**Book Bazaar Project Report**

1. **Introduction & Objective**

* **Project Overview**

Book Bazaar is a comprehensive library management system designed to streamline the management of books, authors, inventory, and sales. It provides functionalities to add, update, and retrieve information about books, authors, and inventory, along with advanced features like inventory tracking and quantity management. The system ensures data integrity, scalability, and user-friendliness, making it a robust solution for library or bookstore operations.

1. **Design Goals**
2. **Scalability:** The system is designed to handle a growing number of books and users.
3. **Flexibility:** By using both relational and non-relational databases, the system supports diverse data storage requirements.
4. **Performance:** Optimized queries and database interactions ensure fast response times.

**Part One: SQLite**

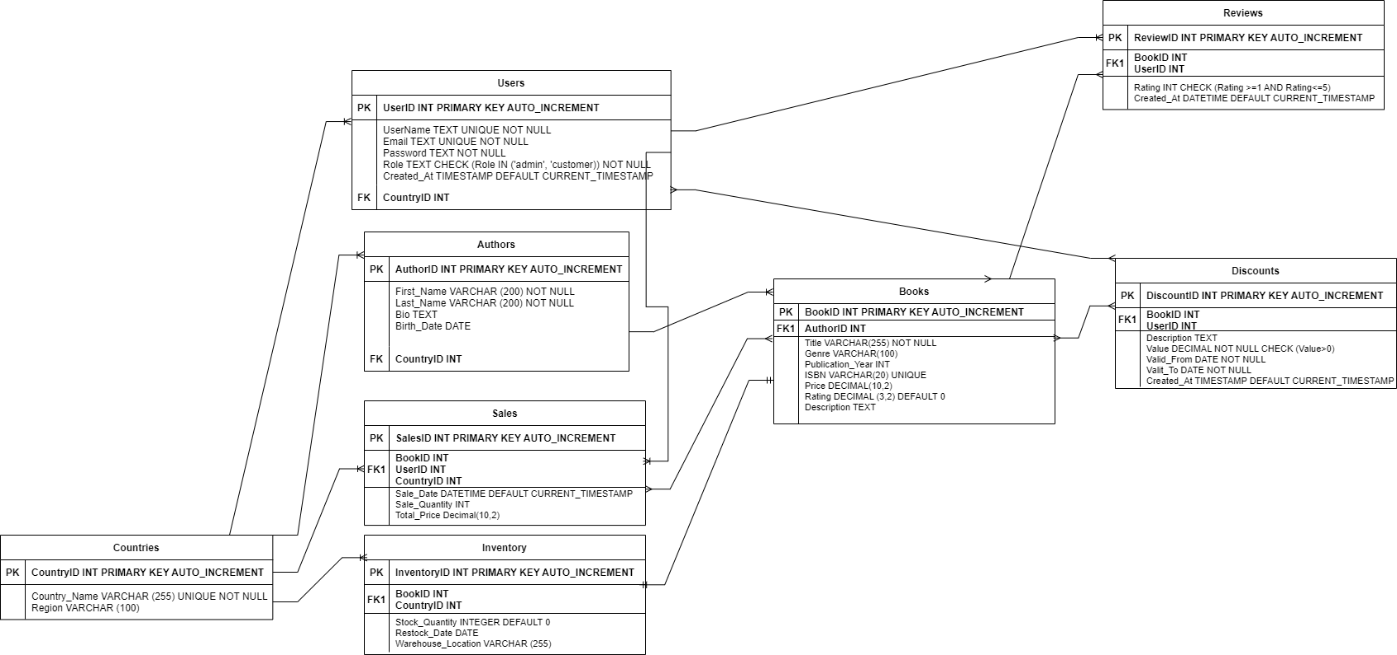
1. **Key Features**
   * 1. Book Management

* Add, update, delete and retrieve book information
* List all books in a well-formatted output.
  + 1. Authors Management
* Add new authors
* Retrieve details of all authors.
  + 1. Inventory Management
* Track stock quantities by warehouse.
* Update stock for specific books.
* Retrieve inventory summaries for warehouses.
  + 1. Advanced Queries
* Get inventory summaries by location.
* Retrieve books and inventory data based on custom queries.
  + 1. Error Handling
* Comprehensive error messages for database constraints and connection issues.
* Validations for inputs like BookID, AuthorID, and InventoryID.

