Classes Code

Contents

[\_\_init\_\_.py 2](#_Toc185158175)

[backend.py 3](#_Toc185158176)

[customer.py 7](#_Toc185158177)

[dashboard.py 11](#_Toc185158178)

[database.py 15](#_Toc185158179)

[inventory\_management.py 17](#_Toc185158180)

[login.py 43](#_Toc185158181)

[navigation.py 53](#_Toc185158182)

[profile.py 57](#_Toc185158183)

[report\_generator.py 67](#_Toc185158184)

[sales.py 70](#_Toc185158185)

[settings.py 74](#_Toc185158186)

[terminate\_user.py 78](#_Toc185158187)

[user\_manual.py 80](#_Toc185158188)

# \_\_init\_\_.py

# Import key modules for easy access

from .backend import \*

from .database import \*

from .inventory\_management import \*

from .navigation import \*

from .login import \*

from .dashboard import \*

from .profile import \*

from .report import \*

from .report\_generator import \*

from .sales import \*

from .settings import \*

from .terminate\_user import \*

# backend.py

import sqlite3

import os

class Backend:

def \_\_init\_\_(self):

self.db\_path = r'db/Sales\_Inventory.db'

self.conn = None

self.cursor = None

def create\_database(self):

"""Creates or recreates the necessary tables."""

self.connect() # Ensure the connection is open

try:

# Create the tables

self.\_create\_users\_table()

self.\_create\_customers\_table()

self.\_create\_inventory\_items\_table()

self.\_create\_orders\_table()

self.\_create\_clock\_times\_table()

self.\_create\_user\_activity\_table()

self.conn.commit()

except sqlite3.Error as e:

print(f"Error creating database schema: {e}")

raise # Re-raise the exception for the calling method to handle

def \_create\_users\_table(self):

self.cursor.execute('''

CREATE TABLE IF NOT EXISTS users (

username TEXT PRIMARY KEY,

password TEXT NOT NULL,

role TEXT NOT NULL

)

''')

def \_create\_customers\_table(self):

self.cursor.execute('''

CREATE TABLE IF NOT EXISTS Customer (

customer\_id INTEGER PRIMARY KEY,

name TEXT NOT NULL,

contact TEXT NOT NULL

)

''')

def \_create\_inventory\_items\_table(self):

self.cursor.execute('''

CREATE TABLE IF NOT EXISTS InventoryItem (

item\_id INTEGER PRIMARY KEY,

Name TEXT NOT NULL,

SKU TEXT NOT NULL,

price REAL NOT NULL,

stockLevel INTEGER NOT NULL

)

''')

def \_create\_orders\_table(self):

self.cursor.execute('''

CREATE TABLE IF NOT EXISTS Orders (

order\_id INTEGER PRIMARY KEY,

order\_date TEXT NOT NULL,

customer TEXT NOT NULL,

item\_id TEXT NOT NULL,

product TEXT NOT NULL,

quantity INTEGER NOT NULL,

price REAL NOT NULL,

delivery\_address TEXT NOT NULL,

contact TEXT NOT NULL,

notes TEXT,

agent\_id TEXT NOT NULL

)

''')

def \_create\_transactions\_table(self):

self.cursor.execute('''

CREATE TABLE IF NOT EXISTS Transaction (

transaction\_id TEXT PRIMARY KEY,

item\_id INTEGER NOT NULL,

customer\_id INTEGER NOT NULL,

quantity INTEGER NOT NULL,

total\_price REAL NOT NULL,

date TEXT NOT NULL,

FOREIGN KEY (item\_id) REFERENCES InventoryItem(item\_id),

FOREIGN KEY (customer\_id) REFERENCES Customer(customer\_id)

)

''')

def \_create\_clock\_times\_table(self):

self.cursor.execute('''

CREATE TABLE IF NOT EXISTS clock\_times (

id INTEGER PRIMARY KEY,

username TEXT NOT NULL,

clock\_in DATETIME NOT NULL,

clock\_out DATETIME

)

''')

def \_create\_user\_activity\_table(self):

self.cursor.execute('''

CREATE TABLE IF NOT EXISTS user\_activity (

id INTEGER PRIMARY KEY,

username TEXT NOT NULL,

action TEXT NOT NULL,

timestamp DATETIME NOT NULL

)

''')

def get\_highest\_priced\_orders(self):

"""Fetch highest-priced orders and their stock levels."""

self.connect()

try:

query = '''

SELECT

i.item\_id,

i.Name AS product,

SUM(o.quantity) AS quantity\_ordered,

i.stockLevel AS stock\_level,

i.price

FROM InventoryItem i

LEFT JOIN Orders o ON i.item\_id = o.item\_id

GROUP BY i.item\_id

ORDER BY i.price DESC

LIMIT 10;

'''

self.cursor.execute(query)

return self.cursor.fetchall()

except sqlite3.Error as e:

print(f"Error fetching highest-priced orders: {e}")

return []

finally:

self.disconnect()

def get\_all\_users(self):

"""Fetch all users from the users table."""

self.connect()

try:

self.cursor.execute('SELECT username FROM users')

return [row[0] for row in self.cursor.fetchall()]

except sqlite3.Error as e:

print(f"Error fetching users: {e}")

return []

finally:

self.disconnect()

def delete\_user(self, username):

"""Delete a user from the users table."""

self.connect()

try:

self.cursor.execute('DELETE FROM users WHERE username = ?', (username,))

self.conn.commit()

except sqlite3.Error as e:

print(f"Error deleting user '{username}': {e}")

finally:

self.disconnect()

def connect(self):

"""Creates a connection to the SQLite database."""

self.conn = sqlite3.connect(self.db\_path)

self.cursor = self.conn.cursor()

def disconnect(self):

"""Closes the connection to the SQLite database."""

if self.conn:

self.conn.close()

# customer.py

import sqlite3

import os

class Customer:

def \_\_init\_\_(self):

self.db\_path = r'db/Sales\_Inventory.db'

self.conn = None

self.cursor = None

def connect(self):

self.conn = sqlite3.connect(self.db\_path)

self.cursor = self.conn.cursor()

def disconnect(self):

if self.conn:

self.conn.close()

def create\_customer(self, customer\_name, customer\_address, customer\_status=1, customer\_notes=""):

"""Create a new customer record in the customers table."""

self.connect()

try:

self.cursor.execute(

'''INSERT INTO customers (customer\_name, customer\_address, customer\_status, customer\_notes) VALUES (?, ?, ?, ?)''',

(customer\_name, customer\_address, customer\_status, customer\_notes)

)

self.conn.commit()

except sqlite3.Error as e:

print(f"Error creating customer: {e}")

finally:

self.disconnect()

def get\_customer(self, customer\_id=None):

"""Fetch customer data, optionally filtered by customer ID."""

self.connect()

try:

if customer\_id:

self.cursor.execute('SELECT \* FROM customers WHERE customer\_ID = ?', (customer\_id,))

return self.cursor.fetchone()

else:

self.cursor.execute('SELECT \* FROM customers')

return self.cursor.fetchall()

except sqlite3.Error as e:

print(f"Error fetching customers: {e}")

return []

finally:

self.disconnect()

def update\_customer(self, customer\_id, customer\_name, customer\_address, customer\_status, customer\_notes):

"""Update an existing customer record."""

self.connect()

try:

self.cursor.execute(

'''UPDATE customers SET customer\_name = ?, customer\_address = ?, customer\_status = ?, customer\_notes = ? WHERE customer\_ID = ?''',

(customer\_name, customer\_address, customer\_status, customer\_notes, customer\_id)

)

self.conn.commit()

except sqlite3.Error as e:

print(f"Error updating customer ID {customer\_id}: {e}")

finally:

self.disconnect()

def delete\_customer(self, customer\_id):

"""Delete a customer from the customers table."""

self.connect()

try:

self.cursor.execute('DELETE FROM customers WHERE customer\_ID = ?', (customer\_id,))

self.conn.commit()

except sqlite3.Error as e:

print(f"Error deleting customer ID {customer\_id}: {e}")

finally:

self.disconnect()

import sqlite3

import os

class CustomerHandler:

def \_\_init\_\_(self):

self.db\_path = 'db/Sales\_Inventory.db'

self.conn = None

self.cursor = None

def connect(self):

self.conn = sqlite3.connect(self.db\_path)

self.cursor = self.conn.cursor()

def disconnect(self):

if self.conn:

self.conn.close()

def create\_customer(self, customer\_name, customer\_address, customer\_status=1, customer\_notes=""):

"""Create a new customer record in the customers table."""

self.connect()

try:

self.cursor.execute(

'''INSERT INTO customers (customer\_name, customer\_address, customer\_status, customer\_notes) VALUES (?, ?, ?, ?)''',

(customer\_name, customer\_address, customer\_status, customer\_notes)

)

self.conn.commit()

except sqlite3.Error as e:

print(f"Error creating customer: {e}")

finally:

self.disconnect()

def get\_customer(self, customer\_id=None):

"""Fetch customer data, optionally filtered by customer ID."""

self.connect()

try:

if customer\_id:

self.cursor.execute('SELECT \* FROM customers WHERE customer\_ID = ?', (customer\_id,))

return self.cursor.fetchone()

else:

self.cursor.execute('SELECT \* FROM customers')

return self.cursor.fetchall()

except sqlite3.Error as e:

print(f"Error fetching customers: {e}")

return []

finally:

self.disconnect()

def update\_customer(self, customer\_id, customer\_name, customer\_address, customer\_status, customer\_notes):

"""Update an existing customer record."""

self.connect()

try:

self.cursor.execute(

'''UPDATE customers SET customer\_name = ?, customer\_address = ?, customer\_status = ?, customer\_notes = ? WHERE customer\_ID = ?''',

(customer\_name, customer\_address, customer\_status, customer\_notes, customer\_id)

)

self.conn.commit()

except sqlite3.Error as e:

print(f"Error updating customer ID {customer\_id}: {e}")

finally:

self.disconnect()

def delete\_customer(self, customer\_id):

"""Delete a customer from the customers table."""

self.connect()

try:

self.cursor.execute('DELETE FROM customers WHERE customer\_ID = ?', (customer\_id,))

self.conn.commit()

except sqlite3.Error as e:

print(f"Error deleting customer ID {customer\_id}: {e}")

finally:

self.disconnect()

# dashboard.py

import customtkinter

import ttkbootstrap as ttkb

from tkinter import StringVar

import matplotlib.pyplot as plt

from matplotlib.backends.backend\_tkagg import FigureCanvasTkAgg

from app.backend import Backend # Ensure this path is correct

import threading

import matplotlib

# Ensure the correct backend for matplotlib

matplotlib.use('TkAgg')

class DashboardScreen:

def \_\_init\_\_(self, app):

self.app = app

# Dashboard content frame

self.dashboard\_frame = customtkinter.CTkFrame(

self.app,

corner\_radius=15,

border\_width=0,

fg\_color="#ecf0f1"

)

self.dashboard\_frame.pack(side="right", fill="both", expand=True, padx=20, pady=20)

# Create the dashboard content

self.create\_dashboard\_content()

def fetch\_highest\_priced\_orders(self):

"""Fetch and display the highest-priced orders dynamically."""

backend = Backend()

# Fetch highest priced orders

return backend.get\_highest\_priced\_orders()

def create\_dashboard\_content(self):

# Fetch highest-priced orders

self.sales\_data = self.fetch\_highest\_priced\_orders()

# Dashboard Header

header\_frame = customtkinter.CTkFrame(self.dashboard\_frame, corner\_radius=10, fg\_color="#ecf0f1")

header\_frame.pack(fill="x", pady=10, padx=10)

header\_label = customtkinter.CTkLabel(

header\_frame,

text="Sales Dashboard",

font=("Verdana", 28, "bold"),

text\_color="#2c3e50"

)

header\_label.pack()

# Sales Data Table

self.create\_sales\_table()

# Detail View

self.detail\_frame = customtkinter.CTkFrame(self.dashboard\_frame, corner\_radius=10, fg\_color="#ecf0f1")

self.detail\_frame.pack(pady=10, fill="both", expand=True, padx=10)

self.detail\_label = customtkinter.CTkLabel(

self.detail\_frame,

text="Select an item to view details",

font=("Verdana", 18),

text\_color="#2c3e50",

wraplength=600,

justify="left"

)

self.detail\_label.pack(pady=20, padx=10)

# Animated Graph

self.graph\_frame = customtkinter.CTkFrame(self.dashboard\_frame, corner\_radius=10, fg\_color="#ecf0f1")

self.graph\_frame.pack(pady=10, fill="both", expand=True, padx=10)

self.fig, self.ax = plt.subplots(figsize=(8, 4))

self.canvas = FigureCanvasTkAgg(self.fig, master=self.graph\_frame)

self.canvas.get\_tk\_widget().pack(fill="both", expand=True)

# Bind table selection to update detail view and animated graph

self.treeview.bind("<<TreeviewSelect>>", self.update\_detail\_view)

def create\_sales\_table(self):

sales\_data\_frame = customtkinter.CTkFrame(self.dashboard\_frame, corner\_radius=10, fg\_color="#ecf0f1")

sales\_data\_frame.pack(pady=10, fill="both", expand=True, padx=10)

style = ttkb.Style()

style.configure("Treeview", rowheight=25, font=("Verdana", 12))

style.configure("Treeview.Heading", font=("Verdana", 14, "bold"))

self.treeview = ttkb.Treeview(

sales\_data\_frame,

columns=("ID", "Item", "Ordered", "InStock", "Price"),

show='headings',

bootstyle="info"

)

self.treeview.heading("ID", text="Item ID")

self.treeview.heading("Item", text="Item Name")

self.treeview.heading("Ordered", text="Quantity Ordered")

self.treeview.heading("InStock", text="In-Stock Quantity")

self.treeview.heading("Price", text="Price ($)")

self.treeview.column("ID", width=100, anchor="center")

self.treeview.column("Item", width=200, anchor="w")

self.treeview.column("Ordered", width=150, anchor="e")

self.treeview.column("InStock", width=150, anchor="e")

self.treeview.column("Price", width=150, anchor="e")

# Populate table with sales data

for row in self.sales\_data:

self.treeview.insert("", "end", values=row)

self.treeview.pack(pady=(0, 0), fill="both", expand=True)

# Add scrollbar

scrollbar = ttkb.Scrollbar(

sales\_data\_frame,

orient="vertical",

command=self.treeview.yview,

bootstyle="info-round"

)

self.treeview.configure(yscroll=scrollbar.set)

scrollbar.pack(side="right", fill="y")

def update\_detail\_view(self, event=None):

selected\_item = self.treeview.selection()

if selected\_item:

item\_data = self.treeview.item(selected\_item, 'values')

item\_id = item\_data[0]

item\_name = item\_data[1]

quantity\_ordered = int(item\_data[2])

in\_stock\_quantity = int(item\_data[3])

price = float(item\_data[4])

detail\_text = (

f"Item ID: {item\_id}\n"

f"Item Name: {item\_name}\n"

f"Quantity Ordered: {quantity\_ordered}\n"

f"In-Stock Quantity: {in\_stock\_quantity}\n"

f"Price: ${price:.2f}"

)

self.detail\_label.configure(text=detail\_text)

# Update the graph

self.update\_animated\_graph(quantity\_ordered, in\_stock\_quantity)

def update\_animated\_graph(self, ordered, in\_stock):

self.ax.clear()

categories = ['Ordered', 'In Stock']

values = [ordered, in\_stock]

self.ax.bar(categories, values, color=['blue', 'green'])

self.ax.set\_title("Stock vs Ordered Quantities", fontsize=16)

self.ax.set\_ylabel("Quantity")

self.canvas.draw()

# database.py

import sqlite3

# Function to create the users database

def create\_database():

"""Creates the users database and the 'users' table if they do not exist."""

try:

conn = sqlite3.connect(r'db/Sales\_Inventory.db')

cursor = conn.cursor()

# Create the users table if it doesn't exist

cursor.execute('''

CREATE TABLE IF NOT EXISTS users (

username TEXT PRIMARY KEY,

password TEXT NOT NULL,

role TEXT NOT NULL

)

''')

conn.commit()

except sqlite3.Error as e:

print(f"Error creating the database: {e}")

finally:

conn.close()

def initialize\_database():

conn = sqlite3.connect(r'db/Sales\_Inventory.db')

cursor = conn.cursor()

# Create user\_activity table

cursor.execute('''

CREATE TABLE IF NOT EXISTS user\_activity (

id INTEGER PRIMARY KEY AUTOINCREMENT,

username TEXT NOT NULL,

action TEXT NOT NULL,

timestamp DATETIME DEFAULT CURRENT\_TIMESTAMP,

FOREIGN KEY (username) REFERENCES users(username)

);

''')

# Create clock\_times table

cursor.execute('''

CREATE TABLE IF NOT EXISTS clock\_times (

id INTEGER PRIMARY KEY AUTOINCREMENT,

username TEXT NOT NULL,

clock\_in DATETIME,

clock\_out DATETIME,

FOREIGN KEY (username) REFERENCES users(username)

);

''')

conn.commit()

conn.close()

# Function to fetch all users from the database

def fetch\_users():

"""Fetches all usernames from the users table."""

try:

conn = sqlite3.connect(r'db/Sales\_Inventory.db')

cursor = conn.cursor()

cursor.execute("SELECT username FROM users")

users = [row[0] for row in cursor.fetchall()]

except sqlite3.Error as e:

print(f"Error fetching users: {e}")

users = []

finally:

conn.close()

return users

# Function to remove a user from the database

def remove\_user(username):

"""Removes a user from the users table based on the provided username."""

try:

conn = sqlite3.connect(r'db/Sales\_Inventory.db')

cursor = conn.cursor()

cursor.execute("DELETE FROM users WHERE username = ?", (username,))

conn.commit()

except sqlite3.Error as e:

print(f"Error removing user '{username}': {e}")

finally:

conn.close()

# inventory\_management.py

import customtkinter

import tkinter as tk

from tkinter import messagebox

from tkinter import ttk

import tkinter.ttk as ttk

import sqlite3

import datetime

from tkcalendar import Calendar

class Database:

def \_\_init\_\_(self):

self.db\_path = r'db/Sales\_Inventory.db'

self.conn = None

def connect(self):

self.conn = sqlite3.connect(self.db\_path)

return self.conn.cursor()

def close(self):

if self.conn:

self.conn.close()

def execute(self, query, params=None):

with sqlite3.connect(self.db\_path) as conn:

cursor = conn.cursor()

cursor.execute(query, params or ())

return cursor.fetchall()

class ManageInventoryScreen:

def \_\_init\_\_(self, app, backend, user\_role, username):

self.app = app

self.backend = backend

self.user\_role = user\_role

self.username = username

self.setup\_ui()

def setup\_ui(self):

self.is\_dark\_mode = False

# Main Frame

self.main\_frame = customtkinter.CTkFrame(self.app, corner\_radius=0, fg\_color=self.get\_background\_color())

self.main\_frame.pack(fill="both", expand=True)

# Create Notebook for Tabs with enhanced styling

style = ttk.Style()

style.configure('TNotebook.Tab', padding=[10, 10], font=('Arial', 14, 'bold'))

style.map('TNotebook.Tab', background=[('selected', '#1ABC9C')], foreground=[('selected', 'white')])

self.tab\_control = ttk.Notebook(self.main\_frame, style='TNotebook')

self.tab\_control.pack(expand=1, fill="both", padx=20, pady=20)

# Define Tabs

self.orders\_tab = customtkinter.CTkFrame(self.tab\_control, corner\_radius=15, fg\_color=self.get\_background\_color())

self.sales\_tab = customtkinter.CTkFrame(self.tab\_control, corner\_radius=15, fg\_color=self.get\_background\_color())

self.items\_tab = customtkinter.CTkFrame(self.tab\_control, corner\_radius=15, fg\_color=self.get\_background\_color())

# Add Tabs to Notebook

if self.user\_role in ["Supervisor", "Salesperson", "Employee"]:

self.tab\_control.add(self.orders\_tab, text='Orders')

self.setup\_orders\_tab()

if self.user\_role in ["Salesperson"]:

self.tab\_control.add(self.sales\_tab, text='Sales')

self.setup\_sales\_tab()

if self.user\_role in ["Supervisor", "Salesperson", "Employee"]:

self.tab\_control.add(self.items\_tab, text='Items')

self.setup\_items\_tab()

def setup\_orders\_tab(self):

title\_color = '#FFFFFF' if self.is\_dark\_mode else '#2C3E50'

title\_label = customtkinter.CTkLabel(self.orders\_tab, text="Orders", font=("Arial", 24, "bold"),

text\_color=title\_color)

title\_label.pack(pady=20)

# Search Bar for Orders

search\_frame = customtkinter.CTkFrame(self.orders\_tab, corner\_radius=10, fg\_color=self.get\_background\_color())

search\_frame.pack(fill="x", padx=10, pady=10)

search\_label = customtkinter.CTkLabel(search\_frame, text="Search Orders:", font=("Arial", 14))

search\_label.pack(side="left", padx=10)

self.search\_entry\_orders = customtkinter.CTkEntry(search\_frame, width=300)

self.search\_entry\_orders.pack(side="left", padx=10)

search\_button\_orders = customtkinter.CTkButton(search\_frame, text="Search", command=self.search\_orders,

fg\_color="#1ABC9C", hover\_color="#16A085")

search\_button\_orders.pack(side="left", padx=10)

# Placeholder for Order Display (Table)

columns = ("Order Date", "Order ID", "Customer")

self.order\_table = ttk.Treeview(self.orders\_tab, columns=columns, show="headings", height=10)

for col in columns:

self.order\_table.heading(col, text=col)

self.order\_table.column(col, width=200, anchor="center")

self.order\_table.pack(pady=10, padx=10, fill="both", expand=True)

# Control Panel for Supervisor

if self.user\_role == "Supervisor":

control\_frame = customtkinter.CTkFrame(self.orders\_tab, corner\_radius=10,

fg\_color=self.get\_background\_color())

control\_frame.pack(fill="x", padx=10, pady=10)

add\_order\_button = customtkinter.CTkButton(control\_frame, text="Add Order", command=self.add\_order,

fg\_color="#007BFF", hover\_color="#0056b3")

add\_order\_button.pack(side="left", padx=10, pady=10)

edit\_order\_button = customtkinter.CTkButton(control\_frame, text="Edit Order", command=self.edit\_order,

fg\_color="#FFA500", hover\_color="#FF8C00")

edit\_order\_button.pack(side="left", padx=10, pady=10)

def edit\_order(self):

selected\_item = self.order\_table.selection()

if selected\_item:

order\_details = self.order\_table.item(selected\_item, 'values')

order\_id = order\_details[1]

# Open a form to edit order details

self.open\_edit\_order\_form(order\_id)

def open\_edit\_order\_form(self, order\_id):

db = Database()

details = db.execute("""

SELECT product, quantity, price, delivery\_address, contact, notes, agent\_id

FROM Orders WHERE order\_id = ?

""", (order\_id,))

db.close()

if details:

details = details[0]

popup = tk.Toplevel(self.app)

popup.title(f"Edit Order - ID {order\_id}")

popup.geometry("450x700") # Slightly larger dimensions

popup.grab\_set()

# Create a frameless window

popup.overrideredirect(True)

popup.configure(bg='#f0f0f0')

# Scrollable container

canvas = tk.Canvas(popup, borderwidth=0, bg='#f0f0f0')

frame = tk.Frame(canvas, bg='#f0f0f0')

vsb = tk.Scrollbar(popup, orient="vertical", command=canvas.yview)

canvas.configure(yscrollcommand=vsb.set)

vsb.pack(side="right", fill="y")

canvas.pack(side="left", fill="both", expand=True)

canvas.create\_window((4, 4), window=frame, anchor="nw")

frame.bind("<Configure>", lambda event, canvas=canvas: canvas.configure(scrollregion=canvas.bbox("all")))

# Editable fields with larger font

entries = {}

labels = ["Product", "Quantity", "Price", "Delivery Address", "Contact", "Agent ID"]

for i, field in enumerate(labels):

label = tk.Label(frame, text=f"{field}:", anchor='w', font=("Arial", 14, "bold"), bg='#f0f0f0')

label.grid(row=i, column=0, sticky='w', pady=2, padx=10)

entry = tk.Entry(frame, font=("Arial", 14), bg='#ffffff', width=35)

entry.insert(0, details[i])

entry.grid(row=i, column=1, sticky='w', pady=2, padx=10)

entries[field] = entry

# Notes editable text box

notes\_label = tk.Label(frame, text="Notes:", anchor='w', font=("Arial", 14, "bold"), bg='#f0f0f0')

notes\_label.grid(row=len(labels), column=0, sticky='w', pady=2, padx=10)

notes\_textbox = tk.Text(frame, height=5, width=40, font=("Arial", 12), bg='#ffffff')

notes\_textbox.insert("1.0", details[5])

notes\_textbox.grid(row=len(labels) + 1, column=0, columnspan=2, sticky="ew", padx=10, pady=2)

# Submit and Cancel buttons

submit\_button = tk.Button(frame, text="Submit",

command=lambda: self.submit\_order\_changes(order\_id, entries, notes\_textbox),

font=("Arial", 14, "bold"), bg="#1ABC9C", fg="white")

submit\_button.grid(row=len(labels) + 2, column=0, pady=10, padx=10, sticky="ew")

cancel\_button = tk.Button(frame, text="Cancel", command=popup.destroy, font=("Arial", 14, "bold"),

bg="#FF6347", fg="white")

cancel\_button.grid(row=len(labels) + 2, column=1, pady=10, padx=10, sticky="ew")

# Make the window draggable

def on\_press(event):

global xwin, ywin

xwin = event.x

ywin = event.y

def on\_drag(event):

deltax = event.x - xwin

deltay = event.y - ywin

x = popup.winfo\_x() + deltax

y = popup.winfo\_y() + deltay

popup.geometry(f"+{x}+{y}")

popup.bind('<ButtonPress-1>', on\_press)

popup.bind('<B1-Motion>', on\_drag)

def submit\_order\_changes(self, order\_id, entries, notes\_textbox):

# Here, extract data from entries and update the database

product = entries["Product"].get()

quantity = entries["Quantity"].get()

price = entries["Price"].get()

delivery\_address = entries["Delivery Address"].get()

contact = entries["Contact"].get()

agent\_id = entries["Agent ID"].get()

notes = notes\_textbox.get("1.0", "end")

db = Database()

try:

db.connect()

db.execute("""

UPDATE Orders SET product = ?, quantity = ?, price = ?, delivery\_address = ?, contact = ?, notes = ?, agent\_id = ?

WHERE order\_id = ?

""", (product, quantity, price, delivery\_address, contact, notes, agent\_id, order\_id))

db.conn.commit() # Commit changes to the database

messagebox.showinfo("Success", "Order updated successfully.")

except Exception as e:

messagebox.showerror("Error", f"Failed to update order: {e}")

finally:

db.close()

def setup\_sales\_tab(self):

title\_color = '#FFFFFF' if self.is\_dark\_mode else '#2C3E50'

title\_label = customtkinter.CTkLabel(self.sales\_tab, text="Create Sale", font=("Arial", 24, "bold"),

text\_color=title\_color)

title\_label.grid(row=0, columnspan=2, pady=20)

self.sales\_tab.grid\_columnconfigure(0, weight=1)

self.sales\_tab.grid\_columnconfigure(1, weight=1)

form\_frame = customtkinter.CTkFrame(self.sales\_tab, corner\_radius=10, fg\_color=self.get\_background\_color())

form\_frame.grid(pady=10, padx=20, sticky="ew")

# Configure grid layout to expand the last column

form\_frame.grid\_columnconfigure(1, weight=1)

customer\_label = customtkinter.CTkLabel(form\_frame, text="Customer Name:", font=("Arial", 14))

customer\_label.grid(row=0, column=0, padx=10, pady=5, sticky="w")

self.customer\_entry = customtkinter.CTkEntry(form\_frame, width=300, fg\_color=self.get\_entry\_background\_color(),

text\_color='#000000')

self.customer\_entry.grid(row=0, column=1, padx=10, pady=5, sticky="ew")

item\_id\_label = customtkinter.CTkLabel(form\_frame, text="Item ID:", font=("Arial", 14))

item\_id\_label.grid(row=1, column=0, padx=10, pady=5, sticky="w")

self.item\_id\_var = tk.StringVar()

self.item\_id\_entry = customtkinter.CTkEntry(form\_frame, textvariable=self.item\_id\_var, width=300,

fg\_color=self.get\_entry\_background\_color(),

text\_color=self.get\_entry\_text\_color())

self.item\_id\_entry.grid(row=1, column=1, padx=10, pady=5, sticky="ew")

self.item\_id\_var.trace\_add("write", lambda name, index, mode: self.update\_product\_name())

self.item\_id\_var.trace\_add("write", lambda name, index, mode: self.update\_product\_details())

self.item\_id\_var.trace\_add("write", self.update\_product\_details) # Use self.update\_product\_details directly

product\_label = customtkinter.CTkLabel(form\_frame, text="Product Name:", font=("Arial", 14))

product\_label.grid(row=2, column=0, padx=10, pady=5, sticky="w")

self.product\_entry = customtkinter.CTkEntry(form\_frame, width=300, fg\_color="grey",

text\_color=self.get\_entry\_text\_color(), state='readonly')

self.product\_entry.grid(row=2, column=1, padx=10, pady=5, sticky="ew")

self.product\_entry.configure(fg\_color="grey", state='readonly')

quantity\_label = customtkinter.CTkLabel(form\_frame, text="Quantity:", font=("Arial", 14))

quantity\_label.grid(row=3, column=0, padx=10, pady=5, sticky="w")

self.quantity\_entry = customtkinter.CTkEntry(form\_frame, width=300, fg\_color=self.get\_entry\_background\_color(), text\_color=self.get\_entry\_text\_color())

self.quantity\_entry.grid(row=3, column=1, padx=10, pady=5, sticky="w")

price\_label = customtkinter.CTkLabel(form\_frame, text="Price per Unit:", font=("Arial", 14))

price\_label.grid(row=4, column=0, padx=10, pady=5, sticky="w")

# Price per Unit Entry: only create one entry, set it as readonly and style it appropriately

self.price\_entry = customtkinter.CTkEntry(

form\_frame, width=300,

fg\_color="grey", # Light grey background to indicate non-editability

text\_color=self.get\_entry\_text\_color(),

state='readonly' # Make the field not editable

)

self.price\_entry.grid(row=4, column=1, padx=10, pady=5, sticky="ew")

delivery\_address\_label = customtkinter.CTkLabel(form\_frame, text="Delivery Address:", font=("Arial", 14))

delivery\_address\_label.grid(row=5, column=0, padx=10, pady=5, sticky="w")

self.delivery\_address\_entry = customtkinter.CTkEntry(form\_frame, width=300, fg\_color=self.get\_entry\_background\_color(), text\_color=self.get\_entry\_text\_color())

self.delivery\_address\_entry.grid(row=5, column=1, padx=10, pady=5, sticky="w")

contact\_label = customtkinter.CTkLabel(form\_frame, text="Contact Information:", font=("Arial", 14))

contact\_label.grid(row=6, column=0, padx=10, pady=5, sticky="w")

self.contact\_entry = customtkinter.CTkEntry(form\_frame, width=300, fg\_color=self.get\_entry\_background\_color(), text\_color=self.get\_entry\_text\_color())

self.contact\_entry.grid(row=6, column=1, padx=10, pady=5, sticky="w")

# Notes Textbox

notes\_label = customtkinter.CTkLabel(form\_frame, text="Notes:", font=("Arial", 14))

notes\_label.grid(row=7, column=0, padx=10, pady=5, sticky="nw")

self.notes\_textbox = customtkinter.CTkTextbox(form\_frame, width=300, height=100, fg\_color=self.get\_entry\_background\_color(), text\_color=self.get\_entry\_text\_color())

self.notes\_textbox.grid(row=7, column=1, padx=10, pady=5, sticky="w")

# Button to Add Sale

add\_sale\_button = customtkinter.CTkButton(self.sales\_tab, text="Add Sale", command=self.add\_sale,

fg\_color="#007BFF", hover\_color="#0056b3")

add\_sale\_button.grid(row=10, column=0, columnspan=2, pady=20, padx=20,

sticky="ew") # Adjust row according to your layout needs

def update\_product\_name(self):

item\_id = self.item\_id\_var.get().strip()

if item\_id.isdigit():

db = Database()

cursor = db.connect() # Ensure you're getting a cursor back correctly

cursor.execute("SELECT Name FROM InventoryItem WHERE item\_id = ?", (item\_id,))

product\_name = cursor.fetchone() # Fetch one result

db.close()

if product\_name:

self.product\_entry.delete(0, 'end')

self.product\_entry.insert(0, product\_name[0]) # Insert product name

else:

self.product\_entry.delete(0, 'end')

def update\_product\_details(self, \*args):

"""Fetch and display product details based on the item ID."""

item\_id = self.item\_id\_var.get().strip()

if item\_id.isdigit():

db = Database()

try:

cursor = db.connect()

cursor.execute("SELECT Name, price FROM InventoryItem WHERE item\_id = ?", (item\_id,))

result = cursor.fetchone()

if result:

product\_name, price = result

self.product\_entry.configure(state='normal') # Temporarily enable the field to update it

self.product\_entry.delete(0, 'end')

self.product\_entry.insert(0, product\_name)

self.product\_entry.configure(state='readonly') # Set back to read-only

self.price\_entry.configure(state='normal') # Temporarily enable the field to update it

self.price\_entry.delete(0, 'end')

self.price\_entry.insert(0, price)

self.price\_entry.configure(state='readonly') # Set back to read-only

else:

self.clear\_product\_and\_price()

finally:

db.close()

def setup\_items\_tab(self):

# Determine text color based on dark mode status

title\_color = '#FFFFFF' if self.is\_dark\_mode else '#2C3E50'

# Create a label for the tab title

title\_label = customtkinter.CTkLabel(self.items\_tab, text="Manage Items", font=("Arial", 24, "bold"),

text\_color=title\_color)

title\_label.pack(pady=20)

# Setup the search frame for items

search\_frame = customtkinter.CTkFrame(self.items\_tab, corner\_radius=10, fg\_color=self.get\_background\_color())

search\_frame.pack(fill="x", padx=10, pady=10)

# Label for the search bar

search\_label = customtkinter.CTkLabel(search\_frame, text="Search Items:", font=("Arial", 14))

search\_label.pack(side="left", padx=10)

# Entry widget for search input

self.search\_entry\_items = customtkinter.CTkEntry(search\_frame, width=300)

self.search\_entry\_items.pack(side="left", padx=10)

# Button to trigger search

search\_button\_items = customtkinter.CTkButton(search\_frame, text="Search", command=self.search\_items,

fg\_color="#1ABC9C", hover\_color="#16A085")

search\_button\_items.pack(side="left", padx=10)

# Setup the Treeview for item display

columns = ("Item ID", "Product Name", "SKU#", "Price", "Quantity")

self.item\_table = ttk.Treeview(self.items\_tab, columns=columns, show="headings", height=10)

for col in columns:

self.item\_table.heading(col, text=col)

self.item\_table.column(col, width=150, anchor="center")

self.item\_table.pack(pady=10, padx=10, fill="both", expand=True)

# Add Item button for supervisors

if self.user\_role == "Supervisor":

add\_item\_button = customtkinter.CTkButton(self.items\_tab, text="Add Item", command=self.add\_item,

fg\_color="#007BFF", hover\_color="#0056b3")

add\_item\_button.pack(pady=10, side="bottom", anchor="center")

def toggle\_dark\_mode(self):

self.is\_dark\_mode = not self.is\_dark\_mode

new\_bg\_color = self.get\_background\_color()

new\_entry\_bg\_color = self.get\_entry\_background\_color()

new\_entry\_text\_color = self.get\_entry\_text\_color()

new\_text\_color = '#FFFFFF' if self.is\_dark\_mode else '#2C3E50'

# Update main frame and tabs

self.main\_frame.configure(fg\_color=new\_bg\_color)

for tab in [self.orders\_tab, self.sales\_tab, self.items\_tab]:

tab.configure(fg\_color=new\_bg\_color)

self.update\_tab\_titles('#000000')

# Update entries and textboxes

for entry in [self.customer\_entry, self.item\_id\_entry, self.product\_entry, self.quantity\_entry, self.price\_entry, self.delivery\_address\_entry, self.contact\_entry]:

entry.configure(fg\_color=new\_entry\_bg\_color, text\_color=new\_entry\_text\_color)

self.notes\_textbox.configure(fg\_color=new\_entry\_bg\_color, text\_color=new\_entry\_text\_color)

self.is\_dark\_mode = not self.is\_dark\_mode

new\_bg\_color = self.get\_background\_color()

new\_text\_color = '#FFFFFF' if self.is\_dark\_mode else '#2C3E50'

self.main\_frame.configure(fg\_color=new\_bg\_color)

for tab in [self.orders\_tab, self.sales\_tab, self.items\_tab]:

tab.configure(fg\_color=new\_bg\_color)

self.update\_tab\_titles(new\_text\_color)

def get\_background\_color(self):

return '#2C3E50' if self.is\_dark\_mode else 'white'

def get\_entry\_background\_color(self):

return '#3E4C59' if self.is\_dark\_mode else '#FFFFFF'

def get\_entry\_text\_color(self):

return '#FFFFFF' if self.is\_dark\_mode else '#000000'

return '#2C3E50' if self.is\_dark\_mode else 'white'

def update\_tab\_titles(self, text\_color):

for tab, title in zip([self.orders\_tab, self.sales\_tab, self.items\_tab],

["Orders", "Create Sale", "Manage Items"]):

if tab.winfo\_exists():

title\_label = tab.winfo\_children()[0]

if isinstance(title\_label, customtkinter.CTkLabel) and title\_label.winfo\_exists():

title\_label.configure(text\_color=text\_color)

def add\_order(self):

# Create a popup for adding a new order

popup = tk.Toplevel(self.app)

popup.title("Add New Order")

popup.geometry("450x700")

popup.grab\_set()

# Frameless window configuration

popup.overrideredirect(True)

popup.configure(bg='#f0f0f0')

# Scrollable container

canvas = tk.Canvas(popup, borderwidth=0, bg='#f0f0f0')

frame = tk.Frame(canvas, bg='#f0f0f0')

vsb = tk.Scrollbar(popup, orient="vertical", command=canvas.yview)

canvas.configure(yscrollcommand=vsb.set)

vsb.pack(side="right", fill="y")

canvas.pack(side="left", fill="both", expand=True)

canvas.create\_window((4, 4), window=frame, anchor="nw")

frame.bind("<Configure>", lambda event, canvas=canvas: canvas.configure(scrollregion=canvas.bbox("all")))

# Entry fields setup

entry\_fields = {

"Customer Name": None,

"Item ID": None,

"Product Name": None,

"Quantity": None,

"Price": None,

"Delivery Address": None,

"Contact Information": None,

"Agent ID": None,

"Notes": None,

}

for i, label in enumerate(entry\_fields.keys()):

tk.Label(frame, text=f"{label}:", anchor='w', font=("Arial", 14, "bold"), bg='#f0f0f0').grid(row=i,

column=0,

sticky='w',

pady=2,

padx=10)

if label == "Notes":

entry = tk.Text(frame, height=5, width=40, font=("Arial", 12), bg='#ffffff')

else:

entry = tk.Entry(frame, font=("Arial", 14), bg='#ffffff', width=35)

entry.grid(row=i, column=1, sticky='w', pady=2, padx=10)

entry\_fields[label] = entry

# Pre-fill Agent ID for convenience, but allow editing

entry\_fields["Agent ID"].insert(0, self.username)

# Submit Button

submit\_button = tk.Button(frame, text="Submit Order", command=lambda: self.submit\_order\_to\_db(entry\_fields),

font=("Arial", 14, "bold"), bg="#1ABC9C", fg="white")

submit\_button.grid(row=len(entry\_fields), column=0, columnspan=2, pady=10, padx=10, sticky="ew")

# Cancel Button

cancel\_button = tk.Button(frame, text="Cancel", command=popup.destroy, font=("Arial", 14, "bold"), bg="#FF6347",

fg="white")

cancel\_button.grid(row=len(entry\_fields) + 1, column=0, columnspan=2, pady=10, padx=10, sticky="ew")

# Make the window draggable

def on\_press(event):

global xwin, ywin

xwin = event.x

ywin = event.y

def on\_drag(event):

deltax = event.x - xwin

deltay = event.y - ywin

x = popup.winfo\_x() + deltax

y = popup.winfo\_y() + deltay

popup.geometry(f"+{x}+{y}")

popup.bind('<ButtonPress-1>', on\_press)

popup.bind('<B1-Motion>', on\_drag)

def submit\_order\_to\_db(self, entry\_fields):

try:

customer\_name = entry\_fields["Customer Name"].get()

item\_id = entry\_fields["Item ID"].get()

product\_name = entry\_fields["Product Name"].get()

quantity = int(entry\_fields["Quantity"].get())

price = float(entry\_fields["Price"].get())

delivery\_address = entry\_fields["Delivery Address"].get()

contact\_info = entry\_fields["Contact Information"].get()

agent\_id = entry\_fields["Agent ID"].get()

notes = entry\_fields["Notes"].get("1.0", "end").strip()

notes += f"\n(Note: Order created by Admin {self.username} on {datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')})"

order\_date = datetime.datetime.now().strftime("%Y-%m-%d")

db = Database()

cursor = db.connect()

# Check if enough stock is available

cursor.execute("SELECT stockLevel FROM InventoryItem WHERE item\_id = ?", (item\_id,))

result = cursor.fetchone()

if not result or result[0] < quantity:

raise Exception("Insufficient stock for the order.")

# Update stock level

new\_stock = result[0] - quantity

cursor.execute("UPDATE InventoryItem SET stockLevel = ? WHERE item\_id = ?", (new\_stock, item\_id))

# Insert the new order

cursor.execute("""

INSERT INTO Orders (order\_date, customer, item\_id, product, quantity, price, delivery\_address, contact, notes, agent\_id)

VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)

""", (

order\_date, customer\_name, item\_id, product\_name, quantity, price, delivery\_address, contact\_info, notes,

agent\_id))

db.conn.commit()

db.close()

messagebox.showinfo("Success", "Order added successfully.")

except Exception as e:

messagebox.showerror("Error", f"Failed to add order: {e}")

def add\_item(self):

# Ensure all required widgets are initialized

self.add\_item\_popup()

def add\_item\_popup(self):

popup = tk.Toplevel(self.app)

popup.title("Add New Inventory Item")

popup.geometry("450x700") # Slightly larger dimensions

popup.grab\_set()

# Create a frameless window

popup.overrideredirect(True)

popup.configure(bg='#f0f0f0')

# Scrollable container

canvas = tk.Canvas(popup, borderwidth=0, bg='#f0f0f0')

frame = tk.Frame(canvas, bg='#f0f0f0')

vsb = tk.Scrollbar(popup, orient="vertical", command=canvas.yview)

canvas.configure(yscrollcommand=vsb.set)

vsb.pack(side="right", fill="y")

canvas.pack(side="left", fill="both", expand=True)

canvas.create\_window((4, 4), window=frame, anchor="nw")

frame.bind("<Configure>", lambda event, canvas=canvas: canvas.configure(scrollregion=canvas.bbox("all")))

# Entry fields setup

entry\_fields = {

"Item Name": None,

"SKU": None,

"Price": None,

"Stock Level": None

}

for i, label in enumerate(entry\_fields.keys()):

tk.Label(frame, text=f"{label}:", anchor='w', font=("Arial", 14, "bold"), bg='#f0f0f0').grid(row=i,

column=0,

sticky='w',

pady=2,

padx=10)

entry = tk.Entry(frame, font=("Arial", 14), bg='#ffffff', width=35)

entry.grid(row=i, column=1, sticky='w', pady=2, padx=10)

entry\_fields[label] = entry

# Submit Button

submit\_button = tk.Button(frame, text="Add Item", command=lambda: self.submit\_item\_to\_db(entry\_fields),

font=("Arial", 14, "bold"), bg="#1ABC9C", fg="white")

submit\_button.grid(row=len(entry\_fields), column=0, columnspan=2, pady=10, padx=10, sticky="ew")

# Cancel Button

cancel\_button = tk.Button(frame, text="Cancel", command=popup.destroy, font=("Arial", 14, "bold"), bg="#FF6347",

fg="white")

cancel\_button.grid(row=len(entry\_fields) + 1, column=0, columnspan=2, pady=10, padx=10, sticky="ew")

# Make the window draggable

def on\_press(event):

global xwin, ywin

xwin = event.x

ywin = event.y

def on\_drag(event):

deltax = event.x - xwin

deltay = event.y - ywin

x = popup.winfo\_x() + deltax

y = popup.winfo\_y() + deltay

popup.geometry(f"+{x}+{y}")

popup.bind('<ButtonPress-1>', on\_press)

popup.bind('<B1-Motion>', on\_drag)

def submit\_item\_to\_db(self, entry\_fields):

product = entry\_fields["Item Name"].get()

sku = entry\_fields["SKU"].get()

price = float(entry\_fields["Price"].get())

stock\_level = int(entry\_fields["Stock Level"].get())

db = Database()

try:

cursor = db.connect()

# Check if the item already exists

cursor.execute("SELECT stockLevel FROM InventoryItem WHERE SKU = ?", (sku,))

result = cursor.fetchone()

if result:

# Update the existing stockLevel

existing\_stock = result[0]

new\_stock = existing\_stock + stock\_level

cursor.execute("UPDATE InventoryItem SET stockLevel = ? WHERE SKU = ?", (new\_stock, sku))

else:

# Insert a new item

cursor.execute("INSERT INTO InventoryItem (Name, SKU, price, stockLevel) VALUES (?, ?, ?, ?)",

(product, sku, price, stock\_level))

db.conn.commit()

messagebox.showinfo("Success", "Item added successfully.")

except Exception as e:

messagebox.showerror("Error", f"Failed to add item: {e}")

finally:

db.close()

def add\_sale(self):

customer = self.customer\_entry.get()

item\_id = self.item\_id\_entry.get()

product = self.product\_entry.get()

quantity = int(self.quantity\_entry.get())

price = float(self.price\_entry.get())

delivery\_address = self.delivery\_address\_entry.get()

contact = self.contact\_entry.get()

notes = self.notes\_textbox.get("1.0", "end").strip()

agent\_id = self.username

order\_date = datetime.datetime.now().strftime("%Y-%m-%d")

db = Database()

try:

cursor = db.connect()

# Check if enough stock is available

cursor.execute("SELECT stockLevel FROM InventoryItem WHERE item\_id = ?", (item\_id,))

result = cursor.fetchone()

if not result or result[0] < quantity:

raise Exception("Insufficient stock for the sale.")

# Update stock level

new\_stock = result[0] - quantity

cursor.execute("UPDATE InventoryItem SET stockLevel = ? WHERE item\_id = ?", (new\_stock, item\_id))

# Insert the sale record

cursor.execute("""

INSERT INTO Orders (order\_date, customer, item\_id, product, quantity, price, delivery\_address, contact, notes, agent\_id)

VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)

""", (order\_date, customer, item\_id, product, quantity, price, delivery\_address, contact, notes, agent\_id))

db.conn.commit()

messagebox.showinfo("Add Sale", "Sale added successfully.")

self.clear\_sale\_form()

except Exception as e:

messagebox.showerror("Error", f"Failed to add sale: {e}")

finally:

db.close()

def search\_orders(self):

search\_term = self.search\_entry\_orders.get()

db = Database()

query = """

SELECT order\_date, order\_id, customer FROM Orders

WHERE customer LIKE ? OR order\_id LIKE ?

ORDER BY order\_date DESC

"""

# Use '%' wildcards to allow partial matches

results = db.execute(query, ('%' + search\_term + '%', '%' + search\_term + '%'))

db.close()

# Clear existing entries in the table

for i in self.order\_table.get\_children():

self.order\_table.delete(i)

# Insert new results into the table

for row in results:

self.order\_table.insert('', 'end', values=row)

# Add a binding to the table for click events

self.order\_table.bind("<Double-1>", self.on\_order\_click)

def on\_order\_click(self, event):

item = self.order\_table.selection()[0]

order\_details = self.order\_table.item(item, 'values')

order\_id = order\_details[1]

db = Database()

details = db.execute("""

SELECT product, quantity, price, delivery\_address, contact, notes, agent\_id

FROM Orders WHERE order\_id = ?

""", (order\_id,))

db.close()

if details:

details = details[0]

popup = tk.Toplevel(self.app)

popup.title(f"Order Details - ID {order\_id}")

popup.geometry("450x700") # Slightly larger dimensions

popup.grab\_set()

# Create a frameless window

popup.overrideredirect(True)

popup.configure(bg='#f0f0f0')

# Scrollable container

canvas = tk.Canvas(popup, borderwidth=0, bg='#f0f0f0')

frame = tk.Frame(canvas, bg='#f0f0f0')

vsb = tk.Scrollbar(popup, orient="vertical", command=canvas.yview)

canvas.configure(yscrollcommand=vsb.set)

vsb.pack(side="right", fill="y")

canvas.pack(side="left", fill="both", expand=True)

canvas.create\_window((4, 4), window=frame, anchor="nw")

frame.bind("<Configure>", lambda event, canvas=canvas: canvas.configure(scrollregion=canvas.bbox("all")))

# Data display with larger font

labels = {

"Order ID": order\_id,

"Product": details[0],

"Quantity": details[1],

"Price": f"${details[2]}",

"Delivery Address": details[3],

"Contact": details[4],

"Agent ID": details[6]

}

for i, (label\_text, value) in enumerate(labels.items()):

label = tk.Label(frame, text=f"{label\_text}:", anchor='w', font=("Arial", 14, "bold"), bg='#f0f0f0')

value\_label = tk.Label(frame, text=value, anchor='w', font=("Arial", 14), bg='#f0f0f0')

label.grid(row=i, column=0, sticky='w', pady=2, padx=10)

value\_label.grid(row=i, column=1, sticky='w', pady=2, padx=10)

# Notes display

notes\_label = tk.Label(frame, text="Notes:", anchor='w', font=("Arial", 14, "bold"), bg='#f0f0f0')

notes\_label.grid(row=len(labels), column=0, sticky='w', pady=2, padx=10)

notes\_textbox = tk.Text(frame, height=5, width=40, font=("Arial", 12), bg='#f0f0f0')

notes\_textbox.grid(row=len(labels) + 1, column=0, columnspan=2, sticky="ew", padx=10, pady=2)

notes\_textbox.insert("1.0", details[5])

# Close button

close\_button = tk.Button(frame, text="Close", command=popup.destroy, font=("Arial", 14, "bold"),

bg="#1ABC9C", fg="white")

close\_button.grid(row=len(labels) + 2, column=0, columnspan=2, pady=10, padx=10, sticky="ew")

# Make the window draggable

def on\_press(event):

global xwin, ywin

xwin = event.x

ywin = event.y

def on\_drag(event):

deltax = event.x - xwin

deltay = event.y - ywin

x = popup.winfo\_x() + deltax

y = popup.winfo\_y() + deltay

popup.geometry(f"+{x}+{y}")

popup.bind('<ButtonPress-1>', on\_press)

popup.bind('<B1-Motion>', on\_drag)

def search\_items(self):

search\_term = self.search\_entry\_items.get()

db = Database()

try:

cursor = db.connect()

cursor.execute("SELECT \* FROM InventoryItem WHERE Name LIKE ?", ('%' + search\_term + '%',))

items = cursor.fetchall()

for i in self.item\_table.get\_children():

self.item\_table.delete(i)

for item in items:

self.item\_table.insert('', 'end', values=item)

finally:

db.close()

def clear\_content\_frame(self):

if hasattr(self, 'content\_frame') and self.content\_frame.winfo\_exists():

for widget in self.content\_frame.winfo\_children():

if widget.winfo\_exists():

widget.destroy()

def clear\_product\_and\_price(self):

"""Clears the product name and price fields."""

self.product\_entry.configure(state='normal')

self.product\_entry.delete(0, 'end')

self.product\_entry.configure(state='readonly')

self.price\_entry.configure(state='normal')

self.price\_entry.delete(0, 'end')

self.price\_entry.configure(state='readonly')

def clear\_sale\_form(self):

self.customer\_entry.delete(0, 'end')

self.item\_id\_entry.delete(0, 'end')

self.product\_entry.delete(0, 'end')

self.quantity\_entry.delete(0, 'end')

self.price\_entry.delete(0, 'end')

self.delivery\_address\_entry.delete(0, 'end')

self.contact\_entry.delete(0, 'end')

self.notes\_textbox.delete("1.0", "end")

def submit\_inventory(self):

# Collect data from fields and save to backend

data = {

"item\_id": self.item\_id\_entry.get(),

"product": self.product\_entry.get(),

"size": self.size\_entry.get(),

"quantity": int(self.quantity\_entry.get()),

"price": float(self.price\_entry.get())

}

self.backend.save\_inventory(data)

messagebox.showinfo("Success", "Inventory data saved successfully!")

def update\_inventory(self):

# Collect data from fields and update in backend

data = {

"item\_id": self.item\_id\_entry.get(),

"product": self.product\_entry.get(),

"size": self.size\_entry.get(),

"quantity": int(self.quantity\_entry.get()),

"price": float(self.price\_entry.get())

}

self.backend.save\_inventory(data)

messagebox.showinfo("Success", "Inventory data updated successfully!")

def delete\_inventory(self):

item\_id = self.item\_id\_entry.get()

confirm = messagebox.askyesno("Confirm Delete", f"Are you sure you want to delete item ID: {item\_id}?")

if confirm and self.item\_id\_entry.winfo\_exists():

self.backend.connect()

try:

self.backend.cursor.execute('DELETE FROM items WHERE item\_ID = ?', (item\_id,))

self.backend.conn.commit()

messagebox.showinfo("Success", f"Item ID: {item\_id} deleted successfully.")

except Exception as e:

messagebox.showerror("Error", f"Failed to delete item ID: {item\_id}. Error: {e}")

finally:

self.backend.disconnect()

def search\_customer(self):

# Placeholder for search customer logic

pass

def show\_staff(self):

# Placeholder for showing staff logic

pass

class ManageStaffScreen:

def \_\_init\_\_(self, app, backend, user\_role):

self.app = app

self.backend = backend

self.user\_role = user\_role

self.setup\_ui()

def setup\_ui(self):

# Main Frame

self.main\_frame = customtkinter.CTkFrame(self.app, corner\_radius=0, fg\_color=self.get\_background\_color())

self.main\_frame.pack(fill="both", expand=True)

# Create Notebook for Tabs with enhanced styling

style = ttk.Style()

style.configure('TNotebook.Tab', padding=[10, 10], font=('Arial', 14, 'bold'))

style.map('TNotebook.Tab', background=[('selected', '#1ABC9C')], foreground=[('selected', 'white')])

self.tab\_control = ttk.Notebook(self.main\_frame, style='TNotebook')

self.tab\_control.pack(expand=1, fill="both", padx=20, pady=20)

# Define Tabs

self.employees\_tab = customtkinter.CTkFrame(self.tab\_control, corner\_radius=15, fg\_color=self.get\_background\_color())

self.manage\_tab = customtkinter.CTkFrame(self.tab\_control, corner\_radius=15, fg\_color=self.get\_background\_color())

# Add Tabs to Notebook

self.tab\_control.add(self.employees\_tab, text='Employees')

self.setup\_employees\_tab()

self.tab\_control.add(self.manage\_tab, text='Manage')

self.setup\_manage\_tab()

def setup\_employees\_tab(self):

title\_color = '#FFFFFF' if self.is\_dark\_mode() else '#2C3E50'

title\_label = customtkinter.CTkLabel(self.employees\_tab, text="Employees", font=("Arial", 24, "bold"), text\_color=title\_color)

title\_label.pack(pady=20)

# Placeholder for Employee Table (TreeView)

columns = ("Username", "Role")

self.employee\_table = ttk.Treeview(self.employees\_tab, columns=columns, show="headings", height=10)

for col in columns:

self.employee\_table.heading(col, text=col)

self.employee\_table.column(col, width=150, anchor="center")

self.employee\_table.pack(pady=10, padx=10, fill="both", expand=True)

# Add a scrollbar

scrollbar = ttk.Scrollbar(self.employees\_tab, orient="vertical", command=self.employee\_table.yview)

self.employee\_table.configure(yscroll=scrollbar.set)

scrollbar.pack(side="right", fill="y")

# Load employee data

self.load\_employees()

def show\_employee\_profile(self, event=None):

selected\_item = self.employee\_table.selection()

if selected\_item:

employee\_data = self.employee\_table.item(selected\_item, 'values')

username = employee\_data[0]

profile\_window = customtkinter.CTkToplevel(self.app)

profile\_window.title("Employee Profile")

profile\_window.geometry("600x500")

profile\_window.attributes('-topmost', True)

# Recent Activity

customtkinter.CTkLabel(

profile\_window, text=f"Recent Activity: {username}", font=("Arial", 18, "bold")

).pack(pady=10)

activity\_frame = customtkinter.CTkFrame(profile\_window)

activity\_frame.pack(fill="both", expand=True, padx=10, pady=10)

conn = sqlite3.connect(r'db/Sales\_Inventory.db')

cursor = conn.cursor()

cursor.execute(

"SELECT action, timestamp FROM user\_activity WHERE username = ? ORDER BY timestamp DESC LIMIT 5",

(username,)

)

activities = cursor.fetchall()

conn.close()

for action, timestamp in activities:

customtkinter.CTkLabel(activity\_frame, text=f"{timestamp}: {action}", font=("Arial", 14)).pack(

anchor="w")

# Clock In/Out Calendar

calendar\_button = customtkinter.CTkButton(

profile\_window, text="View Clock In/Out Times", command=lambda: self.show\_calendar(username)

)

calendar\_button.pack(pady=10)

def show\_calendar(self, username):

calendar\_window = customtkinter.CTkToplevel(self.app)

calendar\_window.title("Clock In/Out Times")

calendar\_window.geometry("400x400")

calendar\_window.attributes('-topmost', True)

cal = Calendar(calendar\_window, selectmode='day')

cal.pack(fill="both", expand=True, padx=10, pady=10)

conn = sqlite3.connect(r'db/Sales\_Inventory.db')

cursor = conn.cursor()

cursor.execute("SELECT clock\_in, clock\_out FROM clock\_times WHERE username = ?", (username,))

records = cursor.fetchall()

conn.close()

for clock\_in, clock\_out in records:

customtkinter.CTkLabel(

calendar\_window, text=f"In: {clock\_in}, Out: {clock\_out}", font=("Arial", 12)

).pack(anchor="w", pady=5)

def setup\_manage\_tab(self):

title\_color = '#FFFFFF' if self.is\_dark\_mode() else '#2C3E50'

title\_label = customtkinter.CTkLabel(self.manage\_tab, text="Manage Staff", font=("Arial", 24, "bold"), text\_color=title\_color)

title\_label.pack(pady=20)

# Terminate User Button

terminate\_user\_button = customtkinter.CTkButton(

self.manage\_tab, text="Terminate User", command=self.terminate\_user, fg\_color="#FF4C4C", hover\_color="#FF1C1C", text\_color="#ffffff"

)

terminate\_user\_button.pack(pady=10)

# Change Role Button

change\_role\_button = customtkinter.CTkButton(

self.manage\_tab, text="Change Role", command=self.change\_role, fg\_color="#3498DB", hover\_color="#2980B9", text\_color="#ffffff"

)

change\_role\_button.pack(pady=10)

def load\_employees(self):

# Connect to the database

db\_path = r'db/Sales\_Inventory.db'

connection = sqlite3.connect(db\_path)

cursor = connection.cursor()

# Query to get all users

cursor.execute("SELECT username, role FROM users;")

users = cursor.fetchall()

# Insert users into the table

for user in users:

self.employee\_table.insert("", "end", values=user)

# Close the database connection

connection.close()

def change\_role(self):

selected\_item = self.employee\_table.selection()

if selected\_item:

employee\_data = self.employee\_table.item(selected\_item, 'values')

username = employee\_data[0]

current\_role = employee\_data[1]

# Create a popup window for changing the role

change\_role\_window = customtkinter.CTkToplevel(self.app)

change\_role\_window.title("Change Role")

change\_role\_window.geometry("400x300")

# Set the window to always be on top

change\_role\_window.attributes('-topmost', True)

customtkinter.CTkLabel(

change\_role\_window,

text=f"Change Role for {username}",

font=("Arial", 18, "bold"),

text\_color="#34495e"

).pack(pady=20)

role\_var = tk.StringVar(value=current\_role)

roles = ["Employee", "Supervisor", "Salesperson"] # Define allowed roles

role\_dropdown = ttk.Combobox(

change\_role\_window,

values=roles,

textvariable=role\_var,

state="readonly"

)

role\_dropdown.pack(pady=10)

def update\_role():

new\_role = role\_var.get()

if new\_role and new\_role != current\_role:

try:

# Update the database

connection = sqlite3.connect(r'db/Sales\_Inventory.db')

cursor = connection.cursor()

cursor.execute("UPDATE users SET role = ? WHERE username = ?", (new\_role, username))

connection.commit()

connection.close()

# Refresh the employee table

self.employee\_table.item(selected\_item, values=(username, new\_role))

messagebox.showinfo("Success", f"Role updated to {new\_role} for {username}.")

change\_role\_window.destroy()

except Exception as e:

messagebox.showerror("Error", f"Failed to update role: {e}")

else:

messagebox.showwarning("No Change", "Role was not updated.")

update\_button = customtkinter.CTkButton(

change\_role\_window, text="Update Role", command=update\_role, fg\_color="#2ECC71", hover\_color="#27AE60",

text\_color="#ffffff"

)

update\_button.pack(pady=20)

def terminate\_user(self):

selected\_item = self.employee\_table.selection()

if selected\_item:

employee\_data = self.employee\_table.item(selected\_item, 'values')

username = employee\_data[0]

confirm = messagebox.askyesno(

"Confirm Termination",

f"Are you sure you want to terminate {username}?"

)

if confirm:

try:

# Connect to the database

db\_path = r'db/Sales\_Inventory.db'

connection = sqlite3.connect(db\_path)

cursor = connection.cursor()

# Delete the user from the database

cursor.execute("DELETE FROM users WHERE username = ?", (username,))

connection.commit()

# Remove the user from the table view

self.employee\_table.delete(selected\_item)

messagebox.showinfo("Success", f"User {username} has been terminated successfully.")

except Exception as e:

messagebox.showerror("Error", f"Failed to terminate user: {e}")

finally:

connection.close()

def is\_dark\_mode(self):

return customtkinter.get\_appearance\_mode() == "Dark"

def get\_background\_color(self):

return '#2C3E50' if self.is\_dark\_mode() else 'white'

def show\_manage\_staff\_screen(app, backend, user\_role):

ManageStaffScreen(app, backend, user\_role)

# login.py

import customtkinter

from tkinter import messagebox

from PIL import Image, ImageTk

import sqlite3

import bcrypt

import base64

from app.user\_manual import UserManual

import threading

from time import sleep

class LoginScreen:

def \_\_init\_\_(self, app, login\_success\_callback):

self.app = app

self.login\_success\_callback = login\_success\_callback

# Set up the appearance mode and color theme

customtkinter.set\_appearance\_mode("light")

customtkinter.set\_default\_color\_theme("blue")

self.app.configure(bg="#ffffff") # Set consistent app background color

# Make the application frameless

self.app.overrideredirect(True)

# Preload the GIF during app initialization

# Main frame for the login screen - centralized card design

self.login\_frame = customtkinter.CTkFrame(

self.app, corner\_radius=5, width=400, height=650,

fg\_color="#ffffff", border\_width=3, border\_color="#c0392b"

)

self.login\_frame.place(relx=0.5, rely=0.5, anchor="center")

self.login\_frame.pack\_propagate(False) # Prevent resizing based on child widgets

# Placeholder for company logo (animated GIF)

self.load\_animated\_gif("static/images/logo.gif")

# Username Entry

customtkinter.CTkLabel(

self.login\_frame, text="Username", font=("Verdana", 16),

fg\_color="#ffffff", text\_color="#c0392b",

).pack(pady=(20, 5))

self.username\_entry = customtkinter.CTkEntry(

self.login\_frame, width=300, height=40, corner\_radius=10,

placeholder\_text="Enter your username", fg\_color="#ffffff", text\_color="#000000"

)

self.username\_entry.pack(pady=5)

self.username\_feedback = customtkinter.CTkLabel(

self.login\_frame, text="", font=("Verdana", 12),

fg\_color="#ffffff", text\_color="red"

)

self.username\_feedback.pack(pady=(0, 0))

# Password Entry

customtkinter.CTkLabel(

self.login\_frame, text="Password", font=("Verdana", 16),

fg\_color="#ffffff", text\_color="#c0392b",

).pack(pady=(20, 5))

self.password\_feedback = customtkinter.CTkLabel(

self.login\_frame, text="", font=("Verdana", 12),

fg\_color="#ffffff", text\_color="red"

)

self.app.bind("<Return>", lambda event: self.login\_action()) # Bind Enter key

self.password\_entry = customtkinter.CTkEntry(

self.login\_frame, show="\*", width=300, height=40, corner\_radius=10,

placeholder\_text="Enter your password", fg\_color="#ffffff", text\_color="#000000"

)

self.password\_entry.pack(pady=5)

# Login and Register Buttons

button\_frame = customtkinter.CTkFrame(self.login\_frame, fg\_color="transparent")

button\_frame.pack(pady=(30, 10))

register\_button = customtkinter.CTkButton(

button\_frame, text="Register", command=self.show\_register\_window, width=120, height=40, corner\_radius=10,

fg\_color="#c0392b", hover\_color="#e74c3c", text\_color="#ffffff"

)

register\_button.grid(row=0, column=0, padx=6)

login\_button = customtkinter.CTkButton(

button\_frame, text="Sign In", command=self.login\_action, width=140, height=40, corner\_radius=10,

fg\_color="#c0392b", hover\_color="#e74c3c", text\_color="#ffffff"

)

login\_button.grid(row=0, column=1, padx=6)

# Help link for signing in

help\_button = customtkinter.CTkButton(

self.login\_frame, text="Help signing in?", command=self.show\_help\_window,

fg\_color="transparent", text\_color="#c0392b", hover=False, font=("Verdana", 12)

)

help\_button.pack(pady=0)

# Custom Close Button to close the frameless window

close\_button = customtkinter.CTkButton(

self.login\_frame, text="X", command=self.close\_app, width=40, height=40, corner\_radius=20,

fg\_color="#e74c3c", hover\_color="#c0392b", text\_color="#ffffff"

)

close\_button.place(relx=0.98, rely=0.02, anchor="ne")

# Optional: Bind movement of the window (click and drag) to the entire frame

self.login\_frame.bind("<B1-Motion>", self.move\_window)

self.login\_frame.bind("<Button-1>", self.start\_move)

# Initialize fade-in effect

self.fade\_in\_effect()

def validate\_username(self):

username = self.username\_entry.get()

if len(username) < 3:

self.username\_feedback.configure(text="Username must be at least 3 characters long")

else:

self.username\_feedback.configure(text="")

def validate\_password(self):

password = self.password\_entry.get()

if len(password) < 6:

self.password\_feedback.configure(text="Password must be at least 6 characters long")

elif not any(char.isdigit() for char in password):

self.password\_feedback.configure(text="Password must include at least one number")

elif not any(char.isupper() for char in password):

self.password\_feedback.configure(text="Password must include at least one uppercase letter")

else:

self.password\_feedback.configure(text="")

def fade\_in\_effect(self, alpha=0):

"""Gradually increases window opacity from 0 to 1"""

if alpha < 1.0:

alpha += 0.05 # Adjust the increment for faster fade-in

self.app.wm\_attributes("-alpha", alpha)

self.app.after(30, lambda: self.fade\_in\_effect(alpha))

else:

self.app.wm\_attributes("-alpha", 1.0) # Ensure full opacity at the end

def load\_animated\_gif(self, gif\_path):

"""Load and display an animated GIF using CTkImage for better scaling."""

try:

# Open the GIF file

self.gif\_image = Image.open(gif\_path)

self.gif\_frames = []

self.current\_frame = 0

# Desired size for the larger logo (e.g., 300x300)

desired\_size = (280, 280)

# Extract frames and resize them to the desired size

for frame in range(self.gif\_image.n\_frames):

self.gif\_image.seek(frame)

img\_frame = self.gif\_image.copy().resize(desired\_size,

Image.Resampling.LANCZOS) # Resize with high-quality filter

# Convert to CTkImage

self.gif\_frames.append(customtkinter.CTkImage(light\_image=img\_frame, size=desired\_size))

# Create the label to display the animation

self.logo\_label = customtkinter.CTkLabel(

self.login\_frame,

image=self.gif\_frames[self.current\_frame],

text="",

fg\_color="#ffffff"

)

self.logo\_label.pack(pady=(20, 0))

# Start the animation

self.animate\_gif()

except Exception as e:

print(f"Error loading animated GIF: {e}")

def animate\_gif(self):

"""Cycle through frames of the GIF to animate."""

try:

if self.gif\_frames:

self.current\_frame = (self.current\_frame + 1) % len(self.gif\_frames)

self.logo\_label.configure(image=self.gif\_frames[self.current\_frame])

self.app.after(20, self.animate\_gif) # Adjust delay for GIF speed

except Exception as e:

print(f"Error during GIF animation: {e}")

def close\_app(self):

self.app.destroy()

def start\_move(self, event):

# Record the starting position for movement

self.\_offsetx = event.x

self.\_offsety = event.y

def move\_window(self, event):

# Calculate the new position based on pointer movement

x = self.app.winfo\_pointerx() - self.\_offsetx

y = self.app.winfo\_pointery() - self.\_offsety

self.app.geometry(f"+{x}+{y}")

def login\_action(self):

username = self.username\_entry.get()

password = self.password\_entry.get()

self.validate\_username()

self.validate\_password()

if not username or not password:

return

conn = sqlite3.connect(r'db/Sales\_Inventory.db')

cursor = conn.cursor()

cursor.execute("SELECT password, role FROM users WHERE username = ?", (username,))

result = cursor.fetchone()

conn.close()

if result:

stored\_hashed\_password\_base64 = result[0]

role = result[1]

stored\_hashed\_password = base64.b64decode(stored\_hashed\_password\_base64.encode('utf-8'))

if bcrypt.checkpw(password.encode('utf-8'), stored\_hashed\_password):

self.login\_success\_callback(username, role)

else:

messagebox.showerror("Login Failed", "Invalid username or password")

else:

messagebox.showerror("Login Failed", "Invalid username or password")

def show\_register\_window(self):

if not hasattr(self, 'register\_window') or not self.register\_window.winfo\_exists():

self.register\_window = RegisterWindow(self.app)

else:

self.register\_window.lift()

def show\_help\_window(self):

# Create and display the User Manual window

UserManual(self.app)

def show\_register\_window(self):

if not hasattr(self, 'register\_window') or not self.register\_window.winfo\_exists():

self.register\_window = RegisterWindow(self.app)

else:

self.register\_window.lift()

class RegisterWindow(customtkinter.CTkToplevel):

def \_\_init\_\_(self, parent):

super().\_\_init\_\_(parent)

self.title("Register")

self.geometry("500x600")

self.resizable(False, False)

self.configure(bg="#ffffff")

# Center the window

self.update\_idletasks()

x = (self.winfo\_screenwidth() - self.winfo\_reqwidth()) // 2

y = (self.winfo\_screenheight() - self.winfo\_reqheight()) // 2

self.geometry(f"+{x}+{y}")

# Registration form

self.create\_registration\_form()

def create\_registration\_form(self):

self.register\_frame = customtkinter.CTkFrame(self, corner\_radius=15, fg\_color="#ffffff")

self.register\_frame.pack(fill="both", expand=True, padx=20, pady=20)

self.register\_frame.pack\_propagate(False)

# Title

title\_label = customtkinter.CTkLabel(

self.register\_frame,

text="Create an Account",

font=("Verdana", 24, "bold"),

text\_color="#c0392b"

)

title\_label.pack(pady=20)

# Username Entry

customtkinter.CTkLabel(

self.register\_frame,

text="Username",

font=("Verdana", 16),

text\_color="#c0392b"

).pack(pady=(10, 5))

self.username\_entry = customtkinter.CTkEntry(

self.register\_frame,

width=300,

height=40,

corner\_radius=10,

placeholder\_text="Enter a username",

fg\_color="#ffffff",

text\_color="#000000"

)

self.username\_entry.pack(pady=5)

# Password Entry

customtkinter.CTkLabel(

self.register\_frame,

text="Password",

font=("Verdana", 16),

text\_color="#c0392b"

).pack(pady=(10, 5))

self.password\_entry = customtkinter.CTkEntry(

self.register\_frame,

show="\*",

width=300,

height=40,

corner\_radius=10,

placeholder\_text="Enter a password",

fg\_color="#ffffff",

text\_color="#000000"

)

self.password\_entry.pack(pady=5)

# Confirm Password Entry

customtkinter.CTkLabel(

self.register\_frame,

text="Confirm Password",

font=("Verdana", 16),

text\_color="#c0392b"

).pack(pady=(10, 5))

self.confirm\_password\_entry = customtkinter.CTkEntry(

self.register\_frame,

show="\*",

width=300,

height=40,

corner\_radius=10,

placeholder\_text="Confirm your password",

fg\_color="#ffffff",

text\_color="#000000"

)

self.confirm\_password\_entry.pack(pady=5)

# Role Selection

customtkinter.CTkLabel(

self.register\_frame,

text="Role",

font=("Verdana", 16),

text\_color="#c0392b"

).pack(pady=(10, 5))

self.role\_optionmenu = customtkinter.CTkOptionMenu(

self.register\_frame,

values=["Employee"],

width=300,

height=40,

corner\_radius=10,

button\_color="#c0392b",

button\_hover\_color="#e74c3c",

text\_color="#000000"

)

self.role\_optionmenu.set("Employee") # Default value

self.role\_optionmenu.pack(pady=5)

# Register Button

register\_button = customtkinter.CTkButton(

self.register\_frame,

text="Register",

command=self.register\_user,

width=200,

height=40,

corner\_radius=10,

fg\_color="#c0392b",

hover\_color="#e74c3c",

text\_color="#ffffff"

)

register\_button.pack(pady=30)

def register\_user(self):

username = self.username\_entry.get()

password = self.password\_entry.get()

confirm\_password = self.confirm\_password\_entry.get()

role = self.role\_optionmenu.get()

if not username or not password or not confirm\_password:

messagebox.showwarning("Input Error", "Please fill out all fields.")

return

if password != confirm\_password:

messagebox.showwarning("Password Mismatch", "Passwords do not match.")

return

# Validate password complexity (optional)

if len(password) < 6:

messagebox.showwarning("Weak Password", "Password must be at least 6 characters long.")

return

# Hash the password

hashed\_password = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())

# Encode the hashed password using Base64

hashed\_password\_base64 = base64.b64encode(hashed\_password).decode('utf-8')

# Save the new user to the database

conn = sqlite3.connect(r'db/Sales\_Inventory.db') # Adjust the path if needed

cursor = conn.cursor()

# Create the users table if it doesn't exist

cursor.execute('''

CREATE TABLE IF NOT EXISTS users (

username TEXT PRIMARY KEY,

password TEXT NOT NULL,

role TEXT NOT NULL

)

''')

# Check if the username already exists

cursor.execute("SELECT \* FROM users WHERE username = ?", (username,))

if cursor.fetchone():

messagebox.showwarning("Username Taken", "This username is already taken.")

conn.close()

return

# Insert the new user

cursor.execute(

"INSERT INTO users (username, password, role) VALUES (?, ?, ?)",

(username, hashed\_password\_base64, role)

)

conn.commit()

conn.close()

messagebox.showinfo("Registration Successful", "Your account has been created.")

self.destroy() # Close the registration window

def lift(self):

self.attributes('-topmost', True)

self.attributes('-topmost', False)

# navigation.py

import customtkinter

from app.inventory\_management import show\_manage\_staff\_screen

customtkinter.set\_appearance\_mode("Dark")

customtkinter.set\_default\_color\_theme("dark-blue")

class CreateNavigationPane:

def \_\_init\_\_(

self,

app,

show\_dashboard\_screen,

show\_manage\_screen,

show\_profile\_screen,

show\_settings\_screen,

show\_manage\_staff\_screen=None,

show\_generate\_report\_screen=None,

):

self.app = app

self.active\_button = None

self.show\_manage\_staff\_screen = show\_manage\_staff\_screen

# Navigation pane setup

self.nav\_frame = customtkinter.CTkFrame(

app,

width=220,

corner\_radius=0,

fg\_color="transparent"

)

self.nav\_frame.pack(side="left", fill="y")

# Navigation Header

self.nav\_header = customtkinter.CTkLabel(

self.nav\_frame,

text="Menu",

font=("Verdana", 20, "bold"),

text\_color=self.get\_text\_color(),

fg\_color="transparent"

)

self.nav\_header.pack(pady=20, fill="x")

# Define navigation buttons

nav\_buttons = [

{"text": "Dashboard", "command": lambda: self.navigate(show\_dashboard\_screen, btn\_index=0)},

{"text": "Manage Inventory", "command": lambda: self.navigate(show\_manage\_screen, btn\_index=1)},

{"text": "Profile", "command": lambda: self.navigate(show\_profile\_screen, btn\_index=2)},

{"text": "Settings", "command": lambda: self.navigate(show\_settings\_screen, btn\_index=3)},

]

# Add supervisor options if applicable

if self.show\_manage\_staff\_screen:

nav\_buttons.append(

{"text": "Manage Staff", "command": lambda: self.navigate(self.show\_manage\_staff\_screen, btn\_index=4)}

)

if show\_generate\_report\_screen:

nav\_buttons.append(

{"text": "Generate Report", "command": lambda: self.navigate(show\_generate\_report\_screen, btn\_index=5)}

)

# Create and style navigation buttons

self.nav\_buttons = []

for i, btn\_info in enumerate(nav\_buttons):

btn = customtkinter.CTkButton(

self.nav\_frame,

text=btn\_info["text"],

command=btn\_info["command"],

height=40,

width=220,

font=("Verdana", 16),

text\_color=self.get\_text\_color(),

fg\_color="transparent",

hover\_color="#34495e",

corner\_radius=0,

border\_width=0

)

btn.pack(fill="x", pady=(0, 2))

self.nav\_buttons.append(btn)

# Add Logout button at the bottom

self.logout\_button = customtkinter.CTkButton(

self.nav\_frame,

text="Log Out",

command=lambda: self.navigate(app.show\_login\_screen, btn\_index=None),

height=40,

width=220,

font=("Verdana", 16),

text\_color="#ffffff",

fg\_color="#e74c3c",

hover\_color="#c0392b",

corner\_radius=0,

border\_width=0

)

self.logout\_button.pack(side="bottom", fill="x", pady=(10, 0))

# Assign nav\_frame to app for access and clearing later

app.nav\_frame = self.nav\_frame

# Apply initial styles to buttons

self.style\_buttons()

def get\_text\_color(self):

mode = customtkinter.get\_appearance\_mode()

return "#ffffff" if mode == "Dark" else "black"

def navigate(self, screen\_command, btn\_index=None):

"""

Navigates to a specified screen and updates the navigation button state.

Args:

screen\_command (callable): The function to execute for navigating to the desired screen.

btn\_index (int, optional): The index of the navigation button to highlight. Defaults to None.

"""

if screen\_command and callable(screen\_command):

try:

# Ensure the app window exists and execute the screen command

if hasattr(self.app, 'winfo\_exists') and self.app.winfo\_exists():

# Clear frames if not navigating to the report generation screen

if screen\_command != self.app.show\_generate\_report\_screen:

self.app.clear\_frames(exclude\_nav=True)

screen\_command()

except Exception as e:

print(f"Error executing screen command: {e}")

else:

print("Invalid screen command provided.")

# Update button styles and set the active button if applicable

self.style\_buttons()

if btn\_index is not None and hasattr(self, 'nav\_buttons') and self.nav\_buttons:

self.set\_active\_button(btn\_index)

def destroy(self):

if hasattr(self, 'nav\_frame') and self.nav\_frame.winfo\_exists():

for widget in self.nav\_frame.winfo\_children():

widget.destroy()

def style\_buttons(self):

text\_color = "#FFFFFF" if customtkinter.get\_appearance\_mode() == "Dark" else "#000000"

if hasattr(self, 'nav\_header') and self.nav\_header.winfo\_exists():

try:

self.nav\_header.configure(text\_color=text\_color)

except Exception as e:

print(f"Error configuring nav\_header: {e}")

def set\_active\_button(self, btn\_index):

# Reset all buttons to default appearance first

if hasattr(self, 'nav\_buttons') and self.nav\_buttons:

for button in self.nav\_buttons:

if button.winfo\_exists(): # Check if the button still exists

button.configure(

fg\_color="transparent",

hover\_color="#34495e" # Set to default hover color

)

# Check if the specified button index is within the bounds of the nav\_buttons list

if len(self.nav\_buttons) > btn\_index:

self.active\_button = self.nav\_buttons[btn\_index]

if self.active\_button.winfo\_exists(): # Ensure the button exists

self.active\_button.configure(

fg\_color="grey", # Active button color

hover\_color="#16a085" # Active button hover color

)

self.style\_buttons() # Refresh the styles to apply changes

# profile.py

import customtkinter

import os

from PIL import Image, ImageTk

import shutil

from tkinter import filedialog, messagebox

import sqlite3

from datetime import datetime

import sqlite3

import bcrypt

import base64

class ProfileScreen:

def \_\_init\_\_(self, app, username, role):

self.app = app

self.username = username

self.role = role

# Profile Frame

self.profile\_frame = customtkinter.CTkFrame(self.app, corner\_radius=15, border\_width=3, border\_color="#c0392b")

self.profile\_frame.pack(side="right", fill="both", expand=True, padx=20, pady=20)

# Profile Header with Role

self.profile\_header\_label = customtkinter.CTkLabel(

self.profile\_frame,

text=f" {self.username} Role: {self.role}",

font=("Arial", 24, "bold"),

text\_color="#34495e",

)

self.profile\_header\_label.pack(pady=10)

# Content Frame for layout

content\_frame = customtkinter.CTkFrame(self.profile\_frame, corner\_radius=10)

content\_frame.pack(fill="both", expand=True, padx=10, pady=10)

# Left Section: Profile Photo and Work Hours

self.create\_left\_section(content\_frame)

# Right Section: Recent Activity and Updates

self.create\_right\_section(content\_frame)

def create\_left\_section(self, parent):

"""Left section with Profile Photo and Work Hours."""

left\_section = customtkinter.CTkFrame(parent, fg\_color="#ecf0f1")

left\_section.pack(side="left", fill="both", expand=False, padx=10, pady=10)

# Profile Photo

customtkinter.CTkLabel(

left\_section,

text="", # Ensure no text is set here

font=("Arial", 18, "bold"),

text\_color="#34495e",

).pack(pady=5)

image\_path = self.get\_profile\_photo\_path()

if os.path.exists(image\_path):

image = Image.open(image\_path).resize((150, 150))

self.profile\_photo = ImageTk.PhotoImage(image)

profile\_photo\_label = customtkinter.CTkLabel(left\_section, image=self.profile\_photo, text="")

else:

profile\_photo\_label = customtkinter.CTkLabel(

left\_section,

text="No Profile Photo", # Only set text if no photo is available

width=150,

height=150,

corner\_radius=15,

fg\_color="#dfe6e9",

)

profile\_photo\_label.pack(pady=5)

# Work Hours Buttons

customtkinter.CTkLabel(

left\_section,

text="Work Hours",

font=("Arial", 18, "bold"),

text\_color="#34495e",

).pack(pady=10)

clock\_in\_button = customtkinter.CTkButton(

left\_section,

text="Clock In",

command=self.clock\_in,

fg\_color="#2ECC71",

text\_color="#ffffff",

)

clock\_in\_button.pack(pady=5)

clock\_out\_button = customtkinter.CTkButton(

left\_section,

text="Clock Out",

command=self.clock\_out,

fg\_color="#E74C3C",

text\_color="#ffffff",

)

clock\_out\_button.pack(pady=5)

def create\_right\_section(self, parent):

"""Right section with Recent Activity and Updates."""

right\_section = customtkinter.CTkFrame(parent, fg\_color="#ffffff")

right\_section.pack(side="right", fill="both", expand=True, padx=10, pady=10)

# Recent Activity

customtkinter.CTkLabel(

right\_section,

text="Recent Activity",

font=("Arial", 18, "bold"),

text\_color="#34495e",

).pack(pady=5)

self.activity\_frame = customtkinter.CTkFrame(right\_section, fg\_color="#ffffff")

self.activity\_frame.pack(fill="x", expand=False, padx=5, pady=5)

self.load\_recent\_activity()

# Username and Password Updates

update\_frame = customtkinter.CTkFrame(right\_section, fg\_color="#f7f9fc")

update\_frame.pack(fill="x", padx=5, pady=10)

# Username Update Section

customtkinter.CTkLabel(

update\_frame,

text="Update Username",

font=("Arial", 18, "bold"),

text\_color="#34495e",

).grid(row=0, column=0, columnspan=2, pady=5)

self.new\_username\_entry = customtkinter.CTkEntry(

update\_frame,

placeholder\_text="Enter new username",

width=200,

corner\_radius=10,

)

self.new\_username\_entry.grid(row=1, column=0, padx=10, pady=5)

update\_username\_button = customtkinter.CTkButton(

update\_frame,

text="Update Username",

command=self.update\_username,

fg\_color="#1abc9c",

text\_color="#ffffff",

)

update\_username\_button.grid(row=1, column=1, padx=10, pady=5)

# Password Update Section

customtkinter.CTkLabel(

update\_frame,

text="Update Password",

font=("Arial", 18, "bold"),

text\_color="#34495e",

).grid(row=2, column=0, columnspan=2, pady=5)

self.old\_password\_entry = customtkinter.CTkEntry(

update\_frame,

placeholder\_text="Enter old password",

show="\*",

width=200,

corner\_radius=10,

)

self.old\_password\_entry.grid(row=3, column=0, padx=10, pady=5)

self.new\_password\_entry = customtkinter.CTkEntry(

update\_frame,

placeholder\_text="Enter new password",

show="\*",

width=200,

corner\_radius=10,

)

self.new\_password\_entry.grid(row=3, column=1, padx=10, pady=5)

self.confirm\_password\_entry = customtkinter.CTkEntry(

update\_frame,

placeholder\_text="Confirm new password",

show="\*",

width=200,

corner\_radius=10,

)

self.confirm\_password\_entry.grid(row=4, column=0, padx=10, pady=5)

update\_password\_button = customtkinter.CTkButton(

update\_frame,

text="Update Password",

command=self.update\_password,

fg\_color="#1abc9c",

text\_color="#ffffff",

)

update\_password\_button.grid(row=4, column=1, padx=10, pady=5)

def update\_password(self):

"""Handle updating the password."""

old\_password = self.old\_password\_entry.get()

new\_password = self.new\_password\_entry.get()

confirm\_password = self.confirm\_password\_entry.get()

if not old\_password or not new\_password or not confirm\_password:

messagebox.showerror("Error", "All password fields are required.")

return

if new\_password != confirm\_password:

messagebox.showerror("Error", "New passwords do not match.")

return

if len(new\_password) < 6:

messagebox.showerror("Error", "Password must be at least 6 characters long.")

return

conn = sqlite3.connect(r'db/Sales\_Inventory.db')

cursor = conn.cursor()

# Fetch the current hashed password

cursor.execute("SELECT password FROM users WHERE username = ?", (self.username,))

result = cursor.fetchone()

if not result:

messagebox.showerror("Error", "User not found.")

conn.close()

return

# Verify the old password

stored\_hashed\_password = base64.b64decode(result[0].encode('utf-8'))

if not bcrypt.checkpw(old\_password.encode('utf-8'), stored\_hashed\_password):

messagebox.showerror("Error", "Old password is incorrect.")

conn.close()

return

# Hash the new password

hashed\_password = bcrypt.hashpw(new\_password.encode('utf-8'), bcrypt.gensalt())

hashed\_password\_base64 = base64.b64encode(hashed\_password).decode('utf-8')

# Update the password in the database

cursor.execute("UPDATE users SET password = ? WHERE username = ?", (hashed\_password\_base64, self.username))

conn.commit()

conn.close()

messagebox.showinfo("Success", "Password updated successfully.")

def get\_profile\_photo\_path(self):

"""Path where the profile photo is stored."""

return os.path.join("../static/images", f"{self.username}\_profile\_photo.png")

def update\_username(self):

"""Handle updating the username."""

new\_username = self.new\_username\_entry.get()

if not new\_username:

messagebox.showerror("Error", "Username cannot be empty.")

return

conn = sqlite3.connect(r'db/Sales\_Inventory.db')

cursor = conn.cursor()

# Check if the username is already taken

cursor.execute("SELECT \* FROM users WHERE username = ?", (new\_username,))

if cursor.fetchone():

messagebox.showerror("Error", "Username already taken.")

conn.close()

return

# Update the username in the database

cursor.execute("UPDATE users SET username = ? WHERE username = ?", (new\_username, self.username))

conn.commit()

conn.close()

messagebox.showinfo("Success", "Username updated successfully.")

self.username = new\_username # Update the current username

self.app.current\_user = new\_username

# Refresh the profile header to show the updated username

self.profile\_header\_label.configure(text=f"Profile: {self.username} (Role: {self.role})")

def get\_profile\_photo\_path(self):

"""Path where the profile photo is stored."""

return os.path.join("../static/images", f"{self.username}\_profile\_photo.png")

def clock\_in(self):

"""Log the user as clocked in."""

self.record\_action("Clocked In")

conn = sqlite3.connect(r'db/Sales\_Inventory.db')

cursor = conn.cursor()

cursor.execute("INSERT INTO clock\_times (username, clock\_in) VALUES (?, ?)", (self.username, datetime.now()))

conn.commit()

conn.close()

messagebox.showinfo("Clock In", "You have successfully clocked in.")

def clock\_out(self):

"""Log the user as clocked out."""

self.record\_action("Clocked Out")

conn = sqlite3.connect(r'db/Sales\_Inventory.db')

cursor = conn.cursor()

cursor.execute(

"UPDATE clock\_times SET clock\_out = ? WHERE username = ? AND clock\_out IS NULL",

(datetime.now(), self.username),

)

conn.commit()

conn.close()

messagebox.showinfo("Clock Out", "You have successfully clocked out.")

def record\_action(self, action):

"""Record the user's action in the activity log."""

conn = sqlite3.connect(r'db/Sales\_Inventory.db')

cursor = conn.cursor()

cursor.execute("INSERT INTO user\_activity (username, action) VALUES (?, ?)", (self.username, action))

conn.commit()

conn.close()

self.load\_recent\_activity()

def load\_recent\_activity(self):

"""Load and display the recent activities of the user."""

for widget in self.activity\_frame.winfo\_children():

widget.destroy()

conn = sqlite3.connect(r'db/Sales\_Inventory.db')

cursor = conn.cursor()

cursor.execute(

"SELECT action, timestamp FROM user\_activity WHERE username = ? ORDER BY timestamp DESC LIMIT 5",

(self.username,),

)

activities = cursor.fetchall()

conn.close()

for action, timestamp in activities:

customtkinter.CTkLabel(

self.activity\_frame,

text=f"{timestamp}: {action}",

font=("Arial", 14),

text\_color="#34495e",

).pack(anchor="w", pady=2)

class SettingsScreen:

def \_\_init\_\_(self, app, user\_role, backend):

self.app = app

self.user\_role = user\_role

self.backend = backend

# Settings Frame

self.settings\_frame = customtkinter.CTkFrame(self.app, corner\_radius=15, border\_width=3, border\_color="#1abc9c")

self.settings\_frame.pack(side="right", fill="both", expand=True, padx=20, pady=20)

# Settings Header

settings\_label = customtkinter.CTkLabel(self.settings\_frame, text="Settings", font=("Arial", 24, "bold"), text\_color="#34495e")

settings\_label.pack(pady=20)

# Theme Change Dropdown

customtkinter.CTkLabel(self.settings\_frame, text="Select Theme:", text\_color="#34495e").pack(pady=5)

theme\_option\_menu = customtkinter.CTkOptionMenu(

self.settings\_frame, values=["Light", "Dark", "Blue"], command=self.change\_theme,

fg\_color="#1abc9c", button\_color="#1abc9c", text\_color="#ffffff"

)

theme\_option\_menu.pack(pady=10)

# Fullscreen Toggle Button

fullscreen\_button = customtkinter.CTkButton(

self.settings\_frame, text="Toggle Fullscreen", command=self.toggle\_fullscreen, width=160, height=50,

corner\_radius=10, fg\_color="#1abc9c", hover\_color="#16a085", text\_color="#ffffff"

)

fullscreen\_button.pack(pady=10)

# Upload Profile Photo Button

upload\_photo\_button = customtkinter.CTkButton(

self.settings\_frame, text="Upload Profile Photo", command=self.upload\_photo, width=160, height=50,

corner\_radius=10, fg\_color="#1abc9c", hover\_color="#16a085", text\_color="#ffffff"

)

upload\_photo\_button.pack(pady=20)

# Terminate User Section (Only for Supervisors)

if self.user\_role == 'Supervisor':

terminate\_button = customtkinter.CTkButton(

self.settings\_frame, text="Terminate User", command=self.show\_terminate\_screen, width=160, height=50,

corner\_radius=10, fg\_color="#FF4C4C", hover\_color="#FF1C1C", text\_color="#ffffff"

)

terminate\_button.pack(pady=10, fill="x")

# Log Out Button with Audio

logout\_button = customtkinter.CTkButton(

self.settings\_frame, text="Log Out", command=self.logout\_with\_audio, width=160, height=50,

corner\_radius=10, fg\_color="#FF4C4C", hover\_color="#FF1C1C", text\_color="#ffffff"

)

logout\_button.pack(pady=10, fill="x")

# Close App Button with Audio

close\_app\_button = customtkinter.CTkButton(

self.settings\_frame, text="Close App", command=self.close\_app\_with\_audio, width=160, height=50,

corner\_radius=10, fg\_color="#FF4C4C", hover\_color="#FF1C1C", text\_color="#ffffff"

)

close\_app\_button.pack(pady=10, fill="x")

def change\_theme(self, theme\_name):

# Apply the selected theme

customtkinter.set\_appearance\_mode(theme\_name.lower())

def toggle\_fullscreen(self):

self.app.is\_full\_screen = not self.app.is\_full\_screen

self.app.attributes("-fullscreen", self.app.is\_full\_screen)

def upload\_photo(self):

"""Upload a profile photo and save it for the current user."""

file\_path = filedialog.askopenfilename(

title="Select a Profile Photo",

filetypes=[("Image files", "\*.jpg \*.jpeg \*.png \*.bmp \*.gif")]

)

if file\_path:

try:

# Create 'images' directory if it doesn't exist

if not os.path.exists("../static/images"):

os.makedirs("../static/images")

# Copy the selected photo to the 'images' directory with a specific naming convention

dest\_path = os.path.join("../static/images", f"{self.app.current\_user}\_profile\_photo.png")

shutil.copy(file\_path, dest\_path)

messagebox.showinfo("Upload Photo", f"Photo successfully uploaded for {self.app.current\_user}.")

except Exception as e:

messagebox.showerror("Upload Error", f"Failed to upload photo: {e}")

else:

messagebox.showwarning("Upload Photo", "No photo was selected.")

def show\_terminate\_screen(self):

# Create the terminate user screen

terminate\_window = customtkinter.CTkToplevel(self.app)

terminate\_window.title("Terminate User")

terminate\_window.geometry("400x400")

# Header

terminate\_label = customtkinter.CTkLabel(terminate\_window, text="Terminate User", font=("Arial", 20, "bold"),

fg\_color="#FF4C4C", text\_color="#ffffff")

terminate\_label.pack(pady=20)

# List of users to select from

user\_list = self.backend.get\_all\_users() # Get all users from backend

self.selected\_user = customtkinter.StringVar()

self

# report\_generator.py

import sqlite3

import os

import csv

from datetime import datetime

class ReportGenerator:

def \_\_init\_\_(self):

self.db\_path = r'db/Sales\_Inventory.db'

self.conn = None

self.cursor = None

def connect(self):

self.conn = sqlite3.connect(self.db\_path)

self.cursor = self.conn.cursor()

def disconnect(self):

if self.conn:

self.conn.close()

def generate\_sales\_report(self, start\_date=None, end\_date=None, output\_file='sales\_report.csv'):

"""Generate a sales report for a specified date range and save it as a CSV file."""

self.connect()

try:

query = '''

SELECT sales.sale\_id, sales.date, sales.quantity, users.user\_id, items.item\_name, items.price

FROM sales

JOIN users ON sales.user\_id = users.user\_id

JOIN items ON sales.item\_id = items.item\_id

'''

params = []

if start\_date and end\_date:

query += ' WHERE date\_sale BETWEEN ? AND ?'

params.extend([start\_date, end\_date])

self.cursor.execute(query, params)

sales\_data = self.cursor.fetchall()

# Write data to CSV

with open(output\_file, 'w', newline='') as csvfile:

fieldnames = ['Sale ID', 'User', 'Customer', 'Grand Total', 'Date of Sale']

writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

writer.writeheader()

for sale in sales\_data:

writer.writerow({

'Sale ID': sale[0],

'User': sale[1],

'Customer': sale[2],

'Grand Total': sale[3],

'Date of Sale': sale[4]

})

print(f"Sales report generated: {output\_file}")

except sqlite3.Error as e:

print(f"Error generating sales report: {e}")

finally:

self.disconnect()

def generate\_inventory\_report(self, output\_file='inventory\_report.csv'):

"""Generate an inventory report and save it as a CSV file."""

self.connect()

try:

self.cursor.execute('SELECT \* FROM items')

items\_data = self.cursor.fetchall()

# Write data to CSV

with open(output\_file, 'w', newline='') as csvfile:

fieldnames = ['Item ID', 'Product', 'Size', 'Quantity', 'Price', 'Date Added']

writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

writer.writeheader()

for item in items\_data:

writer.writerow({

'Item ID': item[0],

'Product': item[1],

'Size': item[2],

'Quantity': item[3],

'Price': item[4],

'Date Added': item[5]

})

print(f"Inventory report generated: {output\_file}")

except sqlite3.Error as e:

print(f"Error generating inventory report: {e}")

finally:

self.disconnect()

def generate\_customer\_report(self, output\_file='customer\_report.csv'):

"""Generate a customer report and save it as a CSV file."""

self.connect()

try:

self.cursor.execute('SELECT \* FROM customers')

customers\_data = self.cursor.fetchall()

# Write data to CSV

with open(output\_file, 'w', newline='') as csvfile:

fieldnames = ['Customer ID', 'Customer Name', 'Address', 'Status', 'Notes', 'Date Order']

writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

writer.writeheader()

for customer in customers\_data:

writer.writerow({

'Customer ID': customer[0],

'Customer Name': customer[1],

'Address': customer[2],

'Status': customer[3],

'Notes': customer[4],

'Date Order': customer[5]

})

print(f"Customer report generated: {output\_file}")

except sqlite3.Error as e:

print(f"Error generating customer report: {e}")

finally:

self.disconnect()

# sales.py

import sqlite3

import os

class Sales:

def \_\_init\_\_(self):

self.db\_path = 'db/Sales\_Inventory.db'

self.conn = None

self.cursor = None

def connect(self):

self.conn = sqlite3.connect(self.db\_path)

self.cursor = self.conn.cursor()

def disconnect(self):

if self.conn:

self.conn.close()

def create\_sales(self, user\_id, customer\_id, items):

"""Create a new sale in the sales table and corresponding items in sales\_items."""

self.connect()

try:

# Insert new sale record

self.cursor.execute(

'''INSERT INTO sales (user\_ID, customer\_ID, grand\_total) VALUES (?, ?, ?)''',

(user\_id, customer\_id, sum(item['price'] \* item['quantity'] for item in items))

)

sale\_id = self.cursor.lastrowid

# Insert each item into sales\_items table

for item in items:

self.cursor.execute(

'''INSERT INTO sales\_items (sale\_ID, item\_ID, quantity, price) VALUES (?, ?, ?, ?)''',

(sale\_id, item['item\_id'], item['quantity'], item['price'])

)

self.conn.commit()

except sqlite3.Error as e:

print(f"Error creating sale: {e}")

finally:

self.disconnect()

def get\_sales(self, sale\_id=None):

"""Fetch sales data, optionally filtered by sale ID."""

self.connect()

try:

if sale\_id:

self.cursor.execute('SELECT \* FROM sales WHERE sale\_ID = ?', (sale\_id,))

return self.cursor.fetchone()

else:

self.cursor.execute('SELECT \* FROM sales')

return self.cursor.fetchall()

except sqlite3.Error as e:

print(f"Error fetching sales: {e}")

return []

finally:

self.disconnect()

def get\_sales\_items(self, sale\_id):

"""Fetch all items related to a specific sale."""

self.connect()

try:

self.cursor.execute('SELECT \* FROM sales\_items WHERE sale\_ID = ?', (sale\_id,))

return self.cursor.fetchall()

except sqlite3.Error as e:

print(f"Error fetching sales items for sale ID {sale\_id}: {e}")

return []

finally:

self.disconnect()

def delete\_sale(self, sale\_id):

"""Delete a sale and corresponding items from the database."""

self.connect()

try:

# Delete items related to the sale

self.cursor.execute('DELETE FROM sales\_items WHERE sale\_ID = ?', (sale\_id,))

# Delete the sale

self.cursor.execute('DELETE FROM sales WHERE sale\_ID = ?', (sale\_id,))

self.conn.commit()

except sqlite3.Error as e:

print(f"Error deleting sale ID {sale\_id}: {e}")

finally:

self.disconnect()

import sqlite3

import os

class SalesHandler:

def \_\_init\_\_(self):

self.db\_path = 'db/Sales\_Inventory.db'

self.conn = None

self.cursor = None

def connect(self):

self.conn = sqlite3.connect(self.db\_path)

self.cursor = self.conn.cursor()

def disconnect(self):

if self.conn:

self.conn.close()

def create\_sale(self, user\_id, customer\_id, items):

"""Create a new sale in the sales table and corresponding items in sales\_items."""

self.connect()

try:

# Insert new sale record

self.cursor.execute(

'''INSERT INTO sales (user\_ID, customer\_ID, grand\_total) VALUES (?, ?, ?)''',

(user\_id, customer\_id, sum(item['price'] \* item['quantity'] for item in items))

)

sale\_id = self.cursor.lastrowid

# Insert each item into sales\_items table

for item in items:

self.cursor.execute(

'''INSERT INTO sales\_items (sale\_ID, item\_ID, quantity, price) VALUES (?, ?, ?, ?)''',

(sale\_id, item['item\_id'], item['quantity'], item['price'])

)

self.conn.commit()

except sqlite3.Error as e:

print(f"Error creating sale: {e}")

finally:

self.disconnect()

def get\_sales(self, sale\_id=None):

"""Fetch sales data, optionally filtered by sale ID."""

self.connect()

try:

if sale\_id:

self.cursor.execute('SELECT \* FROM sales WHERE sale\_ID = ?', (sale\_id,))

return self.cursor.fetchone()

else:

self.cursor.execute('SELECT \* FROM sales')

return self.cursor.fetchall()

except sqlite3.Error as e:

print(f"Error fetching sales: {e}")

return []

finally:

self.disconnect()

def get\_sales\_items(self, sale\_id):

"""Fetch all items related to a specific sale."""

self.connect()

try:

self.cursor.execute('SELECT \* FROM sales\_items WHERE sale\_ID = ?', (sale\_id,))

return self.cursor.fetchall()

except sqlite3.Error as e:

print(f"Error fetching sales items for sale ID {sale\_id}: {e}")

return []

finally:

self.disconnect()

def delete\_sale(self, sale\_id):

"""Delete a sale and corresponding items from the database."""

self.connect()

try:

# Delete items related to the sale

self.cursor.execute('DELETE FROM sales\_items WHERE sale\_ID = ?', (sale\_id,))

# Delete the sale

self.cursor.execute('DELETE FROM sales WHERE sale\_ID = ?', (sale\_id,))

self.conn.commit()

except sqlite3.Error as e:

print(f"Error deleting sale ID {sale\_id}: {e}")

finally:

self.disconnect()

# settings.py

import customtkinter

import tkinter as tk

from tkinter import ttk

import tkinter.ttk as ttk

import sqlite3

import datetime

from tkcalendar import Calendar

import bcrypt

import base64

from tkinter import filedialog

import os

import shutil

from tkinter import messagebox

class SettingsScreen:

def \_\_init\_\_(self, app, user\_role, backend):

self.app = app

self.user\_role = user\_role

self.backend = backend

# Settings Frame

self.settings\_frame = customtkinter.CTkFrame(

self.app, corner\_radius=20, border\_width=2, border\_color="#2ECC71", fg\_color="#F8F9FA"

)

self.settings\_frame.pack(side="right", fill="both", expand=True, padx=20, pady=20)

# Settings Header

settings\_label = customtkinter.CTkLabel(

self.settings\_frame, text="Settings", font=("Arial", 26, "bold"), text\_color="#2C3E50"

)

settings\_label.pack(pady=20)

# Profile Settings Section

self.add\_section\_label("Profile Settings")

upload\_photo\_button = customtkinter.CTkButton(

self.settings\_frame, text="Upload Profile Photo", command=self.upload\_photo, width=200, height=50,

corner\_radius=10, fg\_color="#3498DB", hover\_color="#2980B9", text\_color="#FFFFFF"

)

upload\_photo\_button.pack(pady=10)

# Application Actions Section

self.add\_section\_label("Application Actions")

logout\_button = customtkinter.CTkButton(

self.settings\_frame, text="Log Out", command=self.logout\_with\_audio, width=200, height=50,

corner\_radius=10, fg\_color="#E67E22", hover\_color="#D35400", text\_color="#FFFFFF"

)

logout\_button.pack(pady=10)

close\_app\_button = customtkinter.CTkButton(

self.settings\_frame, text="Close App", command=self.close\_app\_with\_audio, width=200, height=50,

corner\_radius=10, fg\_color="#34495E", hover\_color="#2C3E50", text\_color="#FFFFFF"

)

close\_app\_button.pack(pady=10)

# Supervisor-Specific Actions

if self.user\_role == "Supervisor":

self.add\_section\_label("Supervisor Actions")

self.add\_clear\_db\_button()

def add\_section\_label(self, text):

"""Helper to add section headers."""

section\_label = customtkinter.CTkLabel(

self.settings\_frame, text=text, font=("Arial", 18, "bold"), text\_color="#34495E"

)

section\_label.pack(pady=15)

def add\_clear\_db\_button(self):

"""Add the CLEAR DB button for supervisors."""

clear\_db\_button = customtkinter.CTkButton(

self.settings\_frame,

text="CLEAR DB",

command=self.confirm\_clear\_db,

width=200,

height=50,

corner\_radius=10,

fg\_color="#F1C40F", # Yellow

hover\_color="#F39C12",

text\_color="#E74C3C", # Red text

border\_width=2,

border\_color="#000000", # Black border

)

clear\_db\_button.pack(pady=20)

def confirm\_clear\_db(self):

"""Popup confirmation before clearing the database."""

confirm = messagebox.askyesno(

"Confirm Clear DB",

"Are you sure you want to clear the database? This action cannot be undone.",

)

if confirm:

self.clear\_database()

def clear\_database(self):

"""Clear all tables in the database and recreate them."""

try:

# Ensure the database connection is open

self.backend.connect()

# Drop all existing tables

self.backend.cursor.execute("DROP TABLE IF EXISTS users")

self.backend.cursor.execute("DROP TABLE IF EXISTS InventoryItem")

self.backend.cursor.execute("DROP TABLE IF EXISTS Customers")

self.backend.cursor.execute("DROP TABLE IF EXISTS clock\_items")

self.backend.cursor.execute("DROP TABLE IF EXISTS user\_activity")

self.backend.cursor.execute("DROP TABLE IF EXISTS Orders")

self.backend.conn.commit()

# Recreate the database schema

self.backend.create\_database()

# Hash and encode the default password

default\_password = "TESTING"

hashed\_password = bcrypt.hashpw(default\_password.encode('utf-8'), bcrypt.gensalt())

hashed\_password\_base64 = base64.b64encode(hashed\_password).decode('utf-8')

# Add a default user account

self.backend.cursor.execute(

"INSERT INTO users (username, password, role) VALUES (?, ?, ?)",

("Admin", hashed\_password\_base64, "Supervisor")

)

self.backend.conn.commit()

# Show success message

messagebox.showinfo("Success", "The database has been cleared and recreated successfully.")

except Exception as e:

# Show error message

messagebox.showerror("Error", f"Failed to clear the database: {e}")

finally:

# Ensure the connection is closed even if an error occurs

if self.backend.conn:

self.backend.disconnect()

def upload\_photo(self):

"""Upload a profile photo and save it for the current user."""

file\_path = filedialog.askopenfilename(

title="Select a Profile Photo",

filetypes=[("Image files", "\*.jpg \*.jpeg \*.png \*.bmp \*.gif")]

)

if file\_path:

try:

if not os.path.exists("../static/images"):

os.makedirs("../static/images")

dest\_path = os.path.join("../static/images", f"{self.app.current\_user}\_profile\_photo.png")

shutil.copy(file\_path, dest\_path)

messagebox.showinfo("Upload Photo", f"Photo successfully uploaded for {self.app.current\_user}.")

except Exception as e:

messagebox.showerror("Upload Error", f"Failed to upload photo: {e}")

else:

messagebox.showwarning("Upload Photo", "No photo was selected.")

def logout\_with\_audio(self):

try:

pygame.mixer.music.load(r'static/audio/logout.mp3')

pygame.mixer.music.play()

except Exception as e:

print(f"Error playing logout audio: {e}")

finally:

self.app.show\_login\_screen()

def close\_app\_with\_audio(self):

try:

pygame.mixer.music.load(r'static/audio/logout.mp3')

pygame.mixer.music.play()

except Exception as e:

print(f"Error playing close app audio: {e}")

finally:

self.app.quit()

# terminate\_user.py

import customtkinter

from tkinter import messagebox

class TerminateUserScreen:

def \_\_init\_\_(self, app, backend):

self.app = app

self.backend = backend

# Create the terminate user screen as a Toplevel window

self.terminate\_window = customtkinter.CTkToplevel(app)

self.terminate\_window.title("Terminate User")

self.terminate\_window.geometry("400x400")

self.terminate\_window.grab\_set() # Keep window in focus until closed

# Title

title\_label = customtkinter.CTkLabel(self.terminate\_window, text="Terminate User", font=("Arial", 24, "bold"))

title\_label.pack(pady=20)

# List of users

self.users = self.backend.get\_all\_users()

if not self.users:

# If no users are available, display a message

no\_users\_label = customtkinter.CTkLabel(self.terminate\_window, text="No users available.")

no\_users\_label.pack(pady=20)

return

# User selection dropdown

self.selected\_user = customtkinter.StringVar()

self.user\_dropdown = customtkinter.CTkOptionMenu(self.terminate\_window, variable=self.selected\_user, values=self.users)

self.user\_dropdown.pack(pady=20)

# Terminate button

terminate\_button = customtkinter.CTkButton(self.terminate\_window, text="Terminate", command=self.terminate\_user, fg\_color="red", hover\_color="darkred")

terminate\_button.pack(pady=20)

def terminate\_user(self):

# Confirmation prompt

if not self.selected\_user.get():

messagebox.showwarning("Error", "Please select a user.")

return

confirm = messagebox.askyesno("Confirm Termination", f"Are you sure you want to terminate the user '{self.selected\_user.get()}'?")

if confirm:

self.backend.delete\_user(self.selected\_user.get())

messagebox.showinfo("Terminated", f"User '{self.selected\_user.get()}' has been terminated.")

self.terminate\_window.destroy()

# user\_manual.py

import tkinter as tk

from tkinter import ttk

import customtkinter

class UserManual:

def \_\_init\_\_(self, parent):

# Create a separate Toplevel window for the user manual

self.window = tk.Toplevel(parent)

self.window.title("User Manual")

self.window.geometry("700x800")

self.window.configure(bg="#ffffff") # White background

self.window.resizable(False, False) # Disable resizing

self.window.overrideredirect(True) # Frameless window

# Enable dragging the window

self.window.bind("<ButtonPress-1>", self.start\_drag)

self.window.bind("<B1-Motion>", self.drag\_window)

# Outer container

self.container = customtkinter.CTkFrame(

self.window,

corner\_radius=15,

fg\_color="#ffffff"

)

self.container.pack(fill="both", expand=True, padx=20, pady=20)

# Header

self.create\_header()

# Scrollable content area

self.scroll\_canvas = tk.Canvas(self.container, bg="#ffffff", highlightthickness=0)

self.scroll\_canvas.pack(side="left", fill="both", expand=True)

self.scrollbar = ttk.Scrollbar(

self.container,

orient="vertical",

command=self.scroll\_canvas.yview

)

self.scrollbar.pack(side="right", fill="y")

self.scroll\_canvas.configure(yscrollcommand=self.scrollbar.set)

self.scroll\_frame = customtkinter.CTkFrame(self.scroll\_canvas, fg\_color="#ffffff")

self.scrollable\_window = self.scroll\_canvas.create\_window(

(0, 0),

window=self.scroll\_frame,

anchor="nw"

)

# Populate content

self.populate\_content()

# Update scrollable region

self.scroll\_frame.update\_idletasks()

self.scroll\_canvas.configure(scrollregion=self.scroll\_canvas.bbox("all"))

self.scroll\_canvas.bind\_all("<MouseWheel>", self.on\_mousewheel)

def create\_header(self):

"""Create the header with a close button."""

header\_frame = customtkinter.CTkFrame(self.container, fg\_color="#C0392B")

header\_frame.pack(fill="x", pady=(0, 10))

header\_label = customtkinter.CTkLabel(

header\_frame,

text="\ud83d\udcd8 User Manual",

font=("Arial", 20, "bold"),

text\_color="#ffffff"

)

header\_label.pack(side="left", padx=10)

close\_button = customtkinter.CTkButton(

header\_frame,

text="X",

fg\_color="#E74C3C",

hover\_color="#C0392B",

text\_color="white",

command=self.window.destroy,

width=40,

height=40,

corner\_radius=20

)

close\_button.pack(side="right", padx=10)

def populate\_content(self):

"""Add content sections to the manual."""

sections = [

("📖 Introduction", "Welcome to the Sales Inventory App!\n This manual will guide you through the\n features and functionality of the application."),

("✨ Features", "- 📦 \*\*Manage Inventory\*\*: Organize and\n track your inventory.\n- 📊 \*\*Generate Reports\*\*: Create detailed\n sales and inventory reports.\n- ⚙️ \*\*Configure Settings\*\*: Customize your experience.\n- 👥 \*\*Manage Users\*\*: Add, edit, or remove users."),

("🔑 Default Login", "The default login for the application is:\n- \*\*Username\*\*: Admin\n- \*\*Password\*\*: TESTING"),

("👩‍💼 Role of Supervisors", "Supervisors play a critical role\n in managing the system:\n- Assign roles to users.\n- Ensure proper user \naccess control.\n- Perform critical actions such as \nclearing the database or managing permissions."),

("📋 How to Use", "1. Log in using your credentials.\n2. Use the navigation pane to switch between features.\n3. Follow on-screen prompts to manage\n inventory, generate reports, or modify settings."),

("❓ Help with Logging In", "If you have trouble logging in:\n- Ensure your username and password are correct.\n- Contact support if you encounter persistent issues."),

("📞 Contact Support", "☎️ Phone: +123 456 7890\n📧 Email: support@example.com\n🌐 Website: www.salesinventoryapp.com")

]

for title, content in sections:

title\_label = customtkinter.CTkLabel(

self.scroll\_frame,

text=title,

font=("Arial", 18, "bold"),

text\_color="#C0392B",

wraplength=660 # Dynamically wrap text based on the frame width

)

title\_label.pack(pady=(10, 5), anchor="w")

content\_label = customtkinter.CTkLabel(

self.scroll\_frame,

text=content,

font=("Arial", 14),

text\_color="gray",

wraplength=660 # Dynamically wrap text based on the frame width

)

content\_label.pack(pady=(0, 10), anchor="w")

def start\_drag(self, event):

"""Start dragging the window."""

self.start\_x = event.x

self.start\_y = event.y

def drag\_window(self, event):

"""Drag the window to a new position."""

x = self.window.winfo\_x() - self.start\_x + event.x

y = self.window.winfo\_y() - self.start\_y + event.y

self.window.geometry(f"+{x}+{y}")

def on\_mousewheel(self, event):

"""Scroll the canvas on mouse wheel movement."""

self.scroll\_canvas.yview\_scroll(-1 \* (event.delta // 120), "units")

if \_\_name\_\_ == "\_\_main\_\_":

root = tk.Tk()

root.withdraw() # Hide the root window

user\_manual = UserManual(root)

root.mainloop()