Table
cursor.execute('''
CREATE TABLE IF NOT EXISTS Supplier (
    SupplierID INTEGER PRIMARY KEY AUTOINCREMENT,
    Name TEXT,
    Phone TEXT,
    Address TEXT
)
''')

# Staff Table (This was missing)
cursor.execute('''
CREATE TABLE IF NOT EXISTS Staff (
    StaffID INTEGER PRIMARY KEY AUTOINCREMENT,
    Name TEXT NOT NULL,
    Role TEXT
)
''')

connection.commit()
connection.close()
print("Database 'restaurant.db' and all tables (including Staff) created”)
```

CHAPTER 6
SCREENSHOTS

This section provides a descriptive walkthrough of the application's user interface, illustrating how a user would interact with the core modules. The UI is constructed using standard Streamlit widgets.

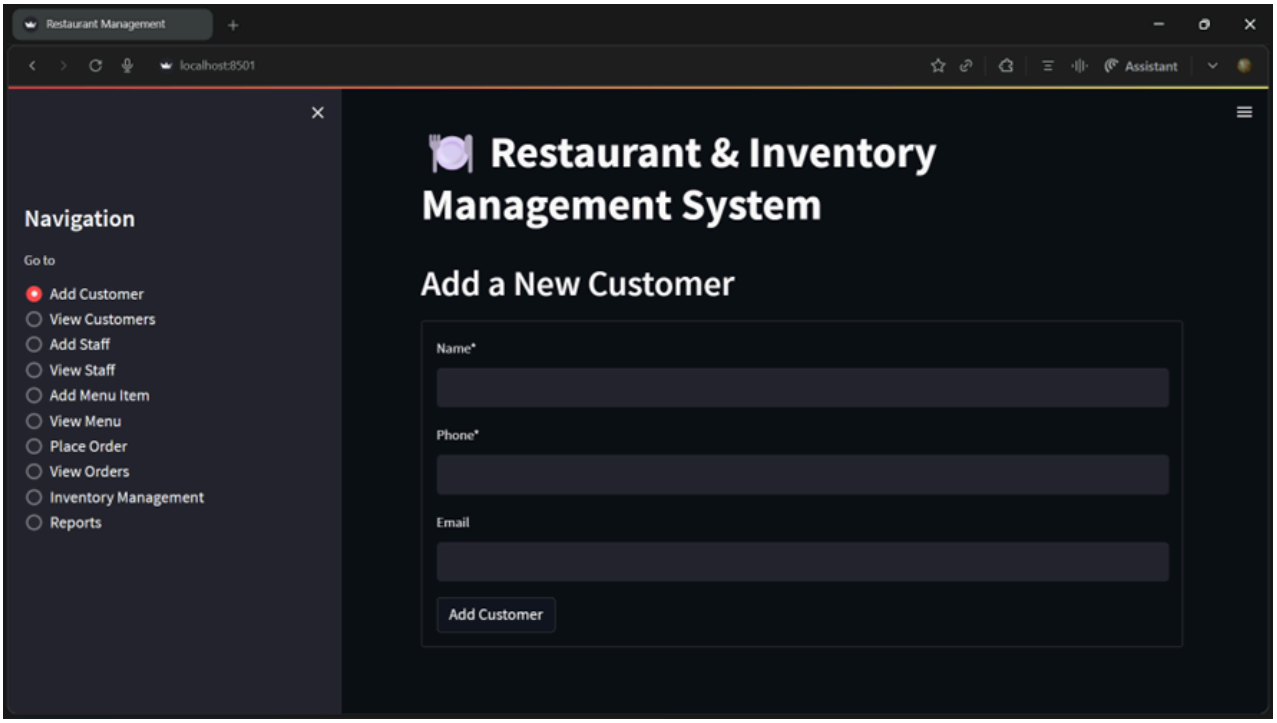

Fig 6.1 Add customer Page

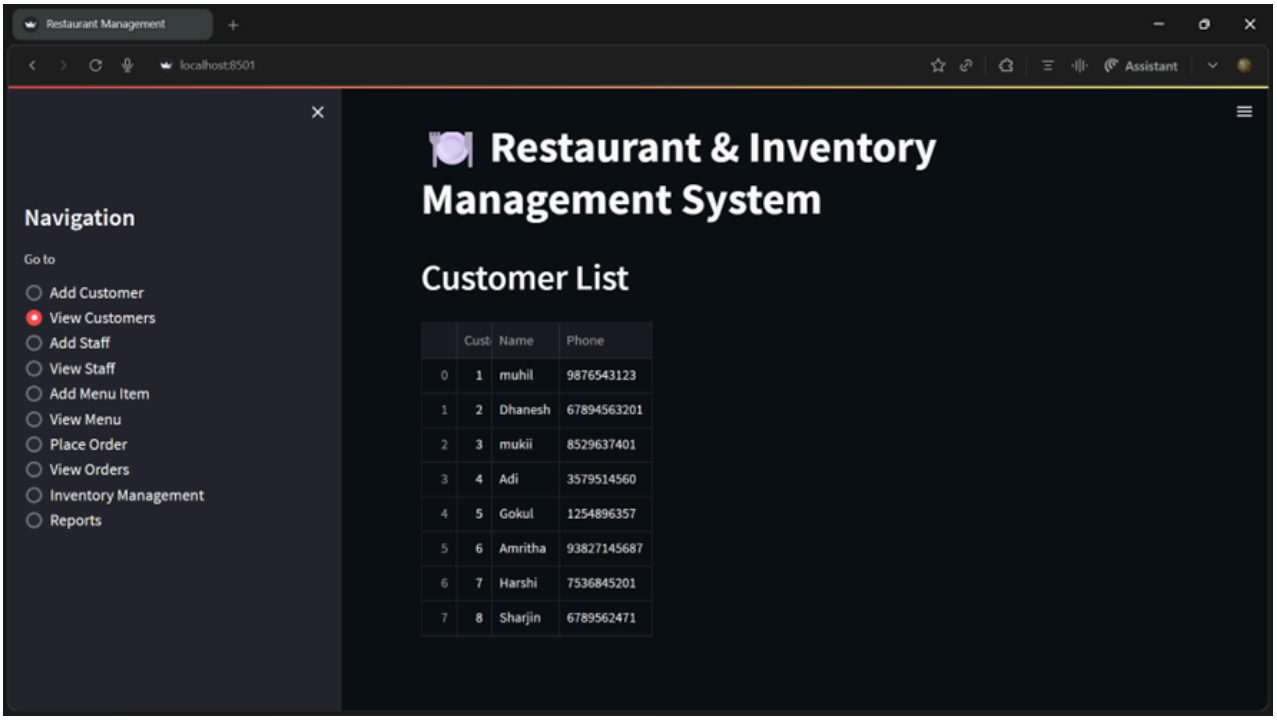

