



# **RAJALAKSHMI ENGINEERING COLLEGE**

**An AUTONOMOUS Institution  
Affiliated to ANNA UNIVERSITY, Chennai**

## **RESTAURANT MANAGEMENT SYSTEM**

### **A MINI PROJECT REPORT**

**Submitted by**

**PHOOJITHA J                      231501126**

**JASHWANTH M                    231501506**

**IRSHAN M                         231501505**

**In partial fulfillment for the award of the degree of**

**BACHELOR OF ENGINEERING**

**IN**

**ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING**

**RAJALAKSHMI ENGINEERING COLLEGE (AUTONOMOUS)**

**THANDALAM**

**CHENNAI-602105**

**2024 - 2025**



## **BONAFIDE CERTIFICATE**

Certified that this project report “**RESTAURANT MANAGEMENT SYSTEM**” is the bonafide work of “**JASHWANTH M (231501506),PHOOJITHAA J (231501116), IRSHAN (231501505)**”

who carried out the project work under my supervision.

**Submitted for the Practical Examination held on \_\_\_\_\_**

### **SIGNATURE**

**Mr. U. Kumaran,**  
**Assistant Professor (SS)**  
**AIML,**  
**Rajalakshmi Engineering College,**  
**(Autonomous),**  
**Thandalam, Chennai - 602 105**

**INTERNAL EXAMINER**

**EXTERNAL EXAMINER**

## **ABSTRACT**

The Restaurant Management System (RMS) is a database-driven application designed to streamline the operations of a restaurant, ensuring efficient management of orders, inventory, billing, and employee records. By utilizing a Database Management System (DBMS), the system offers a centralized platform for managing and tracking key aspects of restaurant operations in real-time. This system enables staff to take customer orders, manage menu items, generate bills, and track inventory, all while reducing manual errors and improving operational efficiency. The system also includes features for user authentication, inventory management, and reporting, offering insights into sales trends and employee performance. The primary objective of the RMS is to automate and simplify restaurant operations, providing a seamless experience for both customers and restaurant staff, while contributing to enhanced productivity and better decision-making.

# **TABLE OF CONTENTS**

## **1.INTRODUCTION**

1.1 INTRODUCTION

1.1 OBJECTIVES

1.2 MODULES

## **2.SURVEY OF TECHNOLOGIES**

**SOFTWARE DESCRIPTION**

**LANGUAGES**

2.2.1 SQLITE 3

2.2.2 PYTHON

2.2.3 TIKINTE

## **3. REQUIREMENTS AND ANALYSIS**

3.1 REQUIREMENT SPECIFICATION

3.2 HARDWARE AND SOFTWARE REQUIREMENTS

3.3 ARCHITECTURE DIAGRAM

3.4 ER DIAGRAM

## **4. PROGRAM CODE**

## **5.PROJECT SCREENSHOTS**

## **6. RESULTS AND DISCUSSION**

## **7. CONCLUSION**

## **8. REFERENCES**

## 1. INTRODUCTION

In the fast-paced and competitive restaurant industry, efficient management of daily operations is critical to ensuring high-quality service and business success. Traditionally, restaurant operations such as order taking, billing, inventory tracking, and employee management have been handled manually, leading to inefficiencies, errors, and delays. With the advent of modern technology, it has become increasingly important for restaurants to adopt automated systems that streamline these processes.

The **Restaurant Management System (RMS)** is a software solution designed to automate and optimize the key functions of a restaurant. Built on a Database Management System (DBMS), this system allows for centralized management of orders, inventory, billing, and employee records. It is aimed at improving the overall efficiency of restaurant operations by reducing human error, speeding up the order-to-billing process, and providing a better customer experience.

The RMS is composed of several modules, each addressing a critical aspect of restaurant management. These include menu management, order processing, inventory tracking, employee scheduling, and reporting tools. By using a relational database, the system ensures that all data is easily stored, accessed, and updated, allowing restaurant staff to make informed decisions in real-time.

This system offers both administrative and staff interfaces, enabling users to manage restaurant operations according to their roles. For instance, restaurant managers can oversee inventory levels, generate sales reports, and update the menu, while waitstaff can take orders and print bills. By automating these tasks, the RMS not only reduces the likelihood of errors but also improves the speed and accuracy of service, ultimately leading to a more efficient and profitable restaurant operation.

## **1.1 OBJECTIVES**

### **1. Streamlining Operations**

The RMS aims to simplify restaurant operations by integrating key functionalities into a single platform. This includes efficient management of customer data, orders, menu items, and table reservations.

### **2. Efficient Billing and Payments**

The system will automate billing processes, handle various payment methods, and securely manage transaction records, ensuring accuracy and speed during checkout.

### **3. Employee Management**

The system will record employee details, track attendance, and manage payroll, supporting effective human resource management and role-based assignments.

### **4. Real-Time Data Handling**

With real-time updates, RMS ensures that changes in orders, inventory, and billing reflect immediately, enhancing responsiveness in a dynamic restaurant environment.

### **5. Sales Reporting**

RMS will generate detailed sales reports to assist managers in understanding sales trends, tracking performance, and making data-driven decisions to increase profitability.

### **6. Enhanced Customer Experience**

By automating operations and reducing wait times, RMS aims to improve service quality, increasing customer satisfaction and loyalty.

## **1.2 MODULES**

### **1. Customer Management Module**

- Purpose: To manage customer information, including profiles, order history, and preferences.
- Features:
  - Customer registration and login.
  - Track customer contact details, order history, and loyalty points.
  - Manage reservations and feedback.

### **2. Menu Management Module**

- Purpose: To manage restaurant menu items, including adding, updating, or removing dishes.
- Features:
  - Categorize menu items (e.g., starters, main course, desserts).
  - Set prices, availability, and descriptions for each item.
  - Add images or nutritional information for dishes.

### **3. Order Management Module**

- Purpose: To handle customer orders from creation to completion.
- Features:
  - Place new orders, modify existing ones.
  - Track order status (e.g., pending, in preparation, served).
  - Generate order summaries and receipts.

### **4. Reservation Management Module**

- Purpose: To manage table reservations and optimize seating arrangements.
- Features:
  - Allow customers to reserve tables by date and time.
  - Check table availability and allocate seating.
  - Record special requests (e.g., smoking/non-smoking areas, seating preferences).

### **3.SURVEY OF TECHNOLOGIES**

#### **1. SOFTWARE DESCRIPTION**

##### **VISUAL STUDIO CODE**

Visual Studio Code, commonly referred to as VS Code, is a powerful, open-source code editor developed by Microsoft. Designed with flexibility and productivity in mind, VS Code combines the simplicity of a text editor with advanced developer tooling, making it suitable for a wide range of programming tasks across different languages and frameworks. It has gained immense popularity in the developer community due to its user-friendly interface, extensibility, and efficient features that streamline the development process.

### **LANGUAGES**

#### **2.2.1 SQLITE 3**

SQLite 3 will serve as the embedded database engine for the Restaurant Management System. It will store and manage all critical data related to restaurant operations, including customer profiles, orders, reservations, employees, and menu items. The use of SQLite makes the system lightweight and easy to deploy, with the database engine being fully embedded in the application.

##### **Database Design**

The **RMS database** will consist of several tables representing different aspects of restaurant operations. Below is an outline of the core tables in the system. **Payments**, these will not be part of the database schema.

##### **Tables Overview**

- **Customers Table:** Stores customer details such as name, contact information, and loyalty points.
- **Menu Table:** Stores details about each menu item, including the name, description, price, and availability.
- **Orders Table:** Manages customer orders, linking them to specific customers and tracking the status of each order.



- **OrderDetails Table:** Represents the many-to-many relationship between orders and menu items.
- **Reservations Table:** Manages table reservations for customers, including reservation time, date, and status.
- **Employees Table:** Stores information about restaurant staff, including roles and schedules.

### Relationships Between Tables

- **Customers** can have multiple **Orders** and **Reservations**.
- Each **Order** can contain multiple **Menu Items**, managed through the **OrderDetails** table.
- **Employees** are linked to specific roles and schedules but do not directly relate to other modules in this schema.

## 2.2.2 PYTHON

Python is an excellent choice for developing a **Restaurant Management System (RMS)** due to its simplicity, ease of integration with databases, and the availability of powerful libraries that streamline development. Python will serve as the core programming language for the RMS, handling business logic, database interactions, and user interfaces.

### 1. Overview of Python in RMS

Python will be used to implement the following features in the Restaurant Management System:

- **Customer Management:** Adding, viewing, and updating customer information.
- **Menu Management:** Adding, updating, and viewing menu items.
- **Order Management:** Creating, viewing, and updating orders.
- **Reservation Management:** Managing customer reservations.
- **Employee Management:** Managing employee records, roles, and schedules.

Python's `sqlite3` library will interact with the SQLite database to store and retrieve data, while Python's built-in libraries (or third-party libraries like Tkinter for GUI) can be used for building the front-end interface.

## 2. Core Features Implemented with Python

### 2.1 Database Interaction (SQLite with Python)

Python interacts with an SQLite database using the `sqlite3` module. This module allows Python to easily create, read, update, and delete data in the database.

- **Customer Management:** Allows adding customer details (e.g., name, contact, address) to the `customers` table.
- **Menu Management:** Allows adding, updating, and retrieving menu items such as food names, prices, and descriptions from the `menu` table.
- **Order Management:** Allows creating new orders and linking customers to ordered items using the `orders` and `order_details` tables.
- **Reservation Management:** Allows managing customer table reservations, including reservation time and status, using the `reservations` table.

#### User Interface:

- **CLI:** Python's `input()` function can be used to create a simple command-line interface for managing the RMS.
- **GUI:** Tkinter can be used to build a graphical interface for tasks like adding customers and menu items.

#### Benefits of Using Python:

- **Ease of Use:** Python's readable syntax makes it easy to implement and maintain.
- **Database Integration:** Easy integration with SQLite for local data storage.
- **Flexibility:** Suitable for both CLI and GUI-based systems.
- **Cross-Platform:** Python works across various operating systems, ensuring portability.

### 2.2.3 TKINTER

Tkinter is a Python library for creating graphical user interfaces (GUIs). In a Restaurant Management System (RMS), Tkinter can be used to create interactive forms and windows for managing different operations such as customer details, menu items, orders, and reservations.

### **Key Uses of Tkinter in RMS:**

1. Customer Management:
  - Create forms to add and view customer details (e.g., name, contact, email).
2. Menu Management:
  - Provide an interface for adding, updating, and viewing menu items with options for prices, descriptions, and availability.
3. Order Management:
  - Allow staff to create, update, and view customer orders, linking them to specific menu items.
4. Reservation Management:
  - Enable customers to make reservations and staff to manage existing bookings, including details like date, time, and customer information.

### **Example GUI Features:**

- Customer Form: Form to add and view customer details.
- Menu Form: Interface to add, update, and display menu items.
- Order Form: Window to create and view orders with menu selection.
- Reservation Form: System to make and track table reservations.