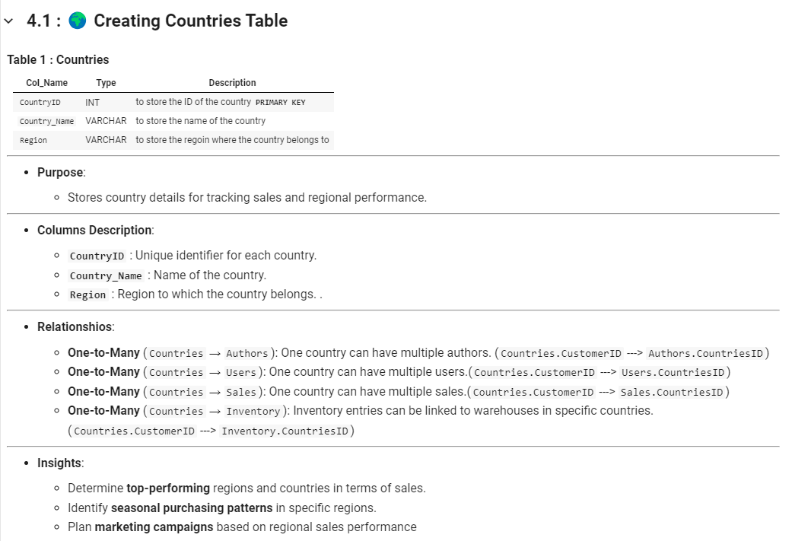
1. **Database Setup & Schema Design**

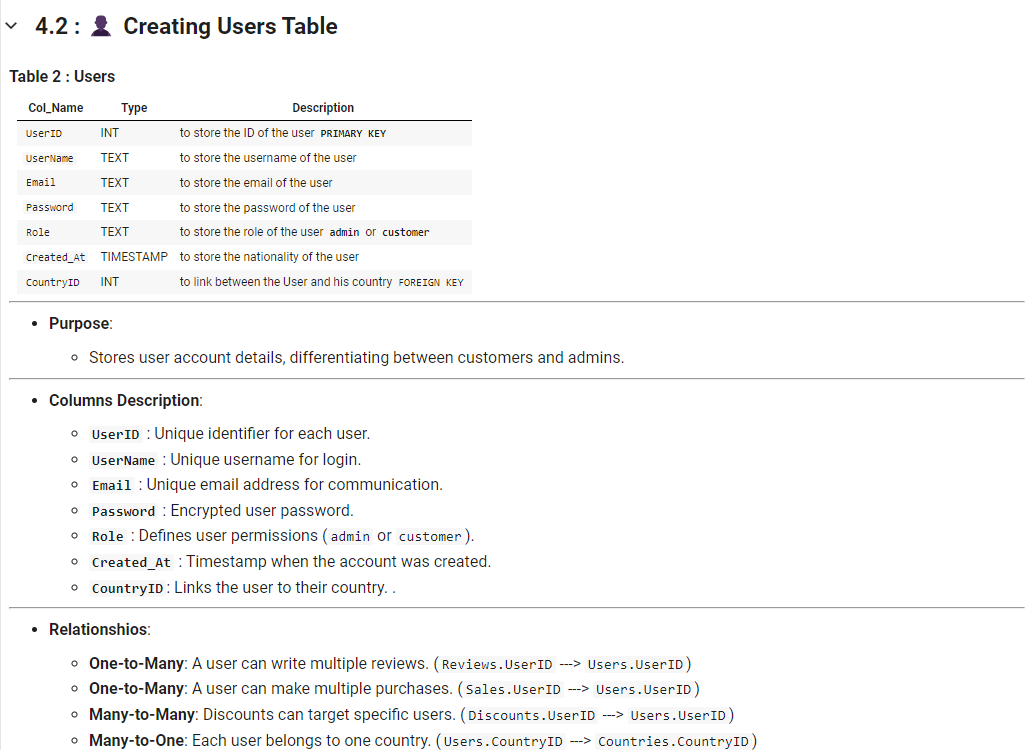
**2.1 Task 1: Set Up the Relational Database with SQLite**

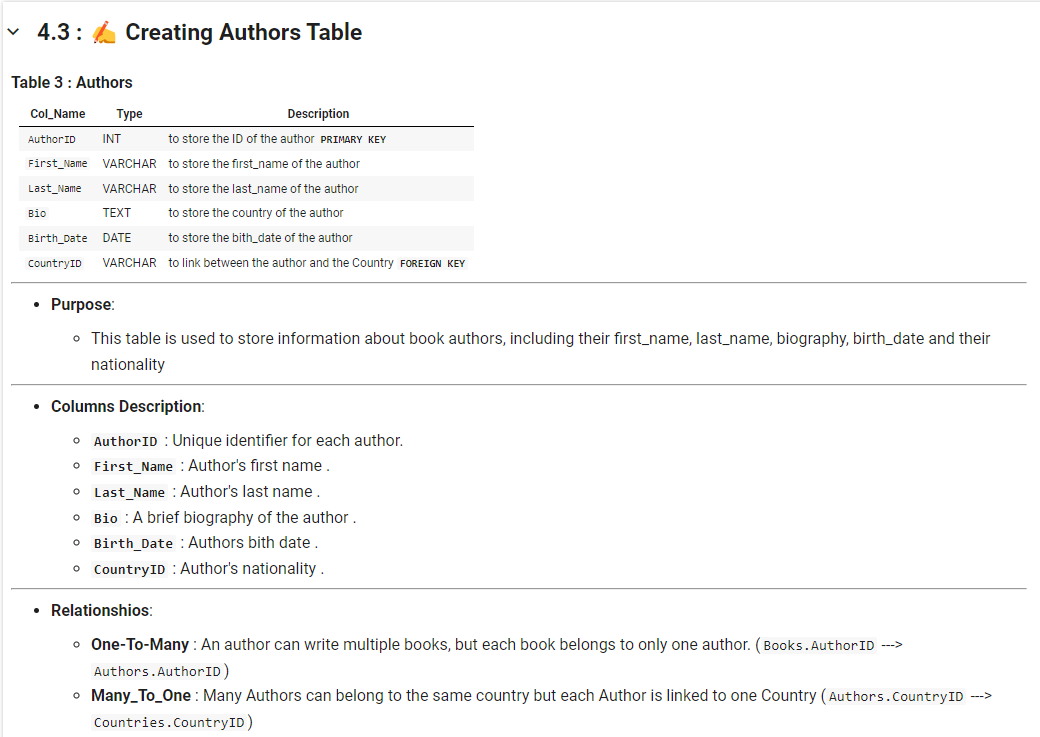
* SQLite Setup:
  + **SQLite is used as the file-based database for storing structured data about books, authors, users, and inventory.**
  + **A new SQLite database file (bookbazaar.db) is created for this purpose.**
  + **The application ensures that necessary read/write permissions are granted to the database file.**
  1. **Task 2: Design the Relational Database Schema**
* Data Base Schema