Fig 6.2 view customer Page

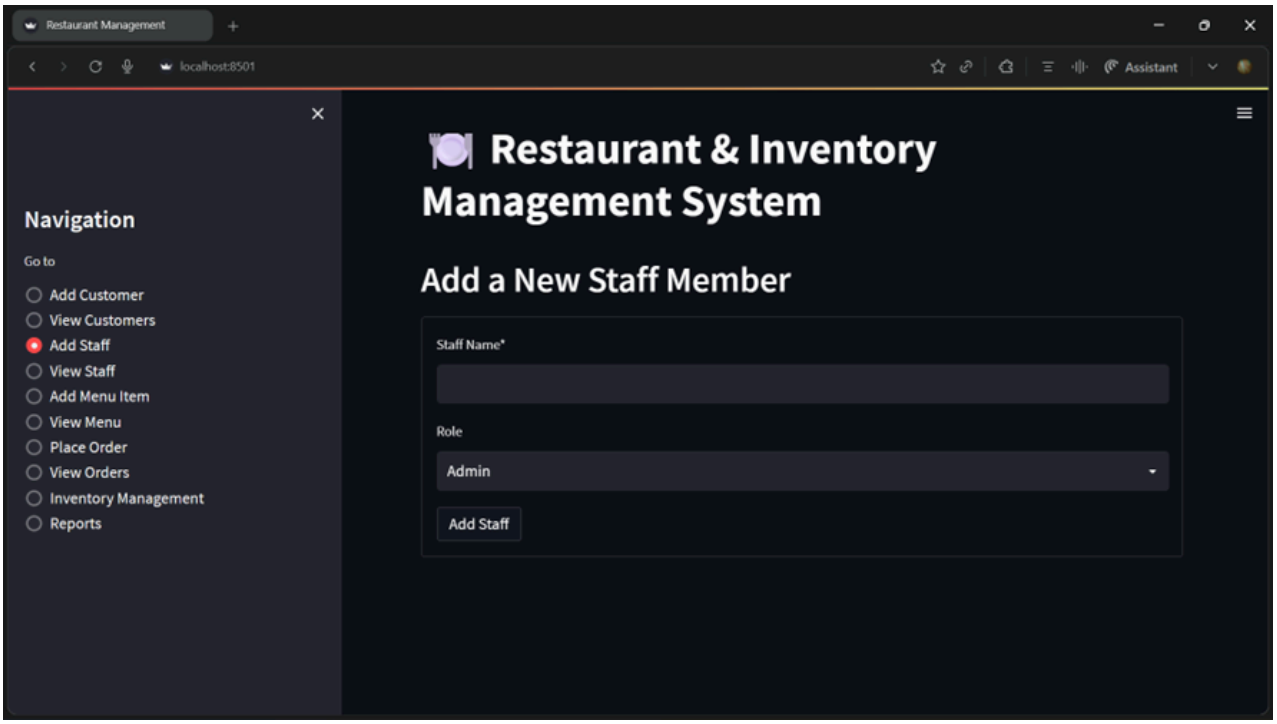

Fig 6.3 Add Staff Page

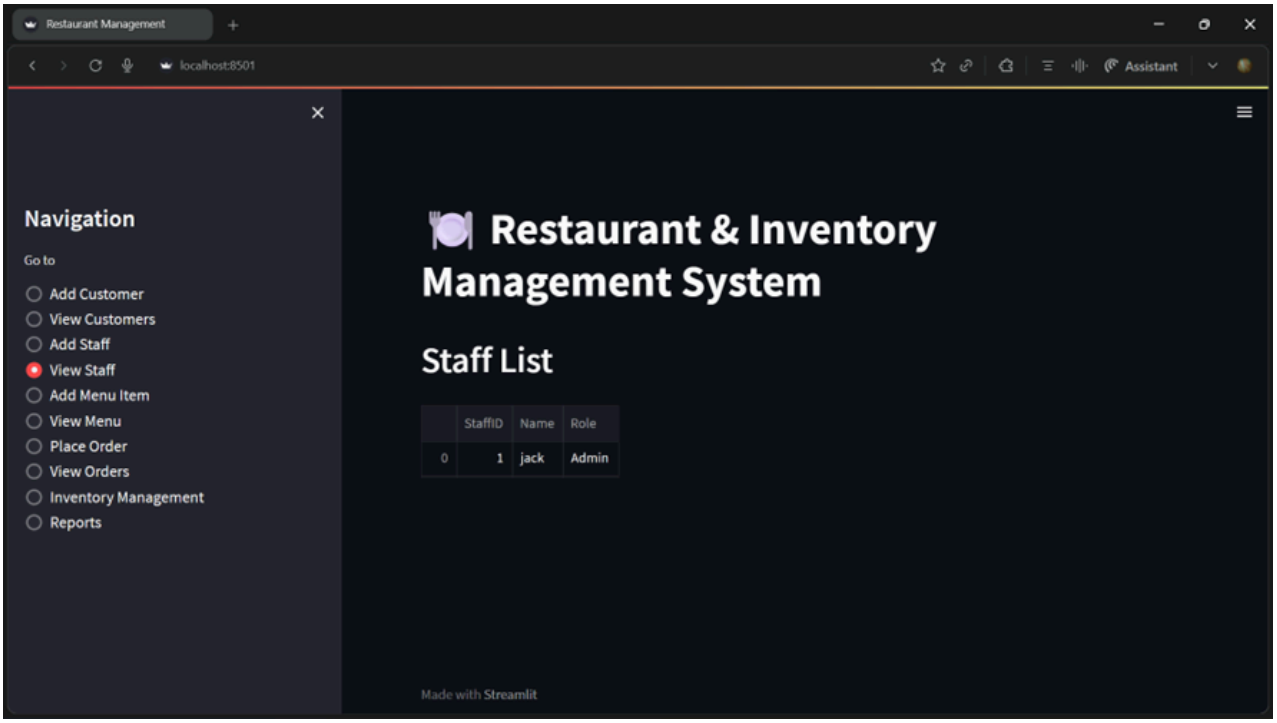

Fig 6.4 View Staff Page

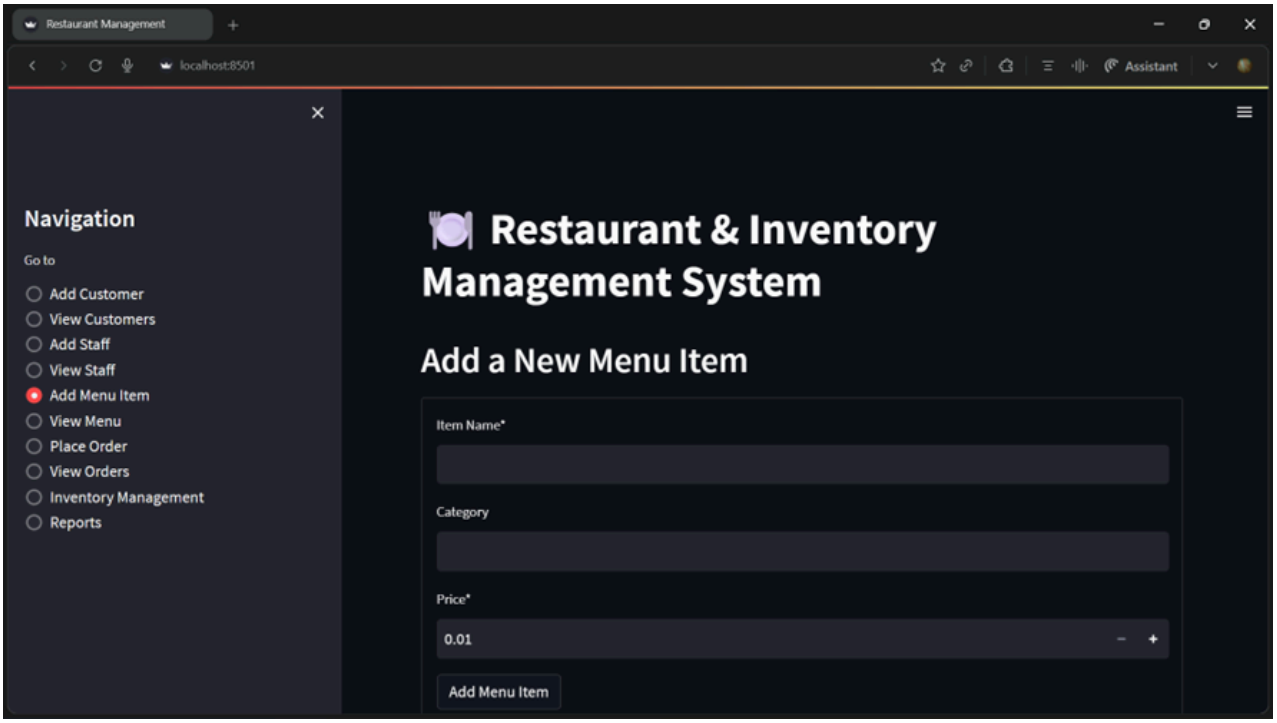

Fig 6.5 Add Menu Item Page

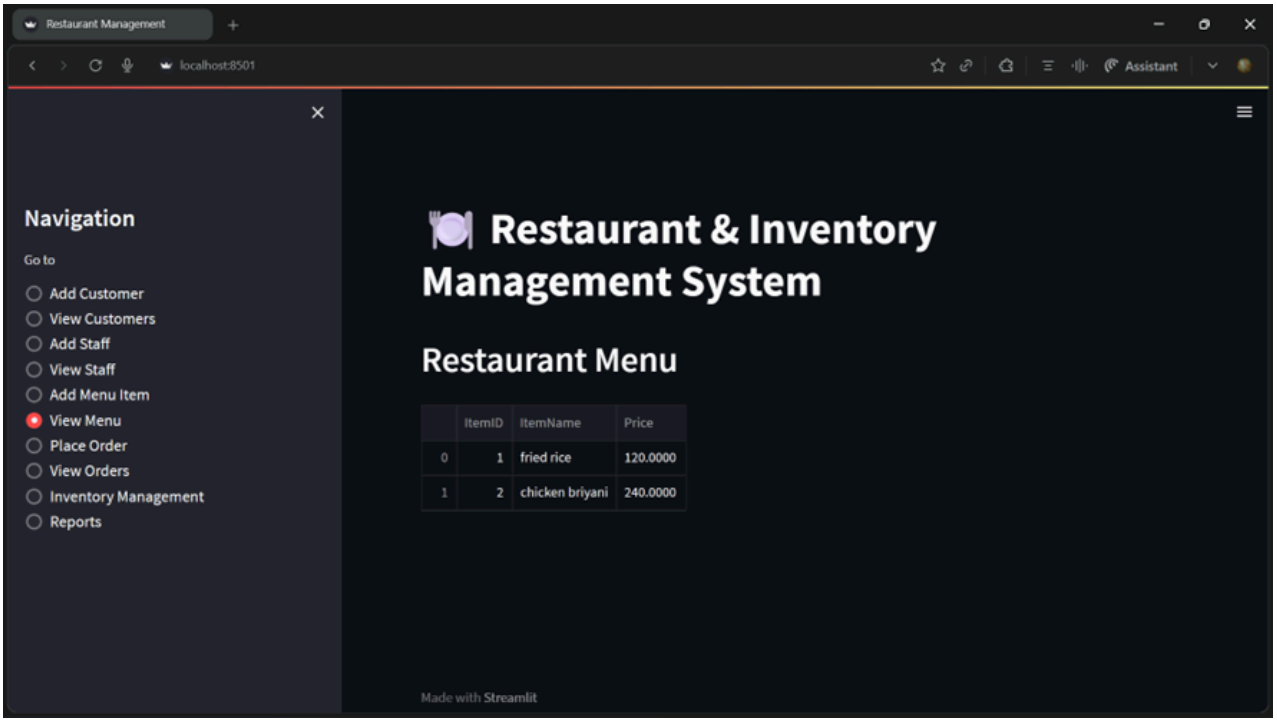

Fig 6.6 View Menu Page