### **Benefits:**

- User-Friendly: Provides an intuitive interface for staff to manage restaurant operations.
- Easy to Develop: Tkinter simplifies GUI development with its built-in widgets.
- Integration: Can easily connect with SQLite for database interactions in the RMS.

### **3. REQUIREMENTS AND ANALYSIS**

#### **3.1 REQUIREMENT SPECIFICATION**

##### **1. Introduction**

The Restaurant Management System (RMS) is a software application designed to streamline and automate the daily operations of a restaurant. This system manages key aspects such as customer orders, reservations, menu management, and employee management, providing an organized and efficient way to run a restaurant business.

##### **2. Functional Requirements**

###### **- Customer Management:**

- Add, update, and view customer details (e.g., name, contact, address).
- Track customer history and loyalty points.

###### **- Menu Management:**

- Add, update, delete, and view menu items (e.g., name, description, price).
- Manage availability status of menu items.

###### **- Order Management:**

- Create, update, and track orders placed by customers.
- Link orders to customers, specify items ordered, and update order status (e.g., pending, completed).

###### **- Reservation Management:**

- Allow customers to make reservations (date, time, number of people).
- Track and modify reservation status.

### **- Employee Management:**

- Add, update, and remove employee details.
- Track roles and working hours

### **3. Non-Functional Requirements**

#### **- Performance:**

- The system should support multiple users (staff) and handle concurrent operations like placing orders and updating menus without performance issues.

#### **- Usability:**

- The user interface (UI) should be simple, intuitive, and responsive, suitable for restaurant staff who may not be tech-savvy.

#### **- Security:**

- The system should ensure that customer and employee data is protected with secure login credentials and password encryption.

#### **- Scalability:**

- The system should be designed in a way that it can be scaled to handle more customers, orders, and menu items as the restaurant grows.

### **4. Software Requirements**

- Programming Language: Python 3.x
- GUI Framework: Tkinter (for building the user interface)
- Database: SQLite 3 (for data storage)
- Operating System: Windows, Linux, or macOS (Cross-platform support)

- Libraries:
- `sqlite3` for database operations.
- `Tkinter` for building the user interface.
- `datetime` for handling dates and times (reservations, order times).

## **5. Hardware Requirements**

- Processor: Minimum 1 GHz processor
- RAM: Minimum 2GB RAM
- Storage: Minimum 500 MB of available disk space
- Display: Screen resolution of at least 1024 x 768

## **6. System Architecture**

The RMS will have a Client-Server Architecture where:

- The client is the interface where restaurant staff interact with the system (using Tkinter GUI).
- The server is the backend, where all data (e.g., customer information, menu items, orders, reservations) is stored and managed using an SQLite database.

The client communicates with the database to perform CRUD (Create, Read, Update, Delete) operations.

## **7. Database Design**

The system uses an SQLite database to store data. Some core tables are:

- Customers: Stores customer information (e.g., name, contact details).
- Menu: Stores menu items with details like name, description, price, and availability status.
- Orders: Stores order details, linking orders to customers.
- Reservations: Stores reservation details, including date, time, and customer ID.

- Employees: Stores employee information (e.g., name, role, shift times).

## **8. Use Case Analysis**

### **- Use Case 1: Add Customer**

- Actor: Restaurant staff
- Description: The staff member enters customer details (e.g., name, contact information) into the system.
- Outcome: Customer data is added to the database.

### **- Use Case : Place Order**

- Actor: Waiter/Staff
- Description: The waiter enters the customer's order (menu items and quantity) into the system.
- Outcome: An order is created and linked to a specific customer.

### **- Use Case 3: Make Reservation**

- Actor: Customer/Staff
- Description: The customer makes a reservation for a table on a specific date and time.
- Outcome: A reservation record is created in the system.

### **- Use Case 4: Manage Menu Items**

- Actor: Manager
- Description: The manager can add, update, or remove menu items based on availability or changes in the menu.
- Outcome: Menu item details are updated in the system.

## **9. System Workflow**

### **1. Customer Registration:**

- Staff inputs customer details into the system.

### **2. Order Placement:**

- Customer places an order, and the waiter selects items from the menu, adds them to the system.

### **3. Order Status Update:**

- Waiter updates the order status as "completed" once the food is delivered.

### **4. Reservation:**

- Customer or staff makes a reservation, and the system stores the reservation details.

### **5. Menu Updates:**

- Manager updates the menu items as needed.

## **10. Challenges and Considerations**

- Data Integrity: Ensuring that customer orders, reservations, and menu items are correctly linked and updated without conflicts.

- Usability: The system should be intuitive for users (restaurant staff), even if they are not familiar with computers.

- Performance: Ensuring that the system can handle multiple operations simultaneously, especially during busy hours.

## **3.2 HARDWARE AND SOFTWARE REQUIREMENTS**

For the Restaurant Management System (RMS) mini project, the following hardware and software requirements are necessary for proper system functionality and performance.



## 1. Hardware Requirements

The RMS will run as a **desktop application** and therefore has specific hardware requirements to ensure smooth operation.

### 1.1 Minimum Hardware Requirements

- **Processor:**
  - **Intel Core i3** or equivalent (1 GHz or higher)
  - Supports 64-bit processing
- **RAM:**
  - **2 GB** or more
- **Storage:**
  - At least **500 MB** of free disk space for installation and data storage (SQLite database).
- **Display:**
  - Screen resolution of at least **1024 x 768 pixels** or higher

#### Input Devices:

- Keyboard and Mouse for interaction with the system

#### Network (optional, for multi-user support):

- If the system is deployed in a networked environment, the restaurant may require a **local area network (LAN)** for accessing shared data.

## 2. Software Requirements

The RMS will require a specific set of software tools and libraries to run efficiently.

### 2.1 Operating System

- **Windows 7/8/10/11, Linux** (Ubuntu, CentOS), or **macOS**.
  - The system will be cross-platform, ensuring compatibility across the most commonly used operating systems for small businesses and restaurants.

## 2.2 Programming Language

- **Python 3.x:**

- Python is the core programming language used to build the RMS. Python is preferred due to its simplicity, readability, and ability to quickly prototype applications.

## 2.3 GUI Framework

- **Tkinter:**

- Tkinter will be used to design the graphical user interface (GUI) for the RMS. Tkinter is Python's standard library for building desktop applications with windows, buttons, forms, and text inputs.

## 2.4 Database Management System

- **SQLite 3:**

- SQLite is a lightweight, serverless, self-contained database engine that is ideal for small-scale applications. It stores data locally, making it a perfect choice for a restaurant management system with a small to medium number of users.
- **SQLite3** module (Python's built-in library) will be used to interact with the database.

## 2.5 Required Python Libraries

- **sqlite3:**

- This Python module provides an interface for interacting with SQLite databases, allowing the system to store and retrieve restaurant-related data.

- **datetime:**

- Python's **datetime** module will be used for handling and manipulating date and time data, which is critical for reservations and order timestamps.

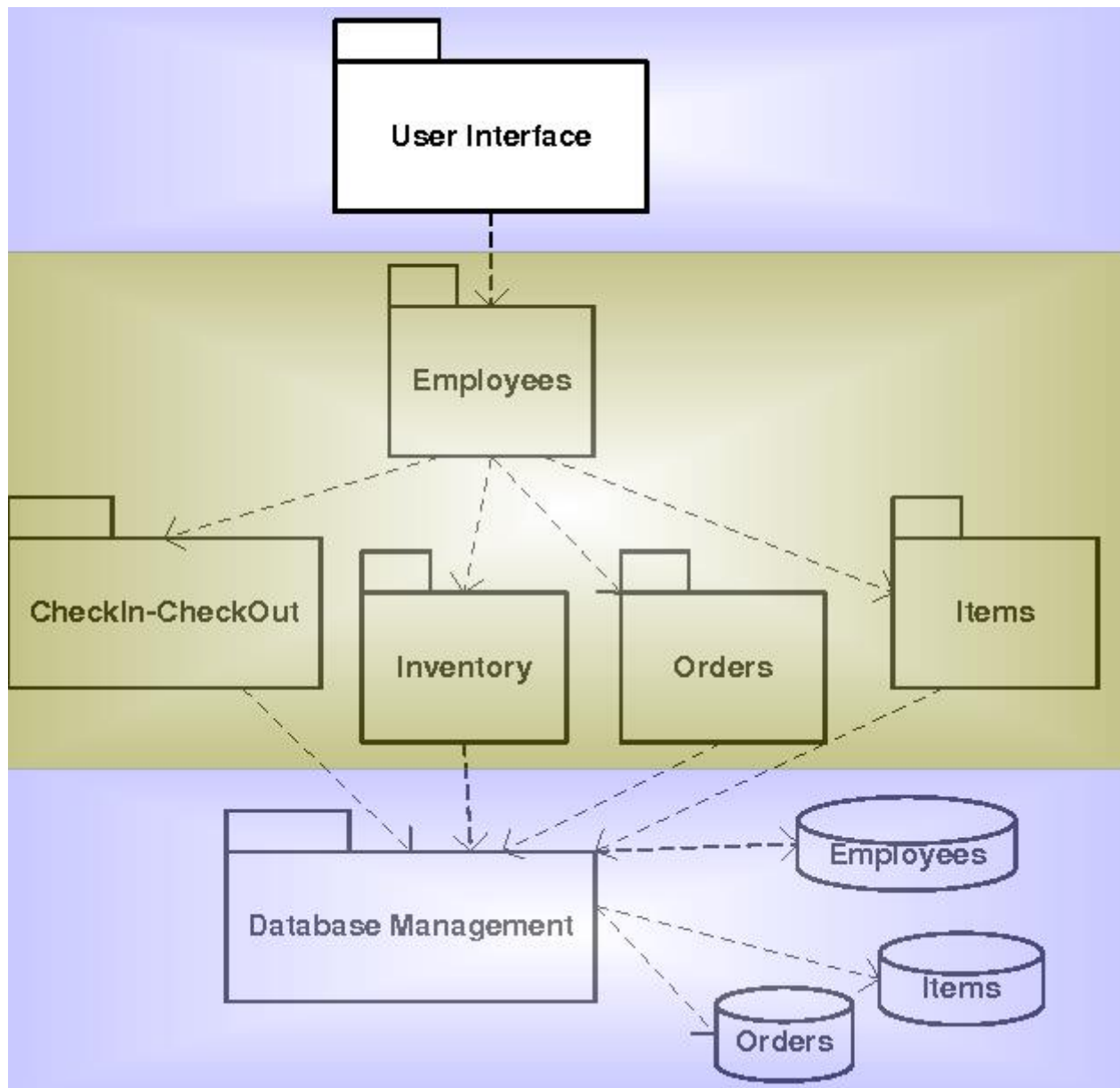
- **tkinter:**

- Tkinter will be used for creating GUI elements like windows, labels, buttons, input fields, and dialogs.