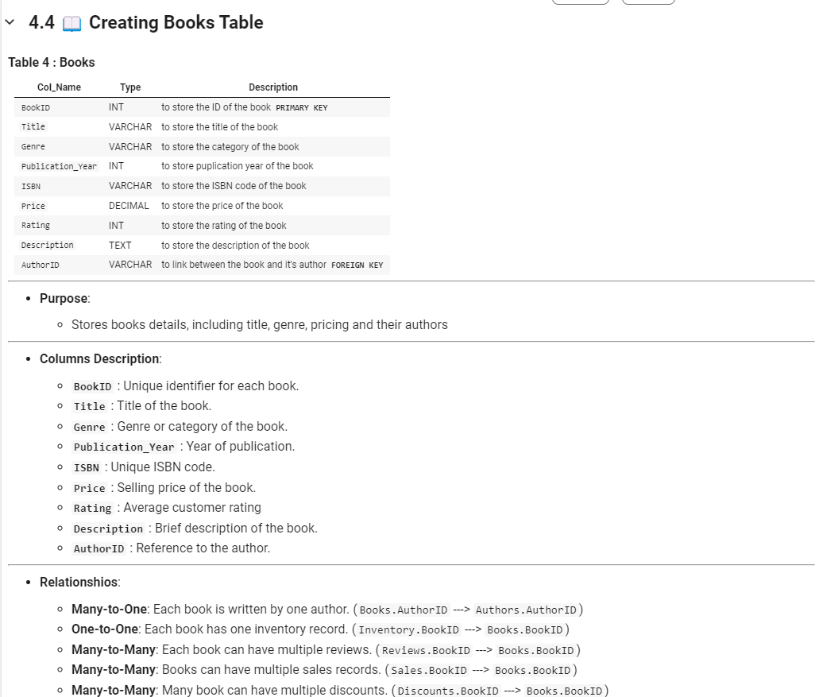


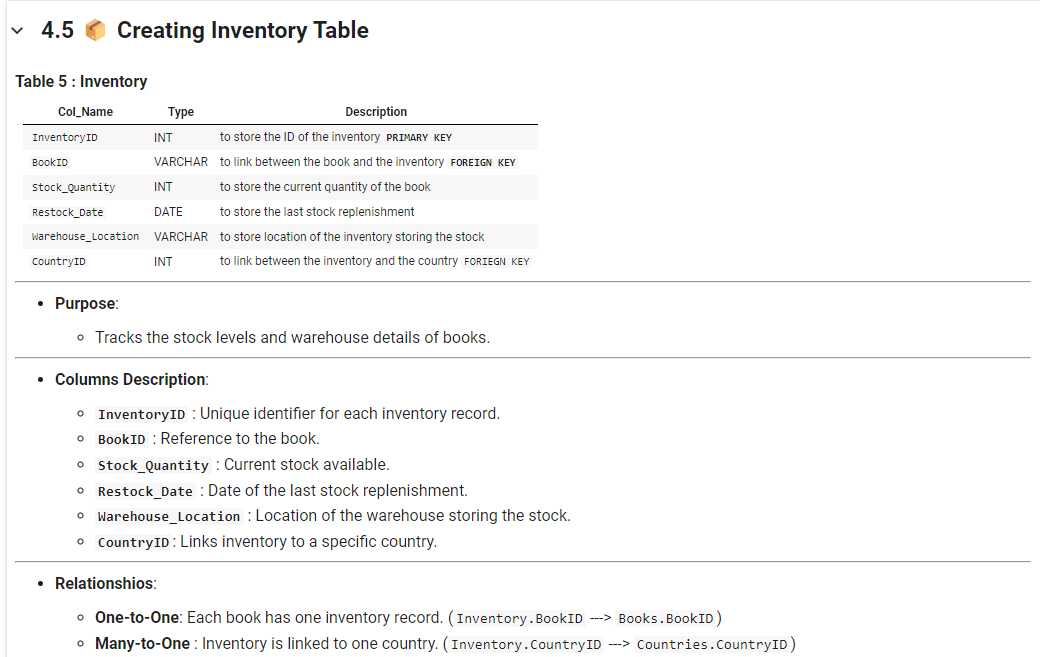
* Entities and Tables

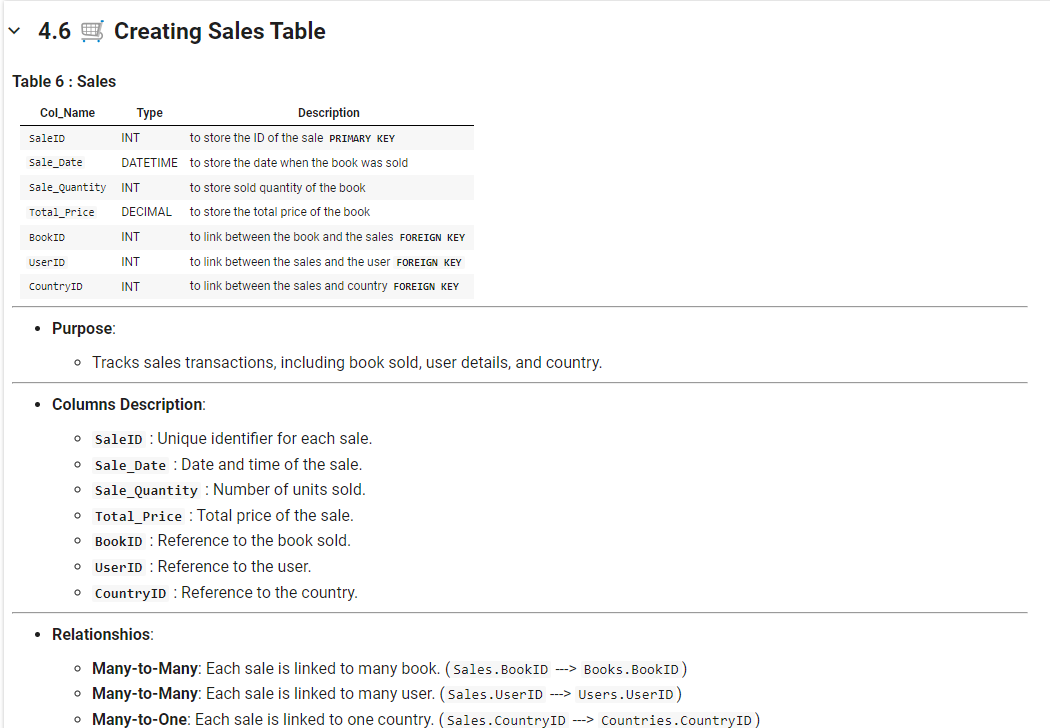
****

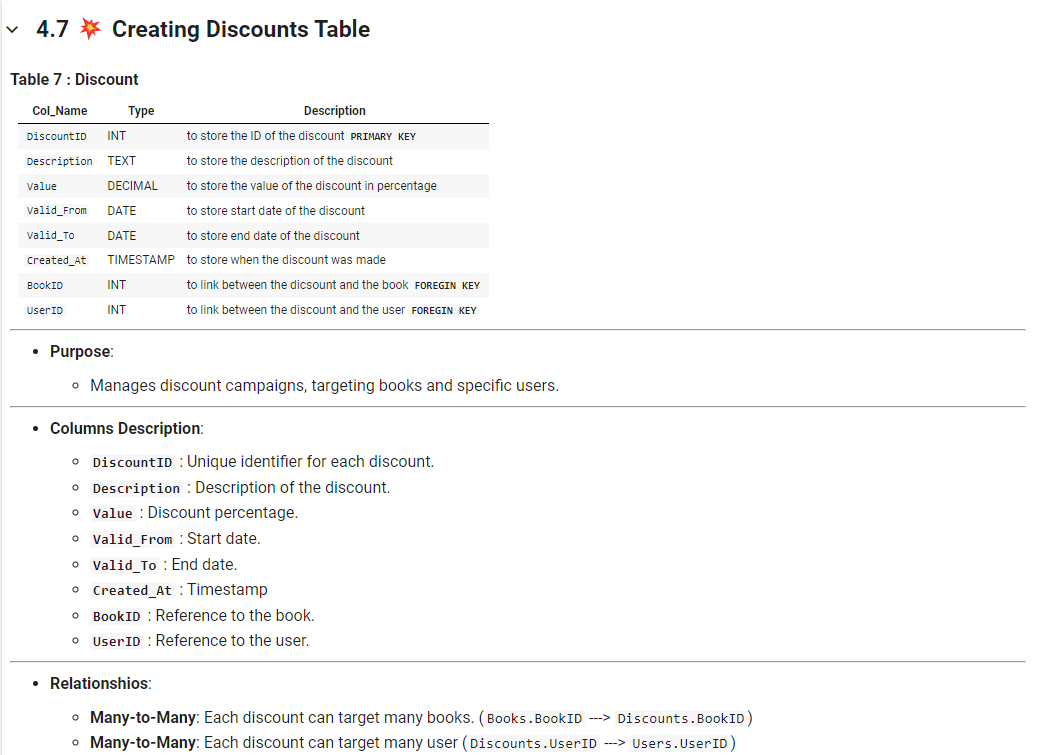
****

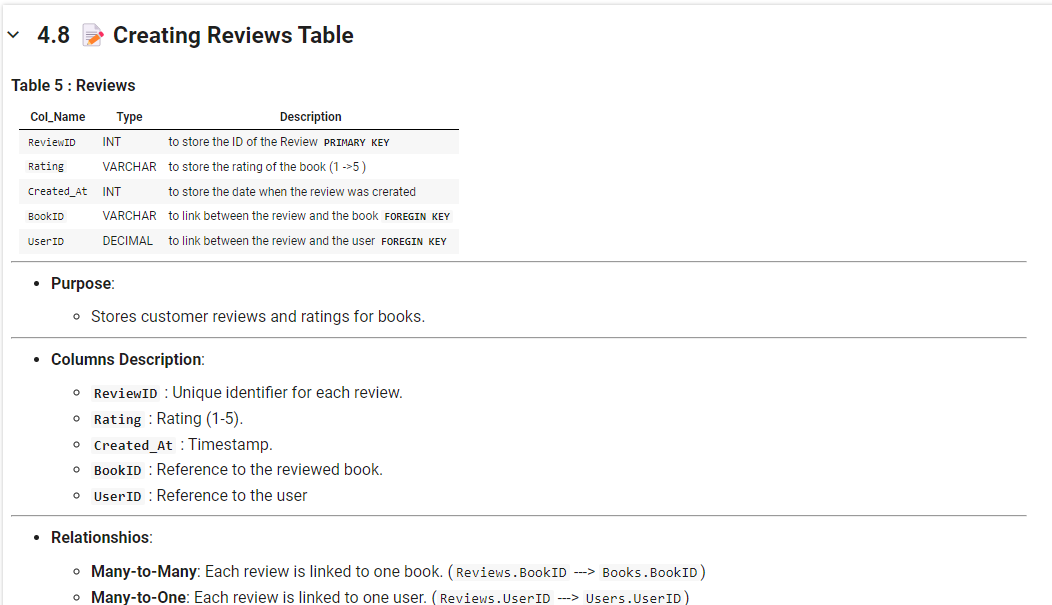
****

****

****

****

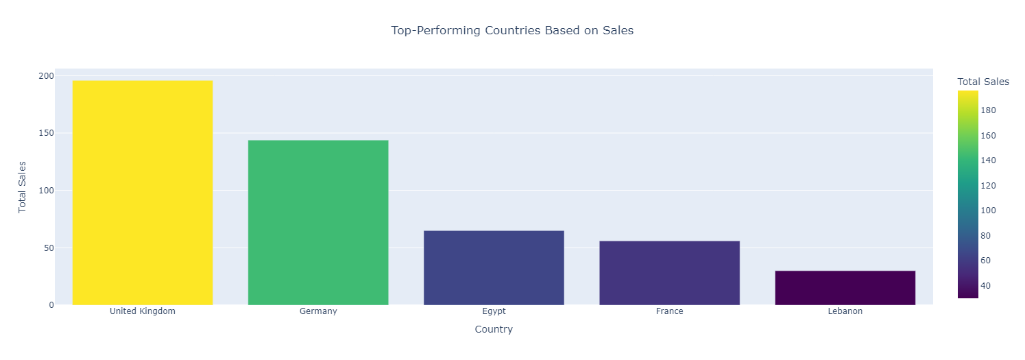
****

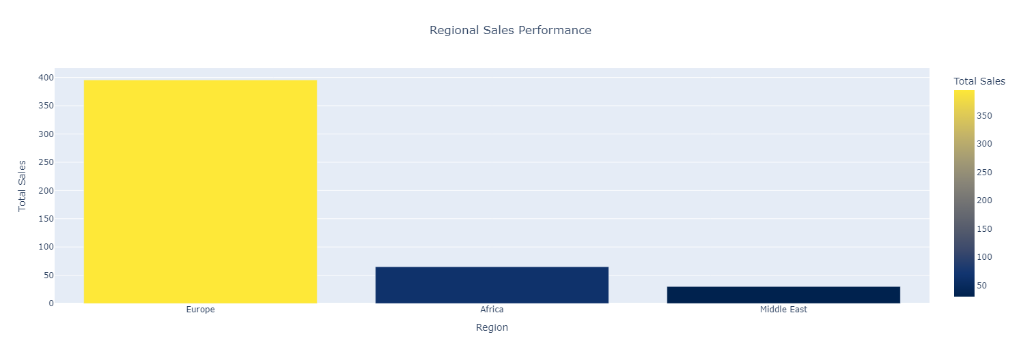
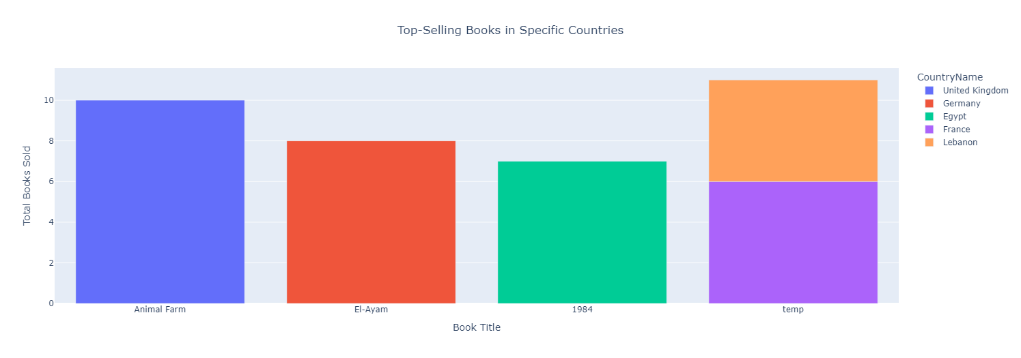
****

* Triggers:

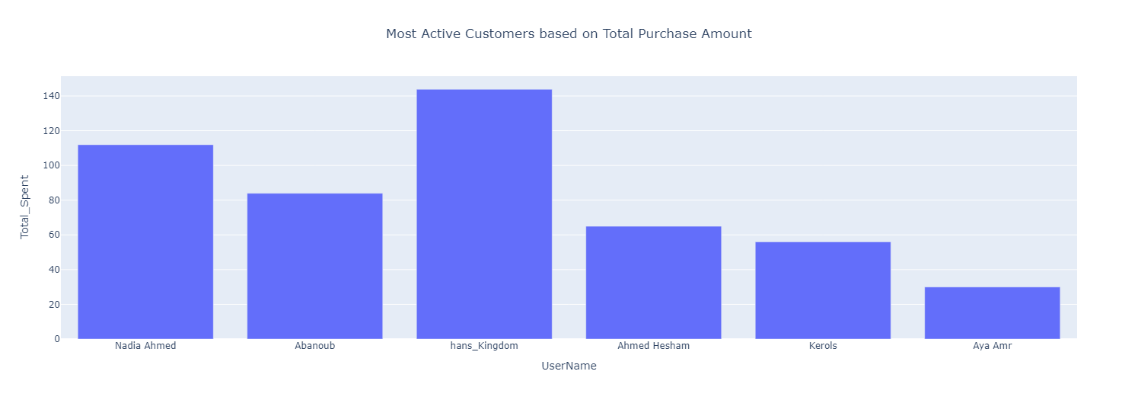
A trigger is implemented to automatically update the stock quantity in the Books table when books are purchased from sales. This ensures accurate and real-time inventory management.

1. **Insights and Analysis**

****

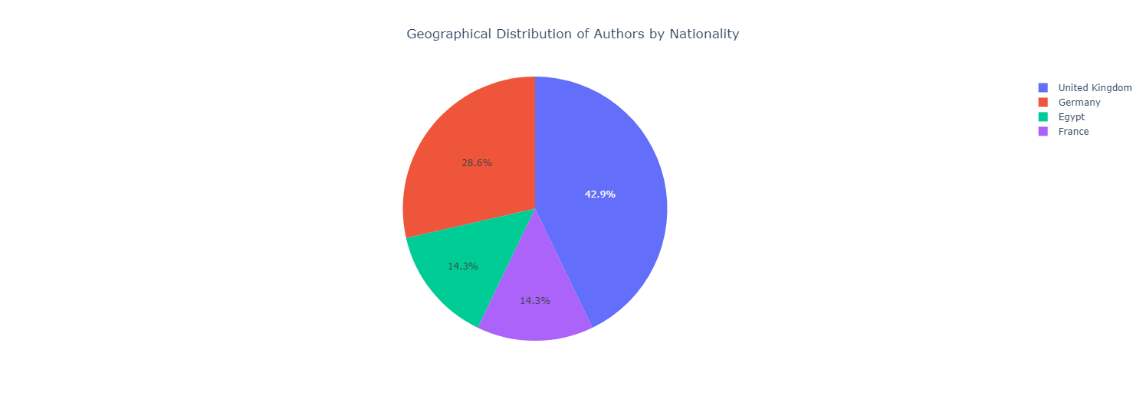
****

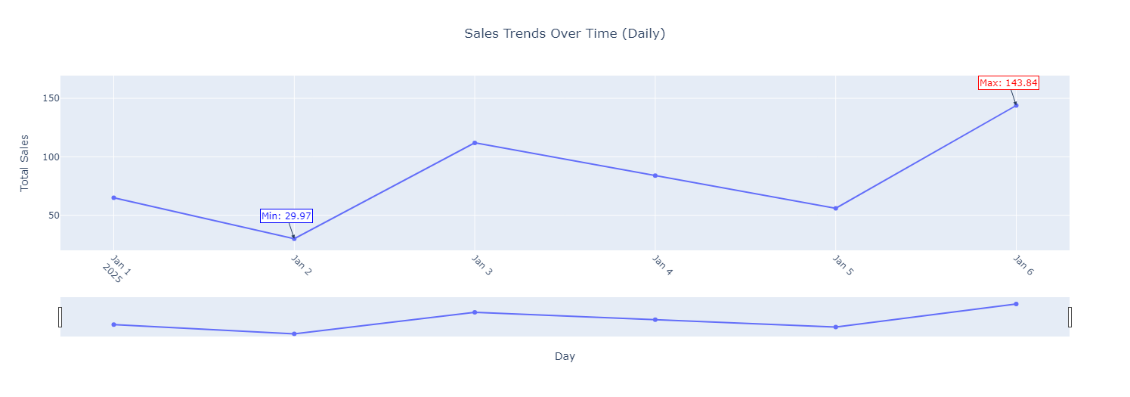
****

****

****

****

****

****

****

1. **Error Handling & Validation**
   * **Comprehensive error handling is implemented to manage database constraints, such as unique book ISBNs, email addresses for users, and valid foreign key relationships.**
   * **Input validations are performed to ensure that necessary fields like BookID, AuthorID, and InventoryID are valid before executing queries.**
2. **CRUD Operations**

CRUD operations are central to the **Book Bazaar** project, providing the backbone for database interactions.

**User Management**

* **Create**: Add new users with details such as name, email, and membership type.
* **Read**: Fetch user details, including borrowing history.
* **Update**: Modify user information, e.g., update contact details.
* **Delete**: Remove users who are no longer active members.

**Book Management**

* **Create**: Add new books to the library catalog.
* **Read**: Search for books by title, author, or genre.
* **Update**: Edit book information such as availability status.
* **Delete**: Remove books that are outdated or damaged.

**Transaction Management**

* **Create**: Record a new book borrowing transaction.
* **Read**: Retrieve transaction histories.
* **Update**: Modify transaction details, e.g., extend borrowing duration.
* **Delete**: Delete erroneous transactions.

1. **Conclusion**

* The **Book Bazaar** project is a robust and scalable library management system, capable of addressing the needs of modern library operations. With its comprehensive features and potential for future enhancements, it aims to provide a seamless and efficient user experience.

**Part Two: MongoDB Integration**

**Overview**

MongoDB is a NoSQL, document-oriented database that is ideal for storing large amounts of data with flexible schemas. In the BookBazaar Library Management System, MongoDB is used to store book reviews, allowing for efficient retrieval and modification of review data. Reviews for each book are stored as documents in a MongoDB collection, enabling dynamic updates and queries. MongoDB’s flexible data model and high performance make it an excellent choice for handling the review data in this system.

The goal of this task is to integrate MongoDB into the BookBazaar system to manage reviews for books, utilizing the PyMongo library for Python to interact with the database. By the end of this task, we will have created functions for performing CRUD (Create, Read, Update, Delete) operations and set up the necessary API routes to interact with these functions.

**Implementation Steps**

1. **Setting Up MongoDB**:
   * First, we installed MongoDB on the local server and ensured it was running.
   * We created a database named bookbazaar\_reviews specifically for storing reviews.
   * The Reviews collection was created within this database to store individual review documents for books.
2. **Installing PyMongo**:
   * PyMongo, the Python driver for MongoDB, was installed using pip install pymongo.
   * This library allows Python applications to communicate with MongoDB by providing a simple interface for performing CRUD operations.
3. **Establishing Connection to MongoDB**:
   * We created the connect\_to\_mongodb() function that establishes a connection to the MongoDB server at localhost:27017, the default MongoDB host and port.
   * The connection ensures that the Python script can access the bookbazaar\_reviews database for performing operations.
4. **Creating CRUD Functions**:
   * **Create (Add a Review)**: A function add\_review() was created to insert new reviews into the Reviews collection. Each review is stored as a document containing book information, user details, rating, and the review comment.
   * **Read (Get Reviews)**: A function get\_reviews\_by\_book() was implemented to retrieve all reviews associated with a specific book using its unique book\_id. This function returns a list of reviews for that book.
   * **Update (Modify a Review)**: The update\_review\_by\_id() function was created to allow updates to existing reviews based on the review’s unique \_id. It supports partial updates to review fields, such as the rating or comment.
   * **Delete (Remove a Review)**: The delete\_review\_by\_id() function allows deletion of a review from the database by its \_id, enabling administrators or users to manage their reviews efficiently.
5. **Testing CRUD Operations**:
   * Each function was tested in isolation to ensure that data could be successfully inserted, queried, updated, and deleted from the database.
   * The add\_review() function was tested by adding several sample reviews to the collection.
   * The get\_reviews\_by\_book() function was tested by retrieving reviews for specific book IDs to ensure correct data retrieval.
   * The update\_review\_by\_id() function was tested by updating specific fields of existing reviews.
   * Finally, the delete\_review\_by\_id() function was tested by removing reviews from the collection.
6. **Integrating MongoDB with Flask API**:
   * We integrated the MongoDB CRUD functions into the Flask application.
   * The following API routes were created to manage reviews:
     + GET /books/<book\_id>/reviews: Retrieve all reviews for a book.
     + POST /books/<book\_id>/reviews: Add a new review to a book.
     + PUT /reviews/<review\_id>: Update a specific review.
     + DELETE /reviews/<review\_id>: Delete a review.
7. **Testing the API**:
   * The Flask API was tested using Postman to ensure that each route correctly interacted with the MongoDB database.
   * Test cases included adding reviews, retrieving reviews, updating a review, and deleting a review.

**Achievements**

* **MongoDB Setup**: Successfully set up MongoDB to store and manage reviews for books in the bookbazaar\_reviews database.
* **PyMongo Integration**: Integrated the PyMongo library to interact with MongoDB from Python, enabling seamless data operations for the application.
* **CRUD Functionality**: Implemented robust CRUD operations for managing reviews, including adding, retrieving, updating, and deleting reviews. These operations ensure that the system is fully capable of handling review data.
* **Flask API Integration**: Developed RESTful API endpoints in Flask to allow external clients to interact with the MongoDB review data, ensuring that reviews can be managed via HTTP requests.
* **Error Handling**: Implemented error handling for each operation to manage scenarios such as failed database connections, review not found, or invalid data formats.

**Summary**

Task 2 focused on integrating MongoDB into the BookBazaar Library Management System for the purpose of storing and managing book reviews. We established a connection to MongoDB, created the necessary CRUD operations, and integrated them into a Flask-based API to make the review management system fully operational. The use of MongoDB enables flexible and efficient management of book reviews, while PyMongo allows seamless interaction between Python and the MongoDB database. The system now supports adding, updating, deleting, and retrieving reviews via both backend functions and API routes.

By completing this task, we have established the foundation for handling review data efficiently in a scalable NoSQL database, enabling the BookBazaar system to provide a flexible and robust review management system.

**Part Three: Book Bazaar API Documentation**

**Overview:**

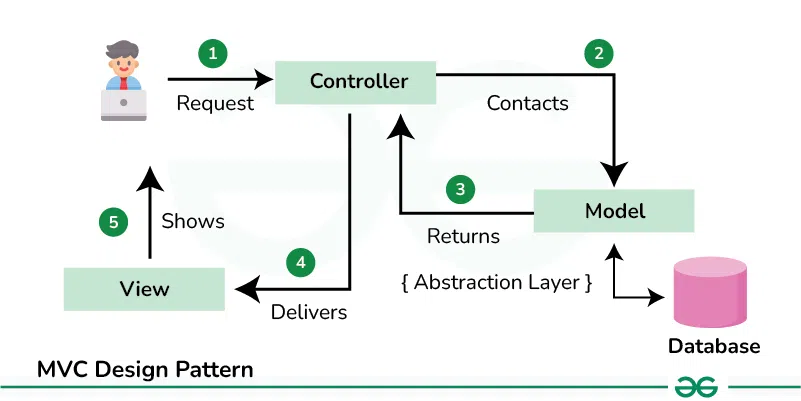
The Book Bazaar project is a comprehensive library management system designed using the Model-View-Controller (MVC) architectural pattern. The system integrates a relational database (SQLite) and a non-relational database (JSON) for efficient data management and provides RESTful APIs for various functionalities like user management, book sales, reviews, and inventory control.

**MVC Design Pattern:**

The Model-View-Controller (MVC) design pattern separates application logic into three interconnected components:

* 1. **Model:** Manages the data, logic, and rules of the application.
  2. **View:** Represents the UI (HTML/CSS files in this project).
  3. **Controller:** Handles user inputs, processes them, and updates the view and model accordingly.

**MVC Architecture Flow:**

* **User Interaction:** Users interact with the system through API endpoints.
* **Controller:** Processes requests, communicates with the model, and determines the response.
* **Model:** Interacts with the database to retrieve or manipulate data.
* **View:** Displays the processed data in an organized manner.

**Project Structure:**

**Root Folder Contents**

* **app.py:** The main application file initializing Flask and defining routes.

**Folders**

1-Config/  
Contains database configuration scripts:

* **db\_config.py:** Manages SQLite and JSON database connections.

2-Model/  
Implements the logic and data management for various entities:

* **user\_model.py:** Handles user data and operations.
* **sale\_model.py:** Manages book sales data.
* **review\_model.py:** Tracks reviews and ratings.
* **inventory\_model.py:** Manages book inventory.
* **helper\_functions.py:** Provides utility functions for the models.
* **discount\_model.py:** Implements discount logic.
* **country\_model.py:** Manages country data for localization.
* **book\_model.py:** Handles book-related data.
* **author\_model.py:** Manages author details.

3-Controllers/  
Contains logic for processing user inputs and API endpoints:

* **user\_controller.py:** CRUD operations for users.
* **sale\_controller.py:** Handles book sales.
* **review\_controller.py:** Manages reviews and ratings.
* **inventory\_controller.py:** Deals with inventory updates.
* **discount\_controller.py:** Processes discount-related operations.
* **country\_controller.py:** Manages countries and regions.
* **book\_controller.py:** Handles book management.
* **author\_controller.py:** Manages authors.

4-Static/  
Contains static files:

* **css/:** Styling for the web interface.
* **logo/:** The Sprints logo.

5-Templates/  
Holds HTML templates:

* users.html, sales.html, reviews.html, etc.: Templates for different views.
* base.html: Base template for consistent UI structure.

**API Documentation**

**Users API**

* **GET /users**Fetch all users from the database.
* **POST /users**Add a new user. **Request Body:**

****

* **GET /users/<int:user\_id>**Fetch a user by ID.
* **PUT /users/<int:user\_id>**Update user details.
* **DELETE /users/<int:user\_id>**Remove a user by ID.

**Sales API**

* **GET /sales**

Fetch all sales records.

* **POST /sales**

Add a new sale.

**Request Body:**

****

* **GET /sales/<int:sale\_id>**

Fetch a sale record by ID.

* **PUT /sales/<int: sale\_id>**

Update a sale record.

* **DELETE /sales/<int: sale\_id>**

Delete a sale record by ID.

**Authors API**

* **GET /authors**

Fetch all authors records.

* **POST /authors**

Add a new author.

**Request Body:**

****

* **GET / authors /<int: author\_id>**

Fetch an author record by ID.

* **PUT / