**Restaurant Management System - Project Documentation**

**1. System Requirements**

* **Programming Language:** Python 3.x
* **Editor/IDE:** Visual studio code (VS Code)
* **Dependencies:** maskpass and (standard Python libraries)
* **Platform:** Local JSON file-based storage, so it runs locally on any machine with Python installed.

**2. Project Overview**

* **Project Name:** Restaurant Management System
* **Description:** A Python-based system that manages restaurant reservations, order placements, menu viewing, and invoices, with data stored in JSON files instead of a database.
* **Version:** 1.0
* **Github :** https://github.com/mdmansooralam/Indixpert-restaurant-management-
* **Authors:** Mazda, Shifa, Rahul, Mansoor (FSWD April batch-1 2024 – 2025 ,Team 1)

**3. Project Description**

* **Purpose:**   
  The Restaurant Management System provides efficient and straightforward management of reservations, orders, and staff functions in a restaurant environment using JSON files for data storage.
* **Key Features:**
  + **Staff Login:** Staff can log in and access functionalities for booking tables, placing and updating orders, cancelling reservations, viewing the menu, viewing reservations, checking invoices, viewing own profile and updating and logout.
  + **Admin Login:** Admins can manage menu items, view and add stock, manage orders, view order report, manage staff profile, view error logs, view own profile and update and logout.
  + **Super Admin Login:** Super Admins have full control, including managing staff and admin accounts and viewing all operational data.

**4. Functional Requirements**

**4.1 User Roles and Access**

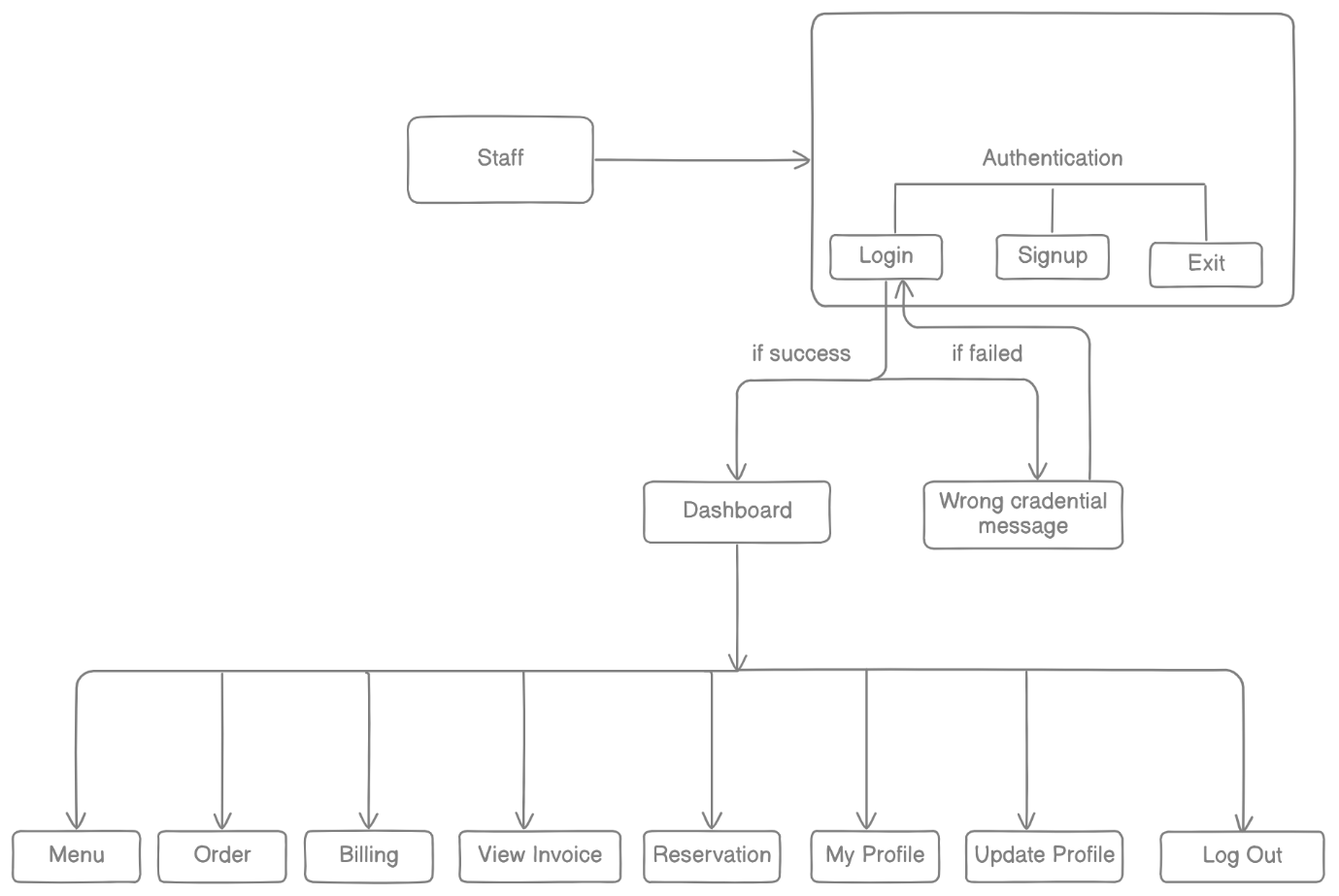
* **Staff**
  + book tables and manage reservations.
  + place, update and view ongoing orders.
  + view the menu and all reservations.
  + cancel reservations and view invoices.
  + view own profile and update.
* **Admin**
  + add, update and delete menu item.
  + view and add stock.
  + view all orders, view order details and cancel.
  + view order report date, day, and staff wise
  + view all staff, view staff profile, and remove staff.
  + view all error, search by email, and search by date.
  + view own profile and update.
* **Super Admin**
  + view all users.
  + make admin.
  + make staff.
  + remove staff or admin.

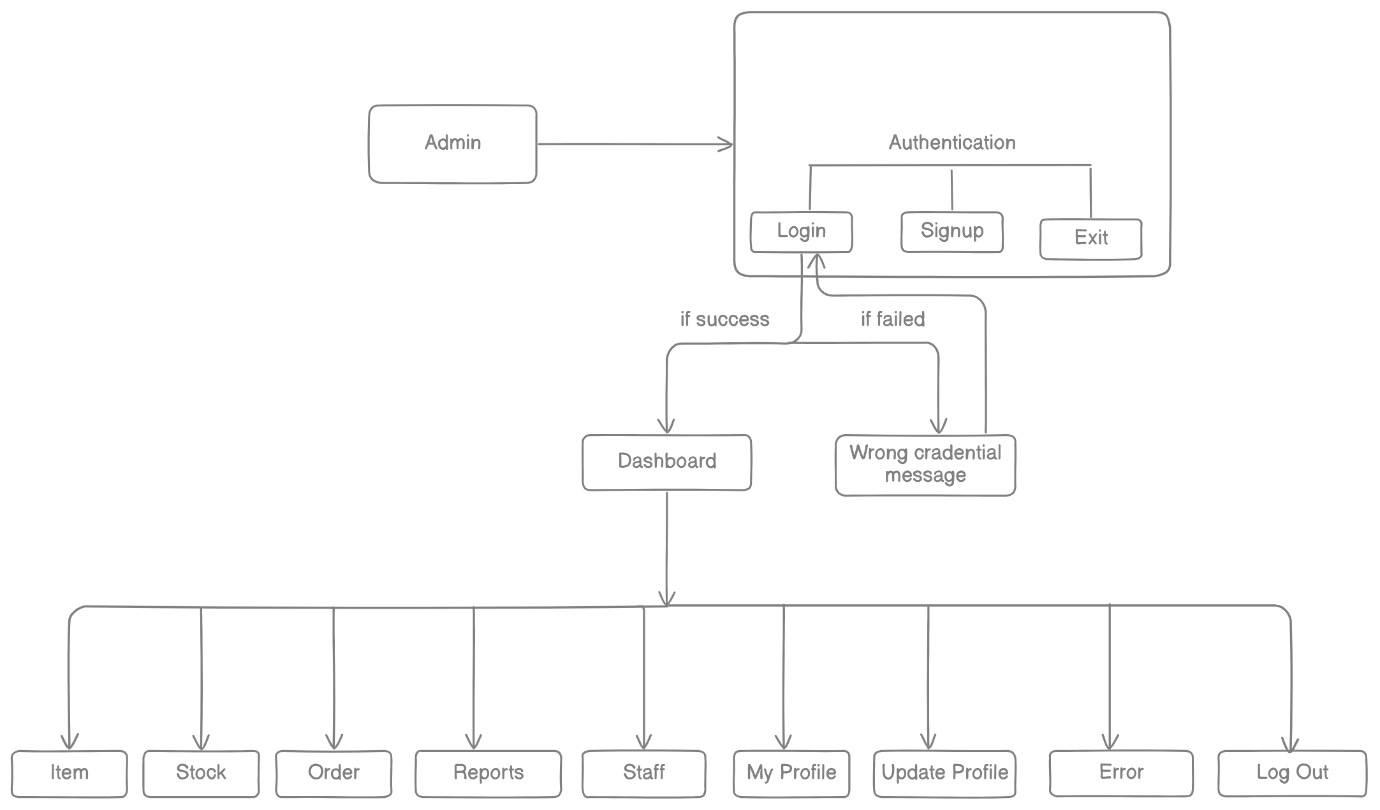
**4.2 Core Functions**

* **Table Reservation:**  
  Staff can book tables for customers and manage these reservations. Reservation details are stored in a JSON file.
* **Order Management:**  
  Staff can place, and update, orders. Orders are logged in a JSON file with details like items, quantities, and statuses etc.
* **Menu Viewing:**   
  Staff can view the restaurant menu, which is stored in a JSON file that can be updated as needed.
* **View Invoice:**  
  Staff can view invoices saved as JSON files.