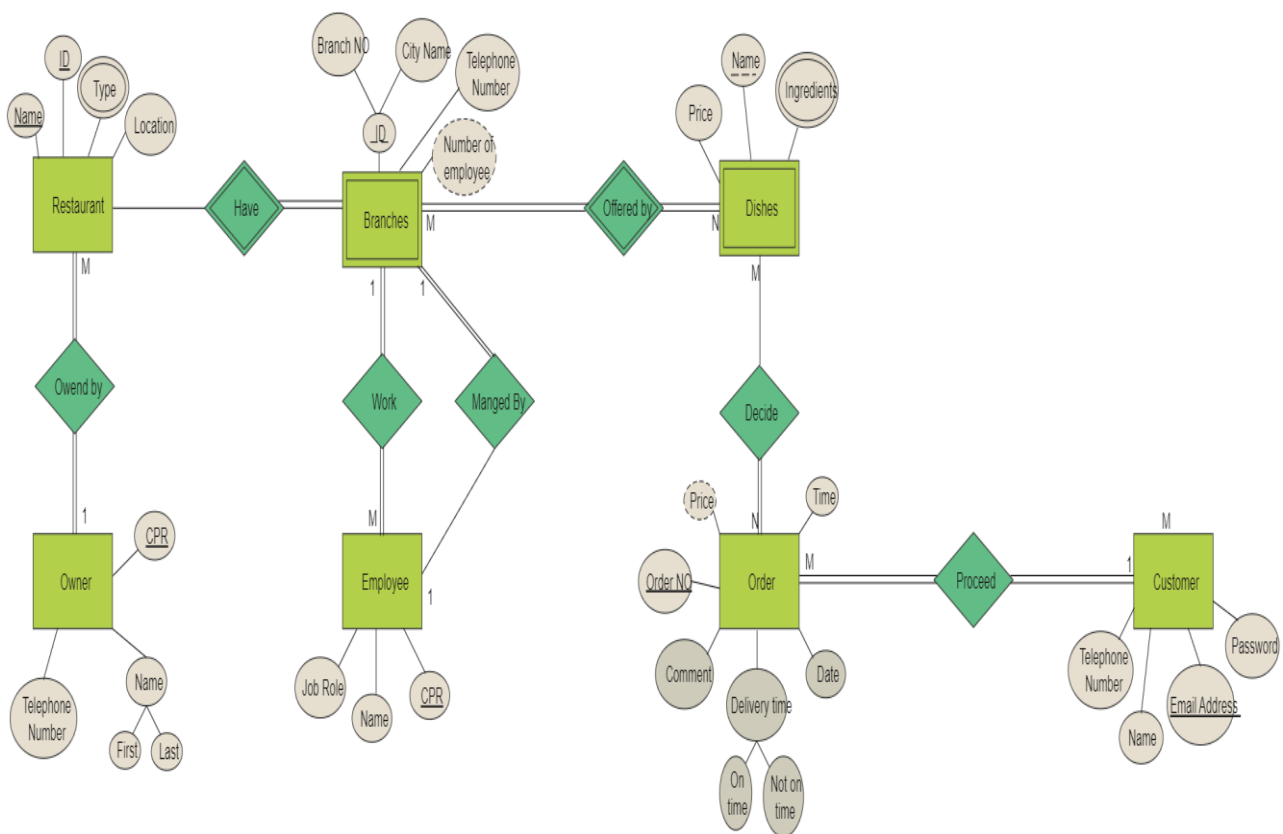
## **2.6 Web Browser (Optional)**

- For deployment of future web-based extensions or reports, a web browser (Chrome, Firefox, etc.) could be required for viewing certain reports or system outputs through a web interface (if applicable).

### 3.3 ARCHITECTURE DIAGRAM



### 3.4 ER DIAGRAM



## **4. PROGRAM CODE**

### **Aboutwindow.py**

```
import tkinter as tk

from tkinter import ttk

from ctypes import windll

windll.shcore.SetProcessDpiAwareness(1)

class AboutWindow(tk.Toplevel):

    def __init__(self, parent):

        super().__init__(parent)

        self.win_width = 360

        self.win_height = 250

        screen_width = self.winfo_screenwidth()

        screen_height = self.winfo_screenheight()

        self.center_x = int(screen_width/2 - self.win_width/2)

        self.center_y = int(screen_height/2 - self.win_height/2)

        self.geometry(

            f'{self.win_width}x{self.win_height}+{self.center_x}+{self.center_y}')

        self.resizable(0, 0)

        self.title('About the application')
```

```
about_lbl = ttk.Label(  
  
self,  
  
wraplength=300,  
  
justify='left',  
  
padding=(5, 50, 0, 0),  
  
font=("Helvetica Bold", 12),  
  
about_lbl.pack()
```

### **configurewindow.py**

```
import tkinter as tk  
  
from tkinter import ttk  
  
from tkinter import messagebox  
  
from database import Database  
  
class ConfigWindow(tk.Toplevel):  
  
def __init__(self, parent, func):  
  
super().__init__(parent)  
  
self.func = func  
  
self.init_database()  
  
self.win_width = 500  
  
self.win_height = 600  
  
screen_width = self.winfo_screenwidth()
```

```

screen_height = self.winfo_screenheight()

self.i = 5

self.center_x = int(screen_width/2 - self.win_width/2)

self.center_y = int(screen_height/2 - self.win_height/2)

self.geometry(
f'{self.win_width}x{self.win_height}+{self.center_x}+{self.center_y}')

self.title('Restaurant Management System')

self.resizable(False, False)

# main frame

self.main_frame = ttk.Frame(self)

self.main_frame.grid(row=0, column=0, sticky=tk.NSEW, padx=10, pady=10)

self.main_frame.rowconfigure(0, weight=1)

self.main_frame.rowconfigure(1, weight=1)

# up frame

self.up_frame = ttk.Frame(self.main_frame)

self.up_frame.grid(column=0, row=0, sticky=tk.NSEW)

# down frame

self.down_frame = ttk.Frame(self.main_frame)

self.down_frame.grid(column=0, row=1, sticky=tk.NSEW)

# Facility Label frame

self.fac_config_lf = ttk.LabelFrame(
self.up_frame, text="Facility Configuration")

```



```

self.fac_config_lf.grid(

column=0,

row=0,

pady=5,

rowspan=4,

columnspan=3,

sticky=tk.EW

)

# Labels

self.fc_name_lb = ttk.Label(self.fac_config_lf, text="Facility Name:")

self.fc_name_lb.grid(column=0, row=1, sticky=tk.W, padx=10, pady=10)


# fc_icon_lb = ttk.Label(fac_config_lf, text="Facility Icon")

# fc_icon_lb.grid(column=0, row=2, sticky=tk.W, padx=15, pady=15)


self.fc_table_num_lb = ttk.Label(

self.fac_config_lf,

text="Number of Tables:"

)

self.fc_table_num_lb.grid(

column=0,

row=2,

sticky=tk.W,

padx=10,

pady=10

)

```

```

self.fc_seat_num_lb = ttk.Label(
self.fac_config_lf,
text="Number of Seats:"
)
self.fc_seat_num_lb.grid(
column=0,
row=3,
sticky=tk.W,
padx=10,
pady=10
)
# Entries
self.fc_name_ent = ttk.Entry(self.fac_config_lf)
self.fc_name_ent.grid(column=1, row=1, sticky=tk.E, padx=15)
# fc_icon_ent = ttk.Entry(fac_config_lf)
# fc_icon_ent.grid(column=1, row=2, sticky=tk.E, padx=15)
vcmd_t = (self.register(self.callback_table))
vcmd_s = (self.register(self.callback_seats))
self.fc_table_num_ent = ttk.Entry(
self.fac_config_lf,
validate='all',
validatecommand=(vcmd_t, "%P")
)

```

```

self.fc_table_num_ent.grid(column=1, row=2, sticky=tk.E, padx=15)

self.fc_seat_num_ent = ttk.Entry(
self.fac_config_lf,
validate='all',
validatecommand=(vcmd_s, "%P")
)

self.fc_seat_num_ent.grid(column=1, row=3, sticky=tk.E, padx=15)

# buttons
self.fc_save_btn = ttk.Button(
self.fac_config_lf, text="Save", command=self.save_fac_config)

self.fc_save_btn.grid(column=2, row=1, pady=5, padx=15)

self.fc_load_btn = ttk.Button(
self.fac_config_lf,
text="Load",
command=self.load_fac_config,
state=tk.DISABLED
)

self.fc_load_btn.grid(column=2, row=2, pady=5, padx=15)

self.fc_clear_btn = ttk.Button(
self.fac_config_lf,
text="Clear",
command=self.fac_conf_clear,

```

```

state=tk.DISABLED

)

self.fc_clear_btn.grid(column=2, row=3, pady=5, padx=15)


# Menu Label Frame

self.menu_conf_lf = ttk.LabelFrame(

self.down_frame, text="Menu Configuration")

# self.menu_conf_lf.config(width=450)

self.menu_conf_lf.grid(column=0, row=0)

# TreeView

self.tr_v_vscr = ttk.Scrollbar(self.menu_conf_lf, orient="vertical")

self.tr_view_columns = ('id', 'name', 'price')

self.tr_view = ttk.Treeview(

self.menu_conf_lf,

columns=self.tr_view_columns,

show='headings',

height=8,

selectmode='browse',

yscrollcommand=self.tr_v_vscr.set

)

self.tr_view.column('id', width=50, anchor=tk.CENTER)

self.tr_view.column('name', width=200, anchor=tk.CENTER)

self.tr_view.column('price', width=200, anchor=tk.CENTER)

```

```

self.tr_view.heading('id', text="ID")

self.tr_view.heading('name', text="Name")

self.tr_view.heading('price', text="Price(ft)")

self.tr_view.grid(column=0, row=0, rowspan=6, columnspan=3, pady=10)

self.tr_v_vscr.config(command=self.tr_view.yview)

self.tr_v_vscr.grid(column=3, row=0, rowspan=6, sticky=tk.NS)

self.tr_view.bind('<ButtonRelease-1>', self.product_selected)

self.tr_view.bind('<Delete>', self.remove_selected)

# add product labelframe

self.add_prd_lbf = ttk.LabelFrame(

self.menu_conf_lf, text="Add product")

self.add_prd_lbf.grid(column=0, row=7, pady=10,

padx=5, columnspan=3, sticky=tk.EW)

self.remove_prd_lbf = ttk.LabelFrame(

self.menu_conf_lf, text="Remove product")

self.remove_prd_lbf.grid(

column=0, row=8, pady=10, padx=5, columnspan=3, sticky=tk.EW)

# Label and entrys

self.food_name_lbf = ttk.Label(

```

```

self.add_prd_lbf, text="Name of the product:")

self.food_name_lbf.grid(column=0, row=0, sticky=tk.W, padx=10, pady=10)

self.food_price_lbf = ttk.Label(
self.add_prd_lbf, text="Price of the product:")

self.food_price_lbf.grid(
column=0, row=1, sticky=tk.W, padx=10, pady=10)

self.food_name_entry = ttk.Entry(self.add_prd_lbf)

self.food_name_entry.grid(column=1, row=0, pady=10, padx=10)

self.food_price_entry = ttk.Entry(self.add_prd_lbf)

self.food_price_entry.grid(column=1, row=1, pady=10, padx=10)

self.pr_id_lbf = ttk.Label(
self.remove_prd_lbf, text="Product Selected:")

self.pr_id_lbf.grid(column=0, row=0, sticky=tk.W, padx=10, pady=10)

self.sel_pr_id_lbf = ttk.Label(self.remove_prd_lbf, text="")

self.sel_pr_id_lbf.grid(column=1, row=0, sticky=tk.W, padx=10, pady=10)

# btn tr

self.tr_view_add = ttk.Button(
self.add_prd_lbf, text="Add", command=self.add_record)

self.tr_view_add.grid(column=2, row=0, rowspan=2, padx=10)

```

```

self.tr_view_add.bind('<Return>', self.add_record)

self.tr_view_remove = ttk.Button(
self.remove_prd_lbf,
text="Remove",
command=self.remove_selected,
state=tk.DISABLED
)

self.tr_view_remove.grid(column=2, row=0, padx=10, sticky=tk.E)

self.check_if_empty_database()

self.check_if_empty_fc_entry()

self.retreive_menu_items()

def retreive_menu_items(self):
load_query = """SELECT * FROM menu_config"""
result = self.fac_db.read_val(load_query)

if len(result) > 0:
for el in result:
self.tr_view.insert("", tk.END, iid=f"{el[0]}", values=el)
else:
self.tr_view_remove.config(state=tk.DISABLED)

def get_product_id(self):
res = self.tr_view.get_children()

if res:

```

```

return res[-1]

else:
    return 1

def init_database(self):
    self.fac_db = Database("restaurant.db")

    fac_conf_query = """
CREATE TABLE IF NOT EXISTS fac_config(
id integer PRIMARY KEY,
fac_name text NOT NULL,
table_num integer NOT NULL,
seat_num integer NOT NULL
);
"""

    menu_conf_query = """
CREATE TABLE IF NOT EXISTS menu_config(
id integer PRIMARY KEY,
product_name text NOT NULL,
product_price real NOT NULL
);
"""

    self.fac_db.create_table(fac_conf_query)
    self.fac_db.create_table(menu_conf_query)

```



```

def check_if_empty_database(self):

load_query = """SELECT * FROM fac_config"""

result = self.fac_db.read_val(load_query)

if len(result) > 0:

f_name = result[0][1]

t_num = result[0][2]

s_num = result[0][3]

if f_name and t_num and s_num:

self.fc_load_btn.config(state=tk.ACTIVE)

else:

self.fc_load_btn.config(state=tk.DISABLED)

self.tr_view_remove.config(state=tk.DISABLED)

def check_if_empty_fc_entry(self):

f_name = self.fc_name_ent.get()

t_num = self.fc_table_num_ent.get()

s_num = self.fc_seat_num_ent.get()

if f_name and t_num and s_num:

self.fc_clear_btn.config(state=tk.ACTIVE)

else:

self.fc_clear_btn.config(state=tk.DISABLED)

def product_selected(self, event):

try:

```

```

selected_item = self.tr_view.selection()[0]

sel_item_val = self.tr_view.item(selected_item)['values']

sel_pr_txt = f"{sel_item_val[0]} {sel_item_val[1]} {sel_item_val[2]}"

self.sel_pr_id_lbl.config(text="")

self.sel_pr_id_lbl.config(text=sel_pr_txt)

self.tr_view_remove.config(state=tk.ACTIVE)

except IndexError as e:

print(e)

def fac_conf_clear(self):

self.fc_name_ent.delete(0, tk.END)

self.fc_seat_num_ent.delete(0, tk.END)

self.fc_table_num_ent.delete(0, tk.END)

self.fc_clear_btn.config(state=tk.DISABLED)

def callback_table(self, P):

if (str.isdigit(P) and int(P) <= 50) or P == "":

return True

else:

messagebox.showerror(

"Input Error", "Maximum number of tables must not exceed 50!")

return False

def callback_seats(self, P):

max_table_num = int(self.fc_table_num_ent.get()) * 8

```

```

if (str.isdigit(P) and int(P) <= max_table_num) or P == "":
    return True
else:
    messagebox.showerror(
        "Input Error", f"Maximum number of seats cannot exceed {max_table_num}")
    return False

def validate_product(self, price, name):
    if (self.is_float(price) and float(price) <= 10000000) and (len(name) <= 20):
        return True
    elif (self.is_float(price) and float(price) > 10000000) and (len(name) <= 20):
        usr_resp = messagebox.askyesno(
            "Overprice Check", "The price you have entered exceeds maximum allowed (10 million Forints), do you wish to continue?")
        if usr_resp:
            return True
        else:
            self.food_price_entry.delete(0, tk.END)
            return False
    elif (self.is_float(price) and float(price) > 10000000) and (len(name) > 20):
        usr_resp = messagebox.showerror(
            "Wrong inputs", "The price you have entered exceeds maximum allowed (10 million Forints) and product name should be less than 20 characters long")
        self.food_price_entry.delete(0, tk.END)
        self.food_name_entry.delete(0, tk.END)
        return False
    else:

```

```

messagebox.showerror(

"Wrong input", "Please enter the product name(max. 20 characters long) and price(max 10 mln forints)
correctly!")

return False

def save_fac_config(self):

load_query = """SELECT * FROM fac_config"""

result = self.fac_db.read_val(load_query)

fac_name = self.fc_name_ent.get()

table_num = self.fc_table_num_ent.get()

seat_num = self.fc_seat_num_ent.get()

if fac_name and table_num and seat_num:

if len(result) >= 1:

update_query = """UPDATE fac_config

SET fac_name = ?,

table_num = ?,

seat_num = ?,

WHERE id = ?

"""

self.fac_db.update(

update_query, (fac_name, table_num, seat_num, 1))

else:

spec_insert_query = """INSERT INTO fac_config VALUES (?, ?, ?, ?)"""

self.fac_db.insert_spec_config(

```

```

spec_insert_query, (1, fac_name, table_num, seat_num))

self.check_if_empty_database()

self.check_if_empty_fc_entry()

else:

messagebox.showerror(

"Empty input fields", "Please enter facility name, table number and seat number accordingly!")

def load_fac_config(self):

self.fac_conf_clear()

load_query = """SELECT * FROM fac_config"""

result = self.fac_db.read_val(load_query)

self.fc_name_ent.insert(0, result[0][1])

self.fc_table_num_ent.insert(0, result[0][2])

self.fc_seat_num_ent.insert(0, result[0][3])

self.check_if_empty_fc_entry()

def remove_selected(self, event=""):

try:

selected_item = self.tr_view.selection()

sel_it_ind = selected_item[0]

delete_query = """DELETE FROM menu_config WHERE id = ?"""

self.fac_db.delete_val(delete_query, [sel_it_ind])

for sel_item in selected_item:

self.tr_view.delete(sel_item)

self.sel_pr_id_lbl.config(text="")

```

```

self.tr_view_remove.config(state=tk.DISABLED)

except IndexError as e:

print(e)

def add_record(self, event='<Return>'):

food_name = self.food_name_entry.get()

food_price = self.food_price_entry.get()

spec_insert_query = """INSERT INTO menu_config VALUES (?, ?, ?)"""

pr_id = self.get_product_id()

pr_ind = pr_id if pr_id == 1 else int(pr_id) + 1

if food_name and food_price:

validate_product = self.validate_product(food_price, food_name)

if validate_product:

self.tr_view.insert("", tk.END, iid=f"{pr_ind}", values=(

pr_ind, food_name, food_price))

self.fac_db.insert_spec_config(

spec_insert_query, (pr_ind, food_name, food_price))

else:

er_msg = "Please fill \"Name of the product \" and \"Price of the product\" fields!"

messagebox.showerror("Empty input fields", er_msg)

self.food_name_entry.delete(0, tk.END)

self.food_price_entry.delete(0, tk.END)

def is_float(self, element):

```

```
if element is None:

return False

try:

float(element)

return True

except ValueError:

return False

def destroy(self):

self.func()

super().destroy()

def __del__(self):

self.func()
```

### **createorders.py**

```
import tkinter as tk

from tkinter import ttk

from tkinter import messagebox

from sqlite3 import Error

from database import Database

from productselector import ProductSelector
```

```

class CreateOrders(tk.Toplevel):
def __init__(self, parent, func):
super().__init__(parent)
self.func = func
self.init_database()
self.order_ls = []
self.win_width = 580
self.win_height = 550
screen_width = self.winfo_screenwidth()
screen_height = self.winfo_screenheight()
self.center_x = int(screen_width/2 - self.win_width/2)
self.center_y = int(screen_height/2 - self.win_height/2)
self.geometry(
f'{self.win_width}x{self.win_height}+{self.center_x}+{self.center_y}')
self.title('Restaurant Management System')
self.resizable(0, 0)
# main frame
self.main_frame = ttk.Frame(self)

```



```

self.main_frame.grid(
row=0,
column=0,
sticky=tk.NSEW,
padx=10,
pady=10,
ipady=5,
ipadx=5
)

# Labels

self.cst_lbl = ttk.LabelFrame(self.main_frame, text="Create Order")

self.cst_lbl.grid(column=0, row=0, columnspan=4,
sticky=tk.NSEW, padx=10)

self.pr_sel_lbl = ttk.Frame(self.cst_lbl)

self.pr_sel_lbl.grid(column=0, row=5, columnspan=4,
rowspan=10, sticky=tk.NSEW)

self.pr_sel_canvas = tk.Canvas(
self.pr_sel_lbl, borderwidth=0, width=520, height=400)

self.pr_sel_canvas.grid(column=0, row=0, sticky=tk.NSEW)

self.pr_sel_canvas_frm = ttk.Frame(self.pr_sel_canvas)

self.cst_lbl_scroller = ttk.Scrollbar(
self.pr_sel_lbl, command=self.pr_sel_canvas.yview)

```

```
self.cst_lbl_scroller.grid(column=4, row=0, sticky=tk.NS)

self.pr_sel_canvas.configure(yscrollcommand=self.cst_lbl_scroller.set)

self.pr_sel_canvas.create_window(
(5, 5), anchor=tk.NW, window=self.pr_sel_canvas_frm)

self.pr_sel_canvas_frm.bind("<Configure>", self.onFrameConfig)

self.fac_info = self.retrieve_fac_info()

self.fc_name = ttk.Label(
self.cst_lbl, text=f"\{self.fac_info[0]}\\"", font="Helvetica 14 bold")

self.fc_name.grid(column=1, row=0)

self.tb_name = ttk.Label(self.cst_lbl, text="Table number:")

self.tb_name.grid(column=1, row=1)

self.pr_name = ttk.Label(self.cst_lbl, text="Product Name")

self.pr_name.grid(column=0, row=3)

self.pr_qty = ttk.Label(self.cst_lbl, text="Quantity")

self.pr_qty.grid(column=1, row=3)

self.pr_st = ttk.Label(self.cst_lbl, text="Order Status")

self.pr_st.grid(column=2, row=3)
```

```

self.row_count = tk.IntVar()

self.row_count.set(1)

self.btn_add_product = ttk.Button(
self.main_frame, text="Add product", command=self.add_product)

self.btn_add_product.grid(column=0, row=2, padx=(50, 0), pady=10)

self.close_btn = ttk.Button(
self.main_frame, text='Close', command=self.destroy)

self.close_btn.grid(column=1, row=2, padx=(50, 0), pady=10)

self.send_to_ch = ttk.Button(
self.main_frame, text='Send to kitchen', state=tk.DISABLED, command=self.send_to_kitchen)

self.send_to_ch.grid(column=2, row=2, padx=(50, 0), pady=10)

vcmd_tn = (self.register(self.callback_table_num))

self.tb_name_entry = ttk.Entry(
self.cst_lbl, width=4, validate='all', validatecommand=(vcmd_tn, "%P"))

self.tb_name_entry.grid(row=1, column=1, padx=(140, 0))


def init_database(self):

self.fac_db = Database("restaurant.db")


orders_query = ""

```

```

CREATE TABLE IF NOT EXISTS orders(
id integer PRIMARY KEY,
table_num integer NOT NULL,
product_name text NOT NULL,
order_quantity integer NOT NULL,
order_status text NOT NULL
);
"""

self.fac_db.create_table(orders_query)

def retrieve_fac_info(self):
load_query = """SELECT * FROM fac_config"""
result = self.fac_db.read_val(load_query)
fac_name = result[0][1]
max_table_num = result[0][3]
return (fac_name, max_table_num)

def callback_table_num(self, P):
if (str.isdigit(P) and int(P) <= self.fac_info[1]) or P == "":
return True
else:
messagebox.showerror(
"Input Error", f"Maximum number of tables must not exceed {self.fac_info[1]}!")
self.tb_name_entry.delete(0, tk.END)
return False

```

```

def add_records(self, orders):

    try:

        load_query = """SELECT * FROM orders ORDER BY id DESC LIMIT 1; """

        spec_insert_query = """INSERT INTO orders VALUES (?, ?, ?, ?, ?)"""

        order_status = "Ordered"

        for order in orders:

            result = self.fac_db.read_val(load_query)

            if result:

                order_id = result[0][0] + 1

            else:

                order_id = 1

            order_name = order[0][0]

            order_quantity = int(order[0][1])

            table_num = int(order[1])

            self.fac_db.insert_spec_config(

                spec_insert_query, (order_id, table_num, order_name, order_quantity, order_status))

        except Error as e:

            print(e)

    def send_to_kitchen(self):

        try:

            orders = []

            table_num = self.tb_name_entry.get()

            if table_num:

                for order in self.order_ls:

                    if order.retrieve_data()[0] == 'Select a meal':

```

```

messagebox.showerror(

"Meal is not selected", "Please, either fill \"Select a meal\" or delete that row to continue!")

return False

else:

orders.append((order.retrieve_data(), table_num))

self.add_records(orders)

usr_resp = messagebox.askyesno(

"Success", "Order have been sent to the kitchen, you can access it through the main window, do you wish
to get more orders?")

if usr_resp:

self.clear()

else:

self.destroy()

else:

messagebox.showerror(

"Empty Fields", "Please enter a valid table number!")

except Error as e:

print(e)

def clear(self):

self.tb_name_entry.delete(0, tk.END)

for order in self.order_ls:

order.destroy_all()

self.order_ls = []

```

```

def add_product(self):
    self.max = self.pr_sel_canvas_frm.grid_size()
    self.row_count.set((self.row_count.get() + 1)
    if self.row_count.get() >= self.max[1] else self.max[1] + 1)
    self.pr_sl = ProductSelector(
    self, self.pr_sel_canvas_frm, self.row_count.get(), func=self.des_pr)
    self.order_ls.insert(self.row_count.get(), self.pr_sl)
    self.send_to_ch.config(state=tk.ACTIVE)
def des_pr(self):
    if len(self.pr_sel_canvas_frm.winfo_children()) == 0:
        self.send_to_ch.config(state=tk.DISABLED)
        self.row_count.set(1)
        self.order_ls = []
        self.set_val = self.row_count.get() - 1 if self.row_count.get() > 1 else 1
        self.row_count.set(self.set_val)
    def onFrameConfig(self, event):
        self.pr_sel_canvas.configure(
        scrollregion=self.pr_sel_canvas.bbox("all"))
    def destroy(self):
        self.func()
        super().destroy()

```

```

def __def__(self):
    self.func

database.py

import sqlite3

from sqlite3 import Error

class Database:

    def __init__(self, db):

        self.conn = sqlite3.connect(db)

        self.cur = self.conn.cursor()

        self.cur.execute(
            "CREATE TABLE IF NOT EXISTS gen_config (id INTEGER PRIMARY KEY, conf_name text);")

        self.conn.commit()

        self.insert_genconfig()

    def create_table(self, create_table_query):

        try:

            cursor = self.cur

            cursor.execute(create_table_query)

            self.conn.commit()

        except Error as e:

            print(e)

```



```

def insert_genconfig(self):
    try:
        self.cur.execute("INSERT OR IGNORE INTO gen_config(id, conf_name) VALUES (?, ?)",
            (1, "fac_config"))
        self.cur.execute("INSERT OR IGNORE INTO gen_config(id, conf_name) VALUES (?, ?)",
            (2, "menu_config"))
        self.cur.execute("INSERT OR IGNORE INTO gen_config(id, conf_name) VALUES (?, ?)",
            (3, "orders"))
        self.conn.commit()
    except Error as e:
        print(e)

def insert_spec_config(self, insert_query, values):
    try:
        con = self.cur
        con.execute(insert_query, values)
        self.conn.commit()
    except Error as e:
        print(e)

def update(self, update_query, values):
    try:
        con = self.cur
        con.execute(update_query, values)
        self.conn.commit()
    except Error as e:
        print(e)

```

```
def read_val(self, read_query, table_num="):
```

```
    try:
```

```
        con = self.cur
```

```
        if "WHERE" in read_query:
```

```
            con.execute(read_query, table_num)
```

```
            rows = con.fetchall()
```

```
        else:
```

```
            con.execute(read_query)
```

```
            rows = con.fetchall()
```

```
        return rows
```

```
    except Error as e:
```

```
        print(e)
```

```
def delete_val(self, delete_query, item_id):
```

```
    try:
```

```
        con = self.cur
```

```
        con.execute(delete_query, item_id)
```

```
        self.conn.commit()
```

```
    except Error as e:
```

```
        print(e)
```

```
def __del__(self):
```

```
    self.conn.close()
```

## **kitchenwindow.py**

```
import tkinter as tk

from tkinter import ttk

from orderedproducts import OrderedProducts

from database import Database

class KitchenWindow(tk.Toplevel):

    def __init__(self, parent, func):

        super().__init__(parent)

        self.func = func

        self.init_database()

        self.win_width = 625

        self.win_height = 390

        screen_width = self.winfo_screenwidth()

        screen_height = self.winfo_screenheight()

        self.center_x = int(screen_width/2 - self.win_width/2)

        self.center_y = int(screen_height/2 - self.win_height/2)

        self.geometry(

            f'{self.win_width}x{self.win_height}+{self.center_x}+{self.center_y}')

        self.title('Restaurant Management System')

        self.resizable(0, 0)
```

```

self.orders = []# main frame

self.main_frame = ttk.Frame(self)

self.main_frame.grid(row=0, column=0, sticky=tk.NSEW, padx=15, pady=10)

# Label

self.kt_lb = ttk.LabelFrame(self.main_frame, text="Kitchen receipt")

self.kt_lb.grid(column=0, row=0, sticky=tk.NSEW, columnspan=3)

# notebook

self.nt = ttk.Notebook(self.kt_lb)

self.nt.grid(column=0, row=0, ipadx=10, ipady=10, padx=10)

self.add_widgets()

def destroy(self):

    super().destroy()

    self.func()

def init_database(self):

    self.fac_db = Database("restaurant.db")

def add_widgets(self):

    retrieve_query = """SELECT table_num FROM orders GROUP BY table_num"""

    res = self.fac_db.read_val(retrieve_query)

    for r in reversed(res):

        self.retrieve_pr(r[0])

```

```
def retrieve_pr(self, table_num):  
  
self.op = OrderedProducts(  
self.main_frame, self.nt, self.kt_lb, str(table_num), self.destroy)
```

### **mainwindow.py**

```
import os  
  
import tkinter as tk  
  
from tkinter import ttk  
  
from PIL import ImageTk, Image  
  
from sqlite3 import Error  
  
from printorders import PrintOrders  
  
from configwindow import ConfigWindow  
  
from kitchenwindow import KitchenWindow  
  
from createorders import CreateOrders  
  
from aboutwindow import AboutWindow  
  
from database import Database# basedir = os.path.dirname(__file__)  
  
# print(basedir)  
  
class MainWindow(tk.Tk):  
  
def __init__(self):  
super().__init__()
```

```
self.win_width = 600

self.win_height = 400

screen_width = self.winfo_screenwidth()

screen_height = self.winfo_screenheight()

self.center_x = int(screen_width/2 - self.win_width/2)

self.center_y = int(screen_height/2 - self.win_height/2)

self.geometry(

f'{self.win_width}x{self.win_height}+{self.center_x}+{self.center_y}')

self.resizable(0, 0)

self.title('Restaurant Management System')

self.m_frame = ttk.Frame(self, width=600, height=400)

self.m_frame.grid(row=0, column=0, sticky=tk.NSEW)

icon_path = os.path.join(os.path.dirname(

os.path.dirname(__file__)), 'assets', 'icon_m.png')

self.icon_image = Image.open(icon_path)

self.python_image = ImageTk.PhotoImage(self.icon_image)

self.iconphoto(True, self.python_image)

self.menubar = tk.Menu(self.m_frame)

self.filebar = tk.Menu(self.menubar, tearoff=0)

self.filebar.add_cascade(

label="Print Receipts", command=self.print_win, state=tk.DISABLED)
```

```

self.filebar.add_cascade(
label="Kitchen", command=self.kitchen_win, state=tk.DISABLED)
self.filebar.add_cascade(
label="Create Orders", command=self.customer_win, state=tk.DISABLED)
self.filebar.add_cascade(
label="Configure Facility/Menu", command=self.config_window)
self.filebar.add_separator()
self.filebar.add_cascade(label="Exit", command=self.quit)
self.menubar.add_cascade(label="File", menu=self.filebar)

self.helpmenu = tk.Menu(self.menubar, tearoff=0)
self.helpmenu.add_command(label="About...", command=self.about_win)
self.menubar.add_cascade(label="About", menu=self.helpmenu)
self.config(menu=self.menubar)
self.img = Image.open(os.path.join(
os.path.dirname(os.path.dirname(__file__)), 'assets', 'main_win_ph.png'))
self.img = self.img.resize((250, 250), Image.Resampling.LANCZOS)
self.img = ImageTk.PhotoImage(self.img)
self.panel = tk.Label(
self.m_frame,
image=self.img,
text="Restaurant Management System",
compound='top',
font=("Helvetica Bold", 20)
)

```

```

self.panel.image = self.img

self.panel.grid(row=0, column=0, sticky=tk.NSEW, padx=90, pady=35)

self.vers = tk.Label(
self.m_frame, text="v0.1.2, N.A", font=("Helvetica", 8))

self.vers.grid(row=1, column=0, sticky=tk.SW, padx=10)

self.check_databases()

def check_databases(self):

try:

self.fac_db = Database("restaurant.db")

load_query = """SELECT * FROM menu_config"""

res = self.fac_db.read_val(load_query)

customer_state = tk.NORMAL if res else tk.DISABLED

self.filebar.entryconfig(2, state=customer_state)


load_query1 = """SELECT * FROM orders"""

res1 = self.fac_db.read_val(load_query1)


kitchen_state = tk.NORMAL if res1 else tk.DISABLED

self.filebar.entryconfig(1, state=kitchen_state)


load_query2 = """SELECT * FROM cooked_orders"""

res2 = self.fac_db.read_val(load_query2)


print_order_state = tk.NORMAL if res2 else tk.DISABLED

```



```
self.filebar.entryconfig(0, state=print_order_state)

except Error as e:

    print(e)

def config_window(self):

    config_window = ConfigWindow(self, self.check_databases)

    config_window.grab_set()

def kitchen_win(self):

    kitchen_win = KitchenWindow(self, self.check_databases)

    kitchen_win.grab_set()

def customer_win(self):

    customer_win = CreateOrders(self, self.check_databases)

    customer_win.grab_set()

def about_win(self):

    about_win = AboutWindow(self)

    about_win.grab_set()

def print_win(self):

    print_win = PrintOrders(self)

    print_win.grab_set()
```

### **orderedproducts.py**

```
import tkinter as tk

from tkinter import ttk

from sqlite3 import Error

from database import Database

class OrderedProducts(tk.Frame):

    def __init__(self, parent, root_frame, label_frame, table_num, func):

        tk.Frame.__init__(self, parent)

        self.root_frame = root_frame

        self.label_frame = label_frame

        self.table_num = f"Table {table_num}"

        self.t_num = table_num

        self.f = func

        self.init_database()

        # Frames

        self.tb = ttk.Frame(self.root_frame)

        self.tb.grid(column=0, row=0, padx=10, pady=10)

        # TreeView

        self.tr_view_columns = ('name', 'quantity', 'orderstatus')

        self.tr_view = ttk.Treeview(
```

```

self.tb,

columns=self.tr_view_columns,

show='headings',

height=10,

selectmode='browse'

)

self.tr_view.column('name', width=200, anchor=tk.CENTER)

self.tr_view.column('quantity', width=100, anchor=tk.CENTER)

self.tr_view.column('orderstatus', width=200, anchor=tk.CENTER)

self.tr_view.heading('name', text="Product Name")

self.tr_view.heading('quantity', text="Quantity")

self.tr_view.heading('orderstatus', text="Order Status")

self.tr_view.grid(column=0, row=0, rowspan=6,

columnspan=3, pady=10, padx=10, ipadx=5, ipady=5)

self.tr_v_vscr = ttk.Scrollbar(

self.tb, orient="vertical", command=self.tr_view.yview)

self.tr_v_vscr.grid(column=3, row=0, rowspan=6, sticky=tk.NS)

self.tr_view.config(yscrollcommand=self.tr_v_vscr.set)

self.root_frame.add(self.tb, text=self.table_num)

self.tr_view.bind("<ButtonRelease-1>", self.selected_item)

```

```

self.cooked_btn = ttk.Button(
self.tb, text="Cooked", command=self.change_state)
self.cooked_btn.grid(column=1, row=7, padx=(0, 200), pady=10)
self.cooked_btn.config(state='disabled')
self.flf_btn = ttk.Button(
self.tb, text="Fulfil order", command=self.fulfil_order, state=tk.DISABLED)
self.flf_btn.grid(column=1, row=7, padx=(200, 0), pady=10)
self.populate_menu()self.root_frame.select(self.tb)
def init_database(self):
self.fac_db = Database("restaurant.db")
cooked_orders = """
CREATE TABLE IF NOT EXISTS cooked_orders(
id integer PRIMARY KEY,
table_num integer NOT NULL,
product_name text NOT NULL,
order_quantity integer NOT NULL,
order_price integer NOT NULL
);
"""
self.fac_db.create_table(cooked_orders)

```

```

def populate_menu(self):

retrieve_query = """SELECT id, table_num, product_name, SUM(order_quantity) as order_quantity,
order_status FROM orders WHERE table_num = ? GROUP BY product_name ;

"""res = self.fac_db.read_val(retrieve_query, (self.t_num,))

for r in res:

product_name = r[2]

product_quantity = f"x{r[3]}"

order_status = r[4]

self.tr_view.insert("", tk.END, values=(
product_name, product_quantity, order_status))

def check_for_cooked(self):

or_stat = []

for item in self.tr_view.get_children():

it_val = self.tr_view.item(item, 'values')

or_status = it_val[2]

or_stat.append(or_status)

btn_state = tk.DISABLED if "Ordered" in or_stat else tk.ACTIVE

self.flf_btn.config(state=btn_state)

def update_order_status(self):

try:

update_query = """

UPDATE orders

SET order_status = ?

```

```

WHERE (table_num = ? AND product_name = ?)

"""

sel_item = self.tr_view.focus()

retrieved_value = self.tr_view.item(sel_item, 'values')

or_status = retrieved_value[2]

or_name = retrieved_value[0]

self.fac_db.update(update_query, (or_status, self.t_num, or_name))

# for item in self.tr_view.get_children():

#     it_val = self.tr_view.item(item, 'values')

#     or_status = it_val[2]

#     self.fac_db.update(update_query, (or_status, self.t_num))

except Error as e:

    print(e)

def get_product_price(self, pr_name):

    load_query = """SELECT product_price FROM menu_config WHERE product_name = ?"""

    res = self.fac_db.read_val(load_query, (pr_name,))

    return res[0][0]

def store_cooked_orders(self):

    try:

        load_query = """SELECT * FROM cooked_orders ORDER BY id DESC LIMIT 1; """

        spec_insert_query = """INSERT INTO cooked_orders VALUES (?, ?, ?, ?, ?)"""

```

```

for item in self.tr_view.get_children():

result = self.fac_db.read_val(load_query)

if result:

order_id = result[0][0] + 1

else:

order_id = 1

it_val = self.tr_view.item(item, 'values')

or_name = it_val[0]

or_quantity = int(it_val[1][1:])

or_price = float(self.get_product_price(or_name))

or_total = or_quantity * or_price

self.fac_db.insert_spec_config(

spec_insert_query, (order_id, self.t_num, or_name, or_quantity, or_total))

except Error as e:

print(e)

def update_order_db(self):

try:

delete_query = """"DELETE FROM orders WHERE order_status = ?""""

self.fac_db.delete_val(delete_query, ["Cooked"])

except Error as e:

print(e)

def fulfil_order(self):

self.store_cooked_orders()

self.update_order_db()

```

```

self.destroy()

self.root_frame.forget(self.tb)

if not self.root_frame.tabs():

    self.f()

def selected_item(self, event):

    self.cooked_btn.config(state=tk.ACTIVE)

    self.check_for_cooked()

def change_state(self):

    sel_item = self.tr_view.focus()

    retrieved_value = self.tr_view.item(sel_item, 'values')

    self.tr_view.item(sel_item, text="", values=(

retrieved_value[0], retrieved_value[1], "Cooked"))

    self.update_order_status()

    self.check_for_cooked()

```

### **printorders.py**

```

import tkinter as tk

from tkinter import ttk

from tkinter import messagebox

from sqlite3 import Error

```



```

import os

import webbrowser

from bs4 import BeautifulSoup
from database import Database

class PrintOrders(tk.Toplevel):

    def __init__(self, parent):

        super().__init__(parent)

        self.init_database()

        self.order_ls = []

        self.win_width = 640

        self.win_height = 470

        screen_width = self.winfo_screenwidth()

        screen_height = self.winfo_screenheight()

        self.center_x = int(screen_width/2 - self.win_width/2)

        self.center_y = int(screen_height/2 - self.win_height/2)

        self.geometry(

            f'{self.win_width}x{self.win_height}+{self.center_x}+{self.center_y}')

```

```

self.title('Restaurant Management System')

self.resizable(0, 0)

# main frame

self.main_frame = ttk.Frame(self)

self.main_frame.grid(

row=0,

column=0,

sticky=tk.NSEW,

padx=10,

pady=10,

ipady=5,

ipadx=5

)

# Labels

self.cst_lbl = ttk.LabelFrame(self.main_frame, text="Print Orders")

self.cst_lbl.grid(column=0, row=0, columnspan=4,

sticky=tk.NSEW, padx=10)

self.fac_info = self.retrieve_fac_info()

self.fc_name = ttk.Label(

self.cst_lbl,

text=f"\{self.fac_info[0]}\",

font="Helvetica 14 bold"

)

```

```

self.fc_name.grid(column=1, row=0)self.tb_name = ttk.Label(self.cst_lbl, text="Table number:")

self.tb_name.grid(column=1, row=1)

vcmd_tn = (self.register(self.callback_table_num))

self.tb_name_entry = ttk.Entry(

self.cst_lbl,

width=4,

validate='all',

validatecommand=(vcmd_tn, "%P")

)

self.tb_name_entry.grid(row=1, column=1, padx=(150, 0))

self.tr_v_vscr = ttk.Scrollbar(self.cst_lbl, orient="vertical")

self.tr_view_columns = ('id', 'name', 'quantity', 'tot_price')

self.tr_view = ttk.Treeview(

self.cst_lbl,

columns=self.tr_view_columns,

show='headings',

height=15,

selectmode='browse',

yscrollcommand=self.tr_v_vscr.set

)

```

```

self.tr_view.column('id', width=50, anchor=tk.CENTER)

self.tr_view.column('name', width=200, anchor=tk.CENTER)

self.tr_view.column('quantity', width=100, anchor=tk.CENTER)

self.tr_view.column('tot_price', width=200, anchor=tk.CENTER)

self.tr_view.heading('id', text="ID")

self.tr_view.heading('name', text="Product Name")

self.tr_view.heading('quantity', text="Quantity")

self.tr_view.heading('tot_price', text="Total Price(ft)")

self.tr_view.grid(column=0, row=3, rowspan=6,
columnspan=3, pady=10, padx=10)

self.tr_v_vscr.config(command=self.tr_view.yview)

self.tr_v_vscr.grid(column=3, row=3, rowspan=6, sticky=tk.NS)

self.row_count = tk.IntVar()

self.row_count.set(1)

self.load_orders_btn = ttk.Button(
self.main_frame, text="Load orders", command=self.load_orders)

self.load_orders_btn.grid(column=0, row=2, padx=(50, 0), pady=10)

self.close_btn = ttk.Button(
self.main_frame, text='Clear', command=self.clear_all)

self.close_btn.grid(column=1, row=2, padx=(50, 0), pady=10)

```

```

self.print_receipt_btn = ttk.Button(
    self.main_frame,
    text='Print receipt',
    state=tk.DISABLED,
    command=self.print_receipt
)

self.print_receipt_btn.grid(column=2, row=2, padx=(50, 0), pady=10)

vcmd_tn = (self.register(self.callback_table_num))

def init_database(self):
    self.fac_db = Database("restaurant.db")

def load_orders(self):
    try:
        self.t_num = self.tb_name_entry.get()

        if self.t_num:
            load_query = """SELECT id, product_name, SUM(order_quantity) as order_quantity,
SUM(order_price) as order_price FROM cooked_orders WHERE table_num = ? GROUP BY
product_name"""

            res = self.fac_db.read_val(load_query, (self.t_num))

            if res:
                for r in res:
                    self.tr_view.insert("", tk.END, values=(
                        r[0], r[1], r[2], f"{r[3]:.2f}")

            self.print_receipt_btn.config(state=tk.ACTIVE)
        else:

```

```

messagebox.showwarning(

"No orders", f"No order has been cooked for table №{self.t_num}")

self.print_receipt_btn.config(state=tk.DISABLED)

else:

messagebox.showerror(

"Empty input field", "Please fill in the table number to get orders!")

self.print_receipt_btn.config(state=tk.DISABLED)

except Error as e:

print(e)

def html_order(self, top_pad, name, quantity, price):

p_name = f"<span style='top:{150+(top_pad * 20)}pt; left:85pt; position:absolute; font-size:20pt;'>{name}</span>"

p_quantity = f"<span style='top:{150+(top_pad * 20)}pt; left:275pt; position:absolute; font-size:20pt;'>x{quantity}</span>"

p_price = f"<span style='top:{150+(top_pad * 20)}pt; right:85pt; position:absolute; font-size:20pt;'>{price}</span>"

return (BeautifulSoup(p_name, "html.parser"), BeautifulSoup(p_quantity, "html.parser"),
BeautifulSoup(p_price, "html.parser"))

def print_receipt(self):

tags = []

ind = 1

tot_p = 0

tot_q = 0

for child in self.tr_view.get_children():

val = self.tr_view.item(child)['values']

```

```

tot_p += float(val[3])

tot_q += int(val[2])

tag = self.html_order(ind, val[1], val[2], val[3])

tags.append(tag)

ind += 1

total_price = f"<span style='top:{170+(len(tags) * 40)}pt; left:85pt; position:absolute; font-size:20pt;'>Total: total ordered products {tot_q}, total price to be paid: {tot_p}</span>"

with open("order_template.html") as html_doc:

    doc = BeautifulSoup(html_doc, 'html.parser')

    doc.find(text="Fac_name").replace_with(f"\{self.fac_info[0]}\")

    doc.find(text="t_num").replace_with(f"Table №{self.t_num}")

    for tag in tags:

        doc.div.append(tag[0])

        doc.div.append(tag[1])

        doc.div.append(tag[2])

        doc.div.append(BeautifulSoup(

            f"<hr style='top:{170+(len(tags)*30)}pt;' />", "html.parser"))

        doc.div.append(BeautifulSoup(total_price, "html.parser"))

    str_doc = str(doc.prettify())

    with open(f"order_{self.t_num}.html", "w+", encoding='utf-8') as p_or_fl:

        p_or_fl.write(str_doc)

    webbrowser.open_new_tab(f"order_{self.t_num}.html")

    self.clear_all()

```

```

def clear_all(self):

self.tb_name_entry.delete(0, tk.END)

for child in self.tr_view.get_children():

self.tr_view.delete(child)

self.print_receipt_btn.config(state=tk.DISABLED)

def retrieve_fac_info(self):

load_query = """SELECT * FROM fac_config"""

result = self.fac_db.read_val(load_query)

fac_name = result[0][1]

max_table_num = result[0][3]

return (fac_name, max_table_num)

def callback_table_num(self, P):

if (str.isdigit(P) and int(P) <= self.fac_info[1]) or P == "":

return True

else:

messagebox.showerror(

"Input Error", f"Maximum number of tables must not exceed {self.fac_info[1]}!")

self.tb_name_entry.delete(0, tk.END)

return False

def destroy(self):

super().destroy()

```



## **Productselector.py**

```
import os

import tkinter as tk

from tkinter import ttk

from PIL import ImageTk, Image

from database import Database

class ProductSelector(tk.Frame):

    def __init__(self, parent, root_frame, row, func):

        self.func = func

        tk.Frame.__init__(self, parent)

        self.init_database()

        self.menuBtn = ttk.Menubutton(root_frame, text="Select a meal")

        self.menu = tk.Menu(self.menuBtn, tearoff=0)

        self.m_var1 = tk.StringVar()

        self.m_var1.set("Select a meal")

        self.retrieve_products()

        self.menuBtn['menu'] = self.menu

        self.menuBtn.grid(column=0, row=row, padx=(15, 85), sticky=tk.W)
```

```

self.pr_qty_var = tk.StringVar(root_frame)

self.pr_qty_var.set("1")

self.spin_box = ttk.Spinbox(
    root_frame,
    from_=1,
    to=100,
    textvariable=self.pr_qty_var,
    wrap=True,
    width=5,
    state=tk.DISABLED,
)

self.spin_box.grid(column=1, row=row)

self.order_st_lb = ttk.Label(root_frame, text="Choosing")

self.order_st_lb.grid(column=2, row=row, padx=(110, 10))

self.del_icon_png = Image.open(
    os.path.join(os.path.dirname(os.path.dirname(__file__)), 'assets', 'delete.png'))

self.del_icon_res = self.del_icon_png.resize(
    (18, 18), Image.Resampling.LANCZOS)

self.del_icon = ImageTk.PhotoImage(self.del_icon_res)

self.destroy_btn = ttk.Button(
    root_frame, image=self.del_icon, width=10, command=self.destroy_all)

self.destroy_btn.image = self.del_icon

self.destroy_btn.grid(column=3, row=row, padx=(10, 0))

```

```

def init_database(self):

self.fac_db = Database("restaurant.db")

def retrieve_products(self):

load_query = """SELECT * FROM menu_config"""

result = self.fac_db.read_val(load_query)

for row in result:

pr_lbl = row[1]

self.menu.add_radiobutton(

label=pr_lbl, variable=self.m_var1, command=self.sel)

def pad_num(self):

var_len = len(self.m_var1.get())

return 165 - ((var_len - 1) * 7.5)

def sel(self):

selx = self.m_var1.get()

self.menuBtn.config(text=selx)

self.pad_n = self.pad_num()

self.order_updt()

def retrieve_data(self):

return (self.m_var1.get(), self.pr_qty_var.get())

def order_updt(self):

self.order_st_lb.config(text="Ordered")

```

```
self.spin_box.config(state=tk.ACTIVE)

self.spin_box.config(textvariable=self.pr_qty_var)

self.menuBtn.grid_configure(padx=(15, self.pad_n))

def destroy_all(self):

    super().destroy()

    self.menuBtn.destroy()

    self.menu.destroy()

    self.spin_box.destroy()

    self.order_st_lb.destroy()

    self.destroy_btn.destroy()

    self.func()
```

### **rms.py**

```
from mainwindow import MainWindow

from ctypes import windll

windll.shcore.SetProcessDpiAwareness(1)

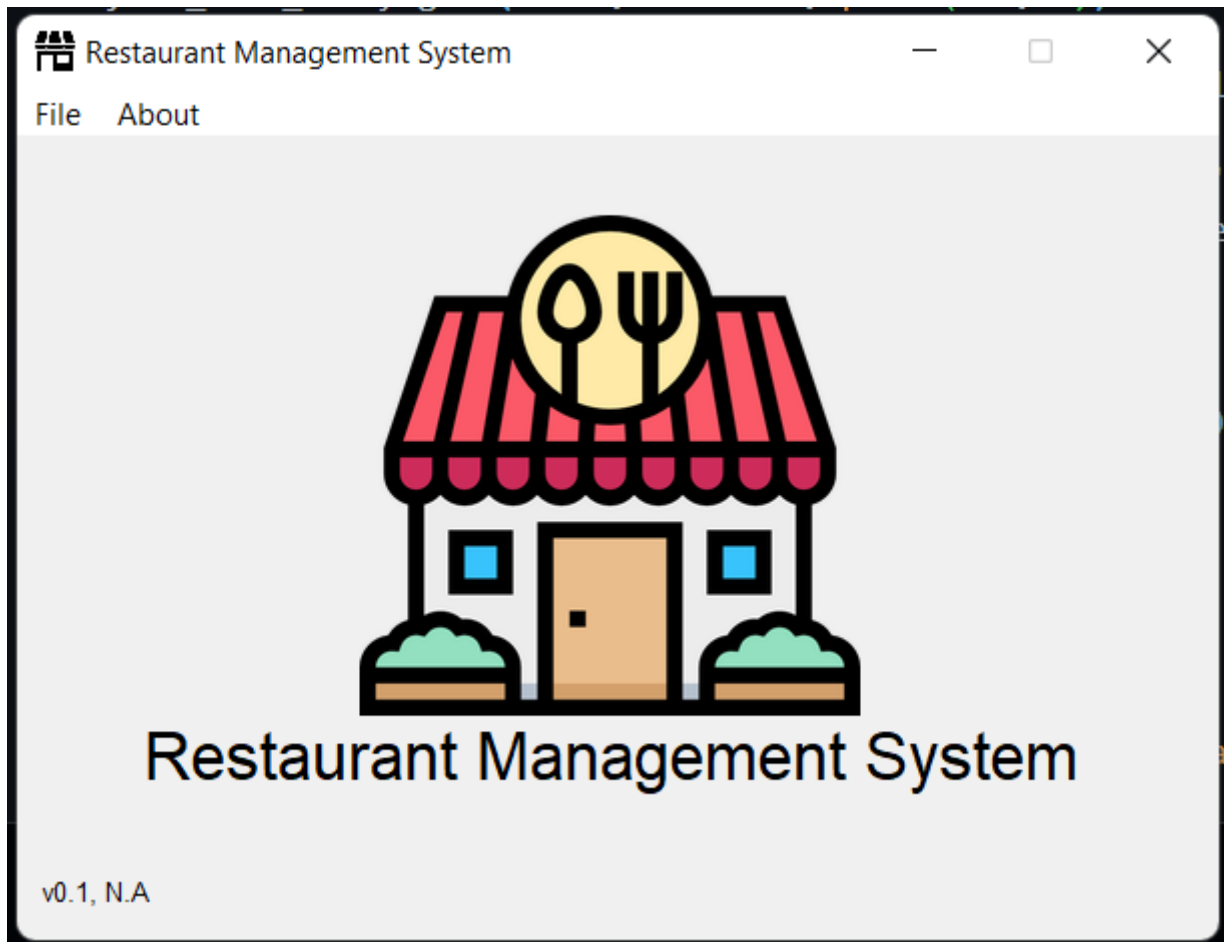
if __name__ == "__main__":

    app = MainWindow()

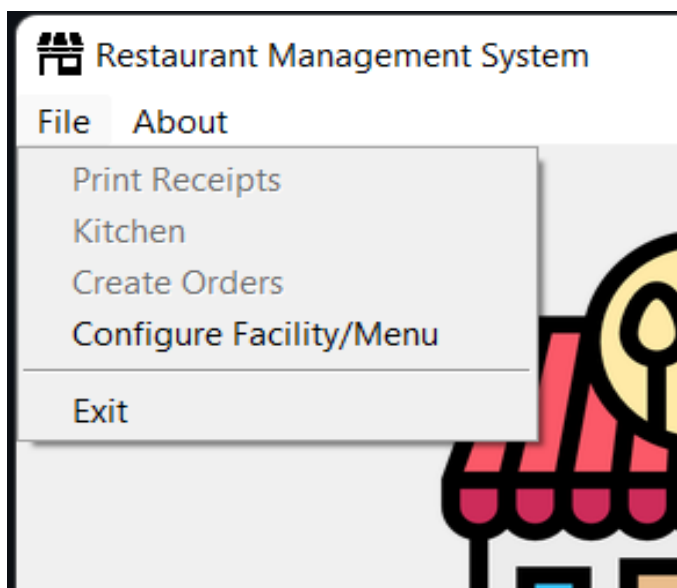
    app.mainloop()
```

## 5.PROJECT SCREENSHOTS


### Initial run



### Configuration facility



## Configuration Window

 Restaurant Management System

Facility Configuration

Facility Name:

Save

Number of Tables:

Load

Number of Seats:

Clear

Menu Configuration

ID	Name	Price(ft)
----	------	-----------

Add product

Name of the product:

Add

Price of the product:

Remove product

Product Selected:

Remove

## Facility configuration

The screenshot shows the 'Facility Configuration' window of the 'Restaurant Management System'. It contains two main sections: 'Facility Configuration' and 'Menu Configuration'.

**Facility Configuration:**

- Facility Name:  Save
- Number of Tables:  Load
- Number of Seats:  Clear

**Menu Configuration:**

ID	Name	Price(ft)
1	Samosa	30.0
2	Behl Puri	30.0

**Add product:**

Name of the product:  Add

Price of the product:

**Remove product:**

Product Selected: 2) Behl Puri 30.0 Remove

## Order window

The screenshot shows the 'Order window' of the 'Restaurant Management System'. It features a 'Create Order' section with a large text area and a table for order details.

**Create Order:**

**"GitHub"**

Table number:

Product Name	Quantity	Order Status
--------------	----------	--------------

At the bottom of the window, there are three buttons: 'Add product', 'Close', and 'Send to kitchen'.

## Printing orders

Restaurant Management System

Kitchen receipt

Table 1

Product Name	Quantity	Order Status
Python Code	x1	Ordered

Cooked      Fulfil order

Restaurant Management System

Print Orders

**"GitHub"**

Table number:

ID	Product Name	Quantity	Total Price(ft)
----	--------------	----------	-----------------

Load orders      Clear      Print receipt



"GitHub"


Table №1

Name	Quantity	Price
Python Code	x1	9999.99

Total: total ordered products 1, total price to be paid: 9999.99

Print2 sheets of paper

Destination

 Microsoft Print to PD

Pages

All

Color

Color

More settings

Print

Cancel

## **6. RESULTS AND DISCUSSION**

### **Results**

The RMS was successfully implemented with the following core functionalities:

#### **1.1 Customer Management**

- Result: The system allows restaurant staff to add, update, and view customer information, including personal details and order history.
- Testing: Successfully added new customers, updated their contact information, and searched customer details using the customer ID.

#### **1.2 Menu Management**

- Result: Staff can manage the restaurant menu by adding new items, updating prices, or removing unavailable items. The system keeps track of menu availability and price changes.
- Testing: Menu items were successfully added and updated with corresponding price changes, and the availability status was correctly displayed.

#### **1.3 Order Management**

- Result: The system allows waitstaff to place orders, link them to customers, and track the status of each order (e.g., pending, completed).
- Testing: Orders were placed, linked to customers, and marked as "completed" once served. The order history was also maintained for each customer.

#### **2.4 Reservation Management**

- Result: Customers can make table reservations by providing details such as date, time, and number of guests. The system tracks all reservations, allowing staff to view, modify, or cancel them.
- Testing: Reservations were successfully entered, displayed, and updated. Customers' reservations were recorded correctly in the system, and cancellations were processed smoothly.

## 1.5 Employee Management

- Result: Restaurant managers can add, update, or delete employee information. This includes employee roles (waiter, chef, manager), working hours, and contact details.
- Testing: Employee records were created, updated, and removed from the system without issues. The system provided accurate employee shift information.

## 1.6 Database Integration

- Result: SQLite was successfully used to store and retrieve restaurant data, including customer details, orders, reservations, and menu items.
- Testing: Data was stored in the database and retrieved efficiently, even for large numbers of records. Data integrity was maintained, and updates to the database were reflected in real time.

## 2. Discussion

### 2.1 User Interface (UI)

#### Strengths:

- The Tkinter-based graphical user interface (GUI) was intuitive and easy to use. Restaurant staff, even those with limited computer knowledge, found the system accessible and easy to navigate.
- The layout of the system was designed to minimize complexity, with clearly labeled buttons, fields, and tabs for customer management, orders, and reservations.

#### Challenges:

- Some users, particularly older staff, initially took some time to adjust to the interface, but after a brief training session, they were able to use the system effectively.
- Some advanced features, such as generating reports or multi-user support, were outside the scope of this project but could be considered for future versions to enhance usability.

## 2.2 System Performance

- Strengths:
  - The system performed well under normal conditions. Adding, updating, and deleting records were executed quickly and without significant delays.
  - The SQLite database was able to handle moderate-sized datasets effectively. For a small to medium-sized restaurant, this system will work efficiently without noticeable performance issues.
- Challenges:
  - Performance could degrade with a large number of concurrent users or complex operations. If the restaurant grows and has multiple locations, it may be necessary to upgrade to a more robust database system (e.g., MySQL, PostgreSQL) or consider a cloud-based solution for scalability.

## 2.3 Security

- Strengths:
  - Basic security features, such as password protection for user roles (e.g., waiter, manager), were implemented to ensure that only authorized personnel could access or modify sensitive data.

### Challenges:

- While basic authentication was implemented, the system could be further enhanced with encryption and more advanced security measures (e.g., role-based access control, logging, or two-factor authentication).

## 2.4 Scalability and Extensibility

- Strengths:
  - The RMS was designed to be scalable within a single restaurant setup. It supports the addition of new menu items, customers, orders, and reservations as the business grows.
- Challenges:
  - While the system is scalable in terms of data entry, multi-location support and real-time synchronization across multiple devices could pose challenges. Future enhancements could include cloud-based deployment or a web-based version of the system for more extensive scalability.

## 2.5 Data Integrity and Reliability

- Strengths:
  - Data integrity was ensured by proper database design and the use of primary and foreign keys for related tables (e.g., orders linked to customers, reservations linked to customers). There were no issues with data loss or corruption during testing.
- Challenges:
  - While SQLite is reliable for a small-scale, single-user setup, there could be potential issues when dealing with multiple users or large-scale operations. Moving to a more robust system like MySQL could improve concurrency and reliability in multi-user environments.

## 7. CONCLUSION

The Restaurant Management System (RMS) mini project successfully automates and streamlines various critical operations within a restaurant, providing a robust, user-friendly platform for managing orders, reservations, menu items, and customer and employee information. By utilizing Python, Tkinter, and SQLite, the system was designed to enhance operational efficiency, reduce manual errors, and improve overall customer service.

### **Key Achievements:**

- **Order Management:** The system allows restaurant staff to efficiently place and track customer orders, ensuring that orders are linked to specific customers and that their status (e.g., pending, completed) is updated in real-time.
- **Reservation Management:** The reservation feature enables customers to book tables for specific dates and times, and restaurant staff can easily manage and modify reservation details.
- **Menu Management:** The system provides a seamless way to add, update, and remove menu items, ensuring that the restaurant's offerings are always up-to-date.
- **Customer and Employee Management:** It allows easy addition, modification, and deletion of customer and employee records, helping maintain accurate and up-to-date information.

### **System Performance and Usability:**

- The system performed efficiently during testing, even with multiple operations running concurrently. It is lightweight and can handle typical restaurant data volumes without noticeable delays.
- The **Tkinter-based graphical user interface (GUI)** is simple and intuitive, making it accessible for restaurant staff with varying levels of technical expertise.

### **Scalability and Future Considerations:**

- While the system is scalable for a single-location restaurant, there is room for improvement in terms of handling larger datasets, multi-user environments, and multi-location restaurant chains. Future versions could implement **multi-user support**, **cloud integration**, and more **advanced reporting features**.
- **Security** measures such as stronger authentication and role-based access control can be added for enhanced data protection.

### **Final Thoughts:**

Overall, the **Restaurant Management System** provides a solid foundation for automating restaurant operations. By reducing reliance on manual processes and improving data accuracy, it can help increase operational efficiency, enhance customer experience, and allow managers to make more informed decisions based on accurate data.

## **8.REFERENCES**

- Management System for a Restaurant: <https://opus.govst.edu/capstones/569/>
- Online Food Ordering System: <https://ijcaonline.org/archives/volume180/number6/28805-2017916046/>
- Digital Ordering System for Restaurant Using Android: <https://www.ijsrp.org/research-paper-0413/ijsrp-p1605.pdf>

Python and SQLite3 Documentation:

- Official Python Documentation: <https://docs.python.org/3/>
- SQLite3 Documentation: <https://docs.python.org/3/library/sqlite3.html>

General References for Restaurant Management:

- Restaurant Management Software: <https://pos.toasttab.com/>
- Restaurant Management Systems: <https://pos.toasttab.com/>

### **GITHUB LINK:**

JASHWANTH M : <https://github.com/Jashwanth507/dbms-project.git>

PHOOJITHAA J : <https://github.com/Phoojithaa2005/SPICY-HUT.git>

IRSHAN M : <https://github.com/Irshan-M/DBMS-PROJECT-.git>