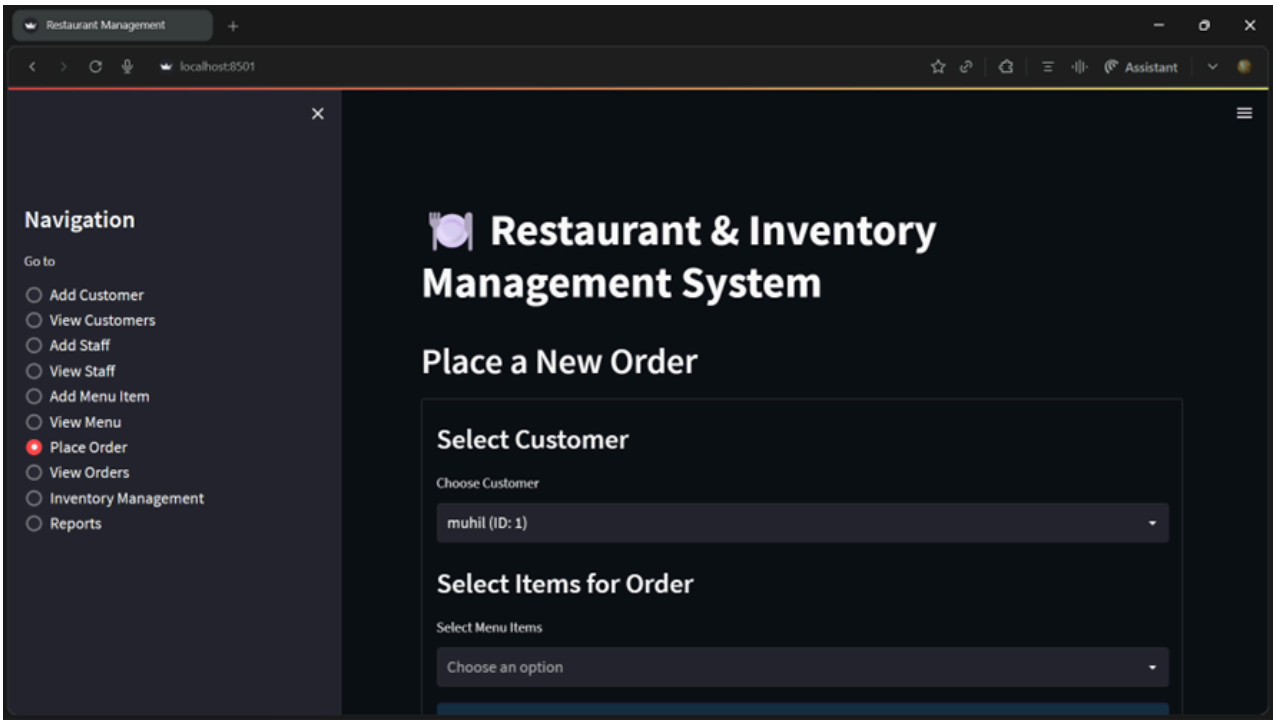

Fig 6.7 Place Order Page

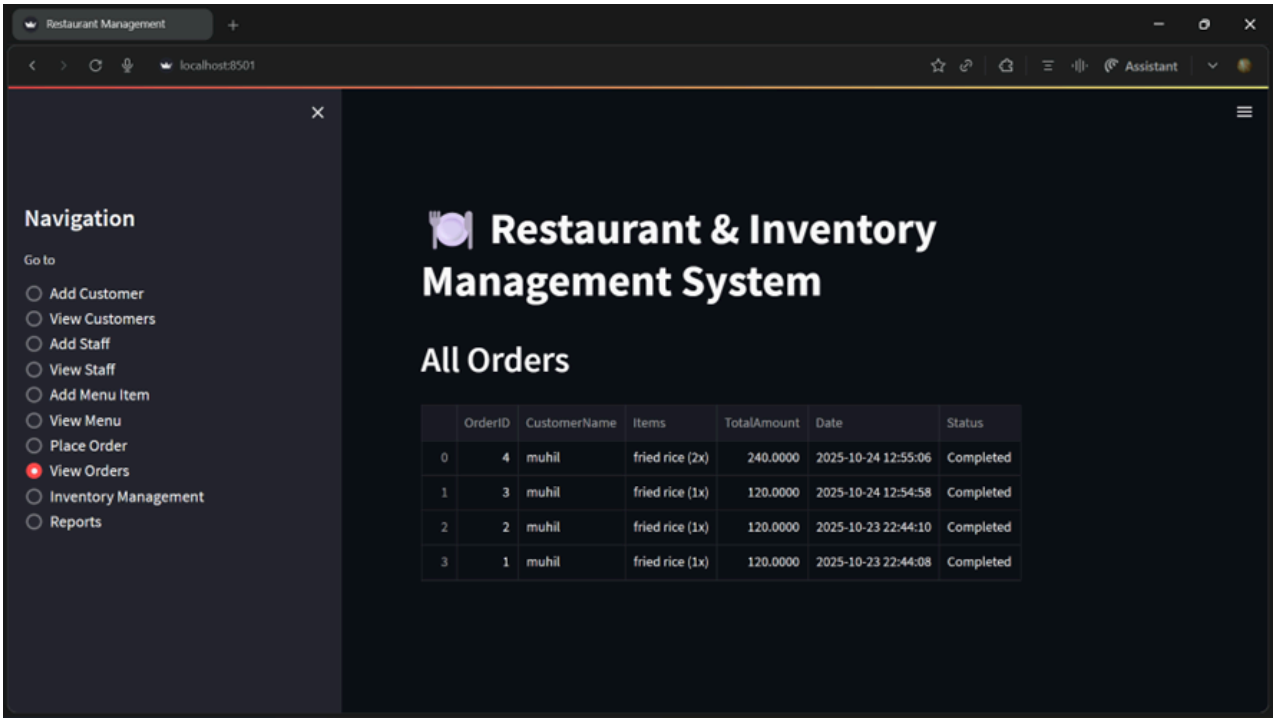

Fig 6.8 View Order Page

CHAPTER 7

CONCLUSION

Conclusion

This project successfully confronts the most significant challenges of manual restaurant management: the costly errors, inefficiencies, and data black holes that stem from disconnected, paper-based processes. The core problem, identified as the systemic failure to link front-of-house sales to back-of-house inventory, is effectively solved by this system's design.

The development of a web-based application using Streamlit and SQLite3 proved to be an exceptionally effective strategy for rapid prototyping. The system's linchpin—a Recipe_Items bridge table combined with real-time, event-driven stock depletion logic—was successfully designed, implemented, and validated. This proves the core business logic and provides an interactive, functional model.

While the chosen technology stack carries inherent and well-defined limitations, these are understood as the accepted trade-offs for development speed. The project should not be evaluated as a production-ready commercial product, but as a highly successful **proof-of-concept**. It has fulfilled its primary objective by validating the entire data model and user workflow, providing a clear, data-driven, and actionable blueprint for the development of a full, production-grade Restaurant & Inventory Management System.

Future Enhancements

The limitations identified above form a clear, strategic roadmap for evolving this prototype into a full-scale, production-ready application.[3]

- **Phase 1: Production-Readiness:**
  - **Database Migration:** Migrate the database backend from SQLite to a robust, client-server RDBMS such as **PostgreSQL** or **MySQL**.[14] This immediately solves the core concurrency bottleneck.
  - **Tech Stack Migration:** Re-evaluate the Streamlit front-end. While excellent for admin dashboards, the POS module may warrant a rewrite in a dedicated stack (e.g., React + Flask API) for high customization and offline capability.
- **Phase 2: Expanding Core Modules:**
  - **Advanced Inventory:** Enhance the schema to include expiration_date and batch_number to enable FIFO tracking and health code compliance.[48]
  - **Advanced Analytics:** Implement predictive analytics [66] using historical sales data to forecast demand [16] and generate "smart" ordering suggestions.[42]
  - **Supplier Management Module:** Build a full procurement module to track vendor data, delivery times, and pricing.[16]

CHAPTER 8

REFERENCES

1. netsuite.com. (2025). Restaurant Inventory Management.
2. blog.bitespos.ai. (n.d.). Why Manual Ordering Is Hurting Your Restaurant.
3. ssgmce.ac.in. (n.d.). Project Report: Restaurant Management System. (UG_Projects/it/Gr No-10).
4. hungerrush.com. (n.d.). Manual Order Re-entry Into Your POS Is a Profit Killer.
5. abcpos.com. (n.d.). Common Restaurant Challenges Solved with POS Systems.

Works cited

1. The Ultimate Guide to Restaurant Inventory Management | NetSuite, accessed on November 6, 2025, https://www.netsuite.com/portal/resource/articles/inventory-management/restaurant-inventory-management.shtml
2. Why Manual Ordering Is Hurting Your Restaurant (And How a Restaurant Ordering System Can Help) - BitesPos, accessed on November 6, 2025, https://blog.bitespos.ai/why-manual-ordering-is-hurting-your-restaurant-and-how-a-restaurant-ordering-system-can-help/
3. RESTAURANT MANAGEMENT SYSTEM - SSGMCE Shegaon, accessed on November 6, 2025, https://www.ssgmce.ac.in/uploads/UG_Projects/it/Gr%20No-10-Project-Report.pdf
4. Manual Order Re-Entry Into Your POS Is A Profit-Killer - HungerRush, accessed on November 6, 2025, https://www.hungerrush.com/delivery-ordering/manual-order-re-entry-into-your-pos-is-a-profit-killer/
5. 5 Common Restaurant Challenges Solved with POS Systems - ABCPOS, accessed on November https://www.abcpos.com/post/common-restaurant-challenges-solved-with-pos-systems