**5. Project Architecture**

Staff login diagram

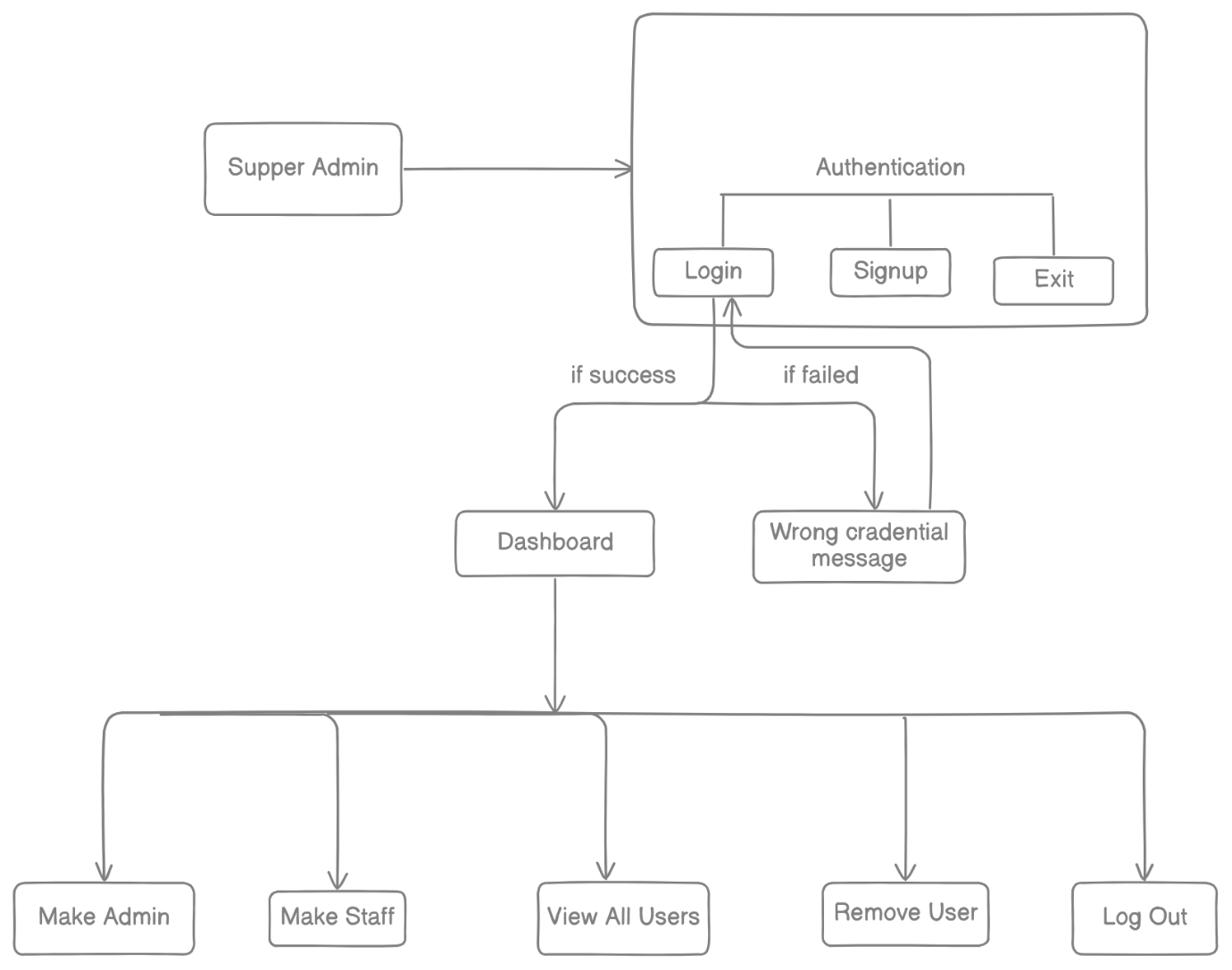
****

****

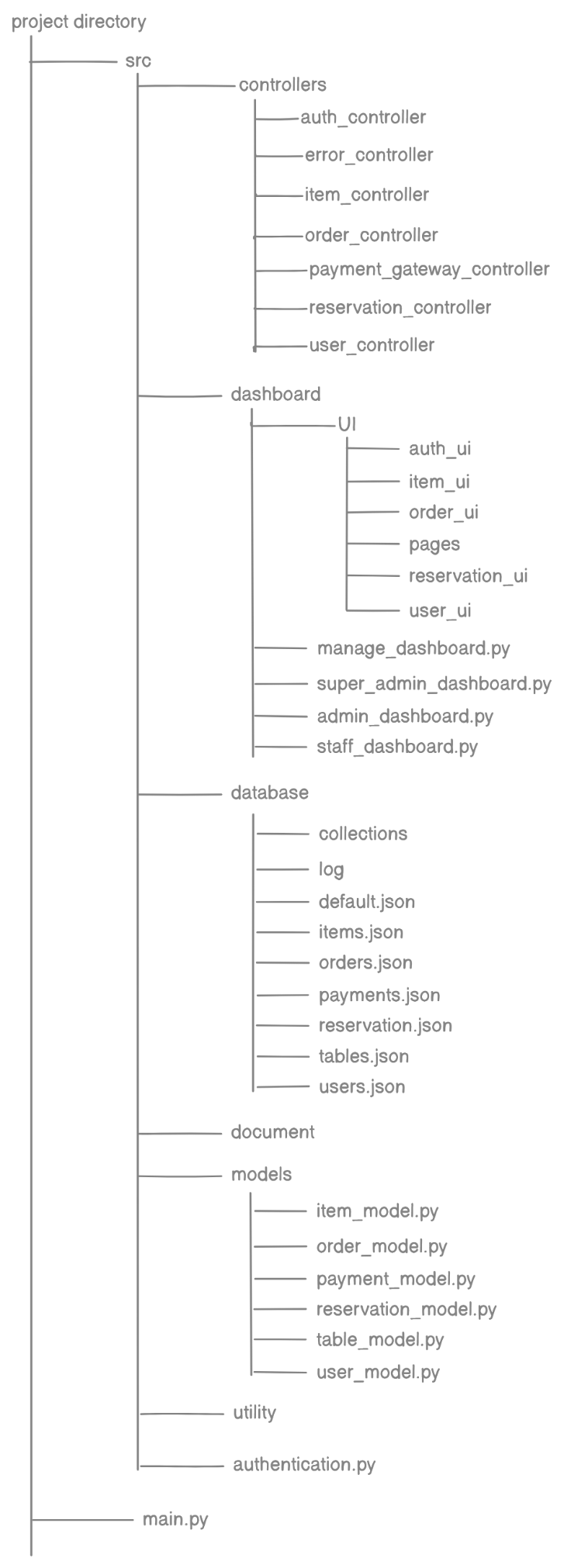
Admin login diagram

**sa**

Super Admin login diagram

****

**5.1 File system Structure**

****

**5.2 Component Descriptions**

* **Auth Module (auth.py):**   
  Manages login for different roles (staff, admin, super admin) with access control logic.
* **Reservation Module (reservation.py):**  
  Handles reservation-related functionalities, including table booking, viewing reservations, and cancellation.
* **Order Module (order.py):**   
  Manages order placement, updates, and order cancellations.
* **Item Module (item.py):**   
  Provide menu access and allows menu viewing and manage menu items like add, update and delete.
* **User Module (user.py):**   
  Provides user access to update, remove and viewing all users with access control logic.

**6. Usage Guide**

**6.1 System Setup**

1. Ensure Python 3.x is installed.
2. Clone or download the project files into your local directory.
3. pip install maskpass
4. Run main.py to start the application.

**6.2 User Login**

* The application prompts users to enter their login credentials.
* Access rights are determined based on the role (staff, admin, or super admin).

**6.3 Staff Functionality**

* **Book Table:**   
  Staff can book tables by specifying customer details and reservation time. Reservations are saved in reservations.json.
* **Place/Update Order:**  
  Staff can place new orders and update existing ones, with data stored in orders.json.
* **Cancel Reservation:**  
  Staff can cancel a reservation, which updates reservations.json.
* **View Invoices:**  
  Staff can view invoices stored as JSON files in the /database/ folder.

**6.4 Admin and Super Admin Functionality**

* **Manage Users:**  
  Admins and super admins can manage staff accounts and review order details.
* **Access All Data:**  
  Super admins have unrestricted access to all files and functionalities in the system.

**7. JSON File Operations**

**7.1 Data Storage**

* **item.json**: Stores the menu in a structured JSON format, allowing easy addition, deletion, and updating of menu items.
* **reservations.json**: Stores each reservation as a JSON object with fields for customer details, reservation time, and status.
* **orders.json**: Contains orders, each represented as a JSON object with item names, quantities, and prices.
* **Payment.json**: Contains all payments details, either failed or success with order id, amount, date, status, customer contact, etc.

**7.2 File Access and Permissions**

* JSON files ensure structured data storage, making it easier to load and update.
* Access to certain files is role-restricted to ensure data security.

**8. Testing and Quality Assurance**

**8.1 Testing Strategy**

* **Unit Testing:**  
  Each module (e.g., reservation.py, order.py) is tested separately.
* **Integration Testing:**   
  Modules are tested together in different user scenarios to confirm that they work seamlessly across roles.

**8.2 Test Cases**

1. **Login Test:** Check login access for all roles (staff, admin, super admin).
2. **Reservation Test:** Test table booking, viewing, and cancellation.
3. **Order Test:** Place, update, and cancel orders, and verify data in orders.json.

**9. Deployment**

**9.1 Deployment Plan**

* **Local Deployment:**   
  The system runs as a local application on any computer with Python installed, using JSON files for structured local storage.

**9.2 Maintenance Plan**

* **Data Backup:**  
  Regular backups of the /data/ folder should be taken to ensure data integrity.
* **Periodic Updates:**  
  Scripts can be modified to update functionalities as new features are added.

**10. Future Enhancements**

* **Database Integration:**  
  Consider migrating to a database for better scalability and performance.
* **User Interface:**  
  Develop a graphical interface for easier user interaction.
* **Advanced Reporting:**   
  Add features for generating more complex reports, such as daily sales and inventory tracking.

**11. Exception Handling**



The log function is designed to log error details into a file whenever an exception occurs, capturing specific information about the error, such as the time, the user's email, and the location within the code where the error happened.

**Code Breakdown**

1. **Function Definition**:

def log(tb, msg=None):

* + The function log takes two parameters:
    - tb: Expected to be traceback information from the exception, which contains details about where the error occurred.
    - msg: An optional parameter allowing a custom message to be logged alongside the error details.

1. **Get Current User**:

user = UserState().get\_state()

* + UserState().get\_state() retrieves the current user information, likely in the form of a dictionary. This function is expected to return None if no user is logged in.

1. **Construct the Error Dictionary**:

error = {

"date\_time": f"{str(datetime.now())}",

"email": f"{user['email'] if user else 'unknown'}",

"path": f"{tb[0]}",

"line": f"{tb[1]}",

"fn\_name": f"{tb[2]}",

"err": f"{tb[3]}",

"msg": f"{msg}"

}

* + **date\_time**: Records the current date and time using datetime.now().
  + **email**: If a user is logged in (user is not None), retrieves their email. Otherwise, logs "unknown".
  + **path**: The file path where the error occurred, extracted from tb[0].
  + **line**: The line number where the error occurred, extracted from tb[1].
  + **fn\_name**: The function name in which the error occurred, extracted from tb[2].
  + **err**: The actual error message or type, extracted from tb[3].
  + **msg**: Any custom message provided when calling log.

1. **Format Error for Logging**:

result = f"|{str(error)}\n"

* + Converts the error dictionary into a string and appends it with a separator | and newline for easy readability in the log file.

1. **Write Error to Log File**:

with open(LOG\_FILE, "a") as file:

file.write(result)

* + Opens the log file in append mode ('a') to add new log entries at the end without overwriting previous entries.
  + Writes the formatted error string result to the file.

1. **Handle Logging Errors**:

except Exception as error:

print(error)

* + If an error occurs within the log function itself, this except block catches it and simply prints it to the console. This prevents the function from failing silently and helps diagnose issues within the logging process.

**Usage Example**

When an exception is caught, you would typically call log as follows:

try:

# some code that might raise an exception

except Exception as e:

log(traceback.extract\_tb(e.\_\_traceback\_\_)[0], str(e))

In this example:

* traceback.extract\_tb(e.\_\_traceback\_\_)[0] extracts the traceback details for the error.
* str(e) provides a description of the error.

This function helps maintain a record of errors, which is useful for debugging and monitoring application stability.

****

main.py

### Code Summary

This Python script is designed to run an authentication function. Here’s a breakdown of the main components:

1. **Import**: It imports the authentication function from the src.authentication module.
2. **Main Function**: The main() function simply calls the authentication() function, which likely performs user authentication.
3. **Execution**: The if \_\_name\_\_ == "\_\_main\_\_": block ensures that the script runs the main() function only if executed directly, not when imported as a module.

### Purpose

This script likely serves as an entry point for an authentication process, where authentication() is responsible for verifying user credentials or establishing access.

****

authentication.py

**Code Summary**

The authentication function handles a user authentication menu, allowing users to choose between logging in, signing up, or exiting. It displays options in a formatted manner, validates user input, and logs errors if invalid inputs are provided.

**Components**

1. **Imports**:
   * login and signup functions from auth\_ui.auth are used to handle user login and registration.
   * Utility modules provide functions for validation, error handling, input retrieval, logging, and colored text.
2. **Error Handling and Input Validation**:
   * ErrorMessage: Used to store and display various error messages.
   * get\_input: Retrieves user input and applies the validate\_int function to ensure it's a valid integer.
   * LogError: Logs any input errors when invalid options are selected.
3. **Main Logic**:
   * A while True loop displays an authentication menu with options for **LOGIN**, **SIGNUP**, or **EXIT**.
   * The choice variable captures user input. If validation fails, it logs an error and exits the loop.
   * Depending on the user’s choice:
     + 1: Calls login() to authenticate an existing user.
     + 2: Calls signup() to register a new user.
     + 3: Exits the authentication loop.
4. **Color Formatting**:
   * bcolors is used to apply color coding to the menu text (e.g., headers, success messages, failure messages).

**Purpose**

This code provides a simple command-line user interface for authentication, guiding users through logging in, signing up, or exiting. It includes input validation, error handling, and visual formatting to enhance the user experience.

auth.py



**Code Summary**

The login function provides a process for user authentication by collecting and validating an email and password. It handles errors with custom error messages and logs any exceptions encountered during the login process.

**Components**

1. **Error Handling Setup**:
   * Initializes an instance of ErrorMessage, which holds various error messages for validation feedback.
2. **Email Input and Validation**:
   * Prompts the user to enter an email, using get\_input with validate\_email to check if the input is a valid email.
   * If email validation fails, it raises an exception with a custom error message (err\_msg.invalid\_email).
3. **Password Input and Validation**:
   * Prompts the user to enter a password, using get\_password with validate\_password to check password validity.
   * If password validation fails, it raises an exception with err\_msg.invalid\_password.
4. **User Login Attempt**:
   * Calls user\_login(email, password) to attempt to authenticate the user with the provided credentials.
5. **Exception Handling and Logging**:
   * Catches any exceptions raised during email or password entry, prints a failure message in red (bcolors.FAIL), and logs the error using log with traceback information to aid in debugging.

**Purpose**

This login function is a user authentication handler that collects, validates, and verifies login credentials, providing custom feedback for invalid inputs and logging errors to help with issue tracking.



Dashboard/ui/auth\_ui/auth.py

**Code Summary**

The signup function is a user registration process that collects, validates, and stores user information. It captures personal and contact information, performs input validation, and logs any errors that occur.

**Components**

1. **Error Handling Setup**:
   * Creates an instance of ErrorMessage, which contains custom error messages for each field's validation.
2. **User Information Collection and Validation**:
   * **Name**: Collected via get\_input with validate\_name. If validation fails, raises an exception with err\_msg.invalid\_name.
   * **Gender**: Collected with validate\_gender and error handling for invalid entries.
   * **Email**: Collected with validate\_email and raises an exception on invalid email format.
   * **Password**: Collected via get\_password with validate\_password. Raises an exception if invalid.
   * **Date of Birth**: Collected with validate\_dob and raises an exception for invalid date formats.
   * **Mobile Number**: Collected with validate\_mobile, handling errors for incorrect formats.
   * **Address**: Collected with validate\_address, with error handling for invalid addresses.
3. **User Registration Attempt**:
   * Once all inputs are validated, user\_signup is called with the collected data to complete the registration.
4. **Exception Handling and Logging**:
   * Catches any validation or input-related exceptions, displays a failure message in red (bcolors.FAIL), and logs the error using log with traceback details for debugging.

**Purpose**

The signup function facilitates a step-by-step user registration process by collecting essential user details, validating them, and logging any errors encountered. It provides feedback for invalid inputs and completes registration through user\_signup if all fields are valid.



controllers/auth\_controller/auth.py

**Code Summary**

The user\_signup function manages the creation of a new user account, checking if the email is already registered, assigning default settings, and storing the user data.

**Components**

1. **Error Handling Setup**:
   * Initializes ErrorMessage to access custom error and success messages.
2. **User Existence Check**:
   * Creates a User instance (USER) and calls check\_user(email) to see if the user’s email is already registered.
   * If the email is found, it displays an error message (err\_msg.email\_already\_resister), indicating that the user is already registered.
3. **Default Settings**:
   * If the email is not registered, it assigns a unique user ID using uuid.uuid4(), truncates it to 10 characters, and sets up default values:
     + **role**, **employment\_type**, **shift\_preferences**, **benefits**, **user\_status**, and **salary** — all retrieved from the Default class.
   * **date\_of\_joining** is set to the current date in dd-mm-yyyy format.
4. **New User Creation**:
   * Initializes a UserModel instance with the user's details and default settings.
   * Converts the user object to a dictionary (\_\_dict\_\_) and appends it to the users list of the USER object.
   * Saves the user data by calling USER.save\_user().
5. **Feedback and Logging**:
   * If the signup is successful, displays a success message (err\_msg.signup\_success) in green (bcolors.OKGREEN).
   * If an error occurs, logs the error with traceback details for debugging and displays an error message in red.

**Purpose**

The user\_signup function handles user registration by verifying email uniqueness, setting up default user settings, storing user data, and providing feedback. It ensures that each user has a unique ID and consistent settings while logging any errors encountered during the process.



database/users.json

**User Json Summary**

This JSON object represents a detailed user profile for a newly registered user in the system. Here is an overview of each attribute:

* **ID**: "f8d12d67-d" - A unique identifier for the user.
* **Name**: "admin" - The user's name.
* **Date of Birth**: "01-01-1996" - The user's birthdate, formatted as dd-mm-yyyy.
* **Email**: "admin@admin.com" - The user's email address.
* **Mobile Number**: "9876543210" - Contact number for the user.
* **Address**: "Darbhanga Bihar" - The user’s residential address.
* **Role**: "admin" - The user’s role within the system, indicating administrative privileges.
* **Salary**: 0000 - Placeholder for salary information (could be customized based on user role).
* **Employment Type**: "full time" - Specifies that the user works full-time.
* **Date of Joining**: "31-10-2024" - Date when the user joined, formatted as dd-mm-yyyy.
* **Shift Preferences**: "day" - The user's preferred working shift.
* **Status**: "active" - Indicates that the user's account is currently active.
* **Benefits**: ["health insurance", "meals"] - Lists benefits provided to the user, such as health insurance and meals.
* **Password**: "demo@123" - The user's password (note: should be stored securely in a real application).
* **Gender**: "male" - The user’s gender.

**Purpose**

This structure provides a comprehensive profile for each user, including personal, contact, employment, and role-based information. It can be used for user management, access control, and providing customized benefits or shifts based on user roles and preferences.

controllers/auth\_controller/auth.py



### Code Summary

The user\_login function handles the login process for a user, verifying the credentials (email and password) and determining the user’s access level. It checks for both normal users and super admins, providing appropriate responses based on the input.

#### Components

1. **Error Handling Setup**:
   * Initializes an instance of ErrorMessage to provide custom error messages during login.
2. **Super Admin Login**:
   * Checks if the provided email is superemail and password is Superpassword.
   * If these credentials match, it calls super\_admin(email, password) to handle the super admin login process.
3. **Regular User Login**:
   * If the credentials do not match the super admin login, the function proceeds to check if the user exists:
     + **Check User Existence**: Calls check\_user(email) to verify if the user exists in the system.
     + If the user is not found, it displays an error message (err\_msg.user\_not\_exist).
4. **Verify User Details**:
   * Loops through USER.users (a list of registered users) to find the matching email.
     + If the user’s status is "active", it checks if the provided password matches.
     + If the password matches, it updates the user's state using UserState().update\_state(user) and proceeds to the dashboard by calling dashboard().
     + If the password is incorrect, it shows a wrong credential error (err\_msg.wrong\_credential).
     + If the user's account status is not active, it displays an error message (err\_msg.account\_deactive).
5. **Exception Handling**:
   * If any exception occurs during the process, it logs the error with traceback information and prints an error message.

### Purpose

The user\_login function manages both regular user and super admin logins, validating credentials, checking user status, and ensuring secure access control. It provides appropriate error messages and handles both normal user and admin login processes with different flows.

Dashboard/mangage\_dashboard.py



### Code Summary

The dashboard function directs users to the appropriate dashboard based on their role. It checks the user’s role from the current state and calls the corresponding dashboard function. If the user’s role is unrecognized, it displays an error message.

#### Components

1. **Error Handling**:
   * An instance of ErrorMessage is created to provide custom error messages.
2. **User State Retrieval**:
   * The user\_state is retrieved using UserState().get\_state(), which returns the current logged-in user's state, including their role.
3. **Role-based Dashboard Navigation**:
   * The function checks the user's role (user\_state['role']):
     + **Super Admin**: If the role is 'super\_admin', it calls super\_admin\_dashboard() to display the super admin dashboard.
     + **Admin**: If the role is 'admin', it calls admin\_dashboard() to display the admin dashboard.
     + **Staff**: If the role is 'staff', it calls staff\_dashboard() to display the staff dashboard.
4. **Authorization Check**:
   * If the user's role is not recognized (i.e., it's not 'super\_admin', 'admin', or 'staff'), it prints an error message (err\_msg.not\_authorized) indicating that the user is not authorized to access the dashboard.

### Purpose

The dashboard function is designed to route users to different dashboards based on their role (super admin, admin, staff). If a user has an invalid or unrecognized role, it prevents access and shows an authorization error message.



Dashboard/super\_admin\_dashboard.py

### Code Summary

The super\_admin\_dashboard function provides a menu-driven interface for the super admin to perform various administrative tasks. It continuously displays options until the super admin chooses to log out or exit the dashboard.

#### Components

1. **Error Handling Setup**:
   * Creates an instance of ErrorMessage to display custom error messages for invalid options.
2. **Dashboard Menu**:
   * A while True loop ensures the super admin remains in the dashboard until they log out or choose to exit.
   * The menu includes the following options:
     + **1 MAKE ADMIN**: Promotes a user to an admin role.
     + **2 MAKE STAFF**: Assigns a user the role of staff.
     + **3 VIEW ALL USERS**: Displays all registered users.
     + **4 REMOVE USER**: Removes a user from the system.
     + **5 LOGOUT**: Logs out the super admin and exits the dashboard.
3. **User Input and Validation**:
   * The function uses get\_input to capture the super admin’s choice and validates it using validate\_int.
   * If the input is invalid, it prints an error message (err\_msg.invalid\_option) and prompts for input again.
4. **Task Execution**:
   * Based on the super admin’s choice, the corresponding function is called:
     + make\_admin(): Promotes a user to admin.
     + make\_staff(): Assigns a staff role to a user.
     + get\_all\_user(): Displays all users.
     + remove\_user(): Removes a user from the system.
   * After each task, it asks the super admin if they want to continue in the dashboard or exit using ask\_for\_dashboard(). If ask\_for\_dashboard() returns False, the loop breaks and exits the dashboard.
5. **Exiting the Dashboard**:
   * If the super admin selects option 5 (log out) or chooses to exit after a task, the loop breaks and the function ends, effectively logging the super admin out.

### Purpose

The super\_admin\_dashboard function offers an interface for the super admin to manage users and perform high-level administrative tasks. It provides a menu-driven experience where tasks can be executed, and the super admin can log out or choose to return to the menu after each operation.



Dashboard/admin\_dashboard.py

### Code Summary

The admin\_dashboard function provides a menu-driven interface for an admin user to access different parts of the Restaurant Management System (RMS). It offers various functionalities such as managing items, stock, orders, reports, and staff, as well as updating their profile.

#### Components

1. **Error Handling Setup**:
   * An instance of ErrorMessage is created to display custom error messages for invalid input options.
2. **Dashboard Menu**:
   * A while True loop keeps the admin in the dashboard menu until they choose to log out or exit.
   * The menu includes the following options:
     + **1 ITEM**: Access the item management page.
     + **2 STOCK**: Manage the stock.
     + **3 ORDER**: Manage customer orders.
     + **4 REPORTS**: View reports related to the business.
     + **5 STAFF**: Manage staff members.
     + **6 MY PROFILE**: View the admin's current profile.
     + **7 UPDATE PROFILE**: Update the admin's profile.
     + **8 ERROR**: View or handle error logs.
     + **9 LOGOUT**: Log out and exit the dashboard.
3. **User Input and Validation**:
   * The function uses get\_input to capture the admin's menu choice, validating it with validate\_int.
   * If the input is invalid, an error message (err\_msg.invalid\_option) is displayed, and the user is prompted to input a valid option again.
4. **Task Execution**:
   * Based on the user's choice, the appropriate function is called:
     + **Item Management**: item\_page() - Manages items in the system.
     + **Stock Management**: stock\_page() - Manages stock levels.
     + **Order Management**: order\_page() - Manages orders placed by customers.
     + **Reports**: report\_page() - Displays business reports.
     + **Staff Management**: staff\_page() - Manages staff members.
     + **View Profile**: get\_current\_user() - Displays the current admin's profile.
     + **Update Profile**: profile\_update\_page() - Allows the admin to update their profile information.
     + **Error Management**: error\_page() - View or handle error logs.
5. **Profile Access and Update**:
   * After viewing the current profile (get\_current\_user()), the admin is asked whether they want to return to the dashboard using ask\_for\_dashboard(). If the response is False, the loop breaks and exits the dashboard.
6. **Exiting the Dashboard**:
   * If the admin selects option 9 (log out) or decides to exit after performing a task, the loop ends, and the admin logs out of the system.

### Purpose

The admin\_dashboard function is designed to provide admins with a centralized control panel where they can manage various aspects of the Restaurant Management System (RMS). From managing stock and orders to handling reports and updating profiles, this dashboard is a comprehensive interface for the admin role.



Dashboard/staff\_dashboard.py

### Code Summary

The staff\_dashboard function provides a menu-driven interface for staff members to manage various tasks within the restaurant, including viewing the menu, handling orders, processing billing, reservations, and updating their profile. The dashboard allows staff to perform their duties with ease and log out when finished.

#### Components

1. **Error Handling Setup**:
   * An instance of ErrorMessage is created to display custom error messages for invalid input options.
2. **Dashboard Menu**:
   * A while True loop keeps the staff member in the dashboard menu until they log out or choose to exit.
   * The menu includes the following options:
     + **1 MENU**: Access and view the menu.
     + **2 ORDER**: Manage customer orders.
     + **3 BILLING**: Process payments and billing for orders.
     + **4 VIEW INVOICE**: View the invoice for a specific order.
     + **5 RESERVATION**: Manage customer reservations.
     + **6 MY PROFILE**: View the staff member's current profile.
     + **7 UPDATE PROFILE**: Update the staff member's profile.
     + **8 LOGOUT**: Log out and exit the dashboard.
3. **User Input and Validation**:
   * The function uses get\_input to capture the staff member's menu choice, validating it with validate\_int.
   * If the input is invalid, an error message (err\_msg.invalid\_option) is displayed, and the user is prompted to input a valid option again.
4. **Task Execution**:
   * Based on the staff member's choice, the appropriate function is called:
     + **Menu**: menu() - Displays the restaurant's menu.
     + **Order Management**: order\_page() - Manages customer orders.
     + **Billing**: pay\_bill() - Processes payments for orders.
     + **Invoice Viewing**: invoice() - Displays the invoice for a customer.
     + **Reservation Management**: reservation\_page() - Manages customer reservations.
     + **View Profile**: get\_current\_user() - Displays the current staff member's profile.
     + **Update Profile**: profile\_update\_page() - Allows the staff member to update their profile information.
5. **Profile Access and Update**:
   * After viewing the current profile (get\_current\_user()), the staff member is asked whether they want to return to the dashboard using ask\_for\_dashboard(). If the response is False, the loop breaks and exits the dashboard.
6. **Exiting the Dashboard**:
   * If the staff member selects option 8 (log out) or chooses to exit after completing a task, the loop ends, logging them out of the system.

### Purpose

The staff\_dashboard function is designed to give restaurant staff members easy access to essential features, such as managing orders, billing, reservations, and their profile. It provides a streamlined interface for handling daily tasks and ensures that staff can log out once they are finished.



Dashboard/UI/pages/order\_page.py

### Code Summary

The order\_page function provides a role-specific order management interface for both staff and admin users. Depending on the user role (retrieved from UserState), different sets of options are displayed for creating, updating, viewing, or canceling orders.

**Components**

1. **User Role Validation**:
   * The user's role (staff or admin) is retrieved from UserState. Based on the role, specific order options are displayed.
2. **Order Options for Staff**:
   * For users with the staff role, a while True loop displays the following options:
     + **1 CREATE ORDER**: Calls order\_system() to create a new order.
     + **2 UPDATE ORDER**: Calls update\_order() to modify an existing order.
     + **3 VIEW ORDER IN PROCESS**: Calls get\_unpaid\_order() to view unpaid or pending orders.
     + **4 VIEW TODAY ORDER**: Calls today\_order() to display all orders from the current day.
     + **5 BACK**: Exits the order page and returns to the previous menu.
   * After each task, ask\_for\_dashboard("Back") is used to ask the staff member if they want to return to the order page or exit.
3. **Order Options for Admin**:
   * For users with the admin role, a different set of order management options is displayed:
     + **1 CANCEL ORDER**: Calls cancel\_order() to cancel an existing order.
     + **2 VIEW ORDER DETAILS**: Calls get\_order\_details() to view detailed information for a specific order.
     + **3 VIEW ALL ORDER**: Calls get\_all\_order() to see a list of all orders.
     + **4 BACK**: Exits the order page and returns to the previous menu.
   * Similar to the staff menu, ask\_for\_dashboard("Back") prompts the admin to either return to the order page or exit.
4. **User Input and Validation**:
   * get\_input captures the user's menu choice and validates it with validate\_int.
   * If an invalid choice is entered, an error message (err\_msg.invalid\_option) is displayed, and the user is prompted again.
5. **Exiting the Order Page**:
   * If the user selects the "Back" option (5 for staff, 4 for admin) or chooses to exit after completing a task, the loop breaks, returning the user to the previous menu.

**Purpose**

The order\_page function is a role-based interface for managing orders within the system. It allows:

* **Staff** to handle order creation, updates, and view pending or daily orders.
* **Admins** to cancel orders, view order details, and access a list of all orders.

This function effectively separates order functionalities by role, ensuring each user type has access only to the relevant options for their duties.

database/orders.json



**Summary of Order Data**

This JSON object represents an **order record** within a restaurant management system. It includes details about the customer, purchased items, pricing, and order status. Below is a breakdown of each section:

1. **Order Details**:
   * **Order ID**: "id": "9437" — A unique identifier for the order.
   * **Customer Name**: "name": "mansoor" — The name of the customer placing the order.
   * **Mobile Number**: "mobile\_no": "7979941442" — Customer’s contact information.
   * **Order Date**: "date": "09-11-2024" — The date the order was placed.
2. **Items Ordered**:
   * **Items Array**: A list of items included in the order. Each item contains:
     + **ID**: Unique identifier for each item (e.g., "id": "F8F3" for water).
     + **Name**: Name of the item (e.g., "name": "water").
     + **Category**: Type or category of the item, such as "DRINK".
     + **Sale Price**: Unit price of the item (e.g., "sale\_price": 20.0 for water).
     + **Quantity**: The number of units ordered (e.g., "quantity": 2).
     + **Added By**: Email of the user who added the item to the order.
3. **Order Financials**:
   * **Total**: "total": 280.0 — The subtotal before taxes or discounts.
   * **Tax**: "tax": 50.4 — Total tax amount applied to the order.
   * **Tax Percentage**: "tax\_percent": 18 — The tax rate applied to the subtotal.
   * **Discount**: "discount": 0 — Any discount applied to the order (none in this case).
   * **Grand Total**: "grand\_total": 330 — The final payable amount after taxes and discounts.
4. **Order Metadata**:
   * **Created By**: "create\_by": "staff@staff.com" — Email of the staff member who created the order.
   * **Status**: "status": "paid" — The payment status of the order.

**Purpose**

This JSON structure provides a detailed snapshot of a customer's order, including items purchased, their categories, and financial breakdowns (subtotal, tax, discount, and grand total). The metadata further clarifies which staff member handled the order and its current status, aiding in record-keeping and analytics within the restaurant management system.



controllers/reservaiton\_controller/reservation.py

**Code Summary**

The reserved\_table function handles the reservation process for tables at a restaurant. Given a customer's name, contact number, desired time slot, and the number of people, it attempts to find an available table that meets the requirements and reserves it if possible.

**Components**

1. **Function Arguments**:
   * name: The customer's name.
   * mobile\_no: The customer's contact number.
   * time\_slot: The requested reservation time.
   * persons: The number of people for the reservation.
2. **Process Flow**:
   * **Date Setup**: Sets date to the current date in "dd-mm-yyyy" format.
   * **Table Search**: Calls find\_table(time\_slot, persons) to locate an available table that fits the time slot and size requirement.
     + If no table is available, an error message is displayed (table\_not\_available), and the process ends.
     + If an available table is found, it proceeds with the reservation.
   * **Reservation Creation**:
     + A unique ID is generated for the reservation.
     + status is set to "reserved".
     + A new ReservationModel object is created, storing reservation details including the table ID, date, time slot, and customer information.
     + The new reservation is added to RESERVATION.reservations, and RESERVATION.save\_reservation() is called to save it persistently.
   * **Confirmation Message**: Displays a success message with the table ID and reservation ID for confirmation.
3. **Error Handling**:
   * Any exceptions that occur are caught, printed as an error message, and logged with log(traceback.extract\_tb(error.\_\_traceback\_\_)[0], error).

**Purpose**

This function provides an interface for making table reservations. It checks for table availability, assigns an ID to each reservation, and stores it in the system. Additionally, the function informs the customer about the success or failure of their reservation request, and logs errors as needed for further investigation.

**Project Summary**

This Python-based restaurant management system is designed to manage various operational tasks within a restaurant setting. It uses the local file system for storing data in JSON-like structures, bypassing the need for an external database. The project includes these main components:

**Key Features and Modules**

1. **Authentication System**:
   * Provides a login system for multiple roles, including super admin, admin, and staff, each with unique permissions.
   * Handles new user signups, login, and password validation.
2. **Role-Specific Dashboards**:
   * **Super Admin**: Can manage user roles (create or remove admins and staff), view all users, and perform high-level administrative functions.
   * **Admin**: Accesses features like inventory (stock) management, order tracking, and report generation. They can view, create, and update orders and manage staff.
   * **Staff**: Has access to essential tools for day-to-day tasks, such as creating reservations, managing the menu, processing orders, handling billing, and updating profiles.
3. **Order Management**:
   * Allows creation, updating, and tracking of customer orders.
   * Supports the viewing of current and historical orders, with options for different staff roles to view relevant order information.
4. **Table Reservation**:
   * Staff members can reserve tables for customers, specifying the name, time slot, and number of people.
   * Reservation details are stored in the local file system.
5. **Profile Management**:
   * Users can update personal details like name, address, mobile number, and password.
6. **Error Logging**:
   * Captures error messages and application issues in a structured format, recording details like the function name, error message, and user email if available.
7. **Local File-Based Data Storage**:
   * All user, order, reservation, and error log data is stored in files, simulating database-like operations with file handling in Python.
   * Data for users and reservations is saved as JSON-like structures, making it easy to store and retrieve structured data.

The system supports real-time operations for a restaurant while maintaining simple file-based storage, making it lightweight and ideal for small-scale applications without database dependency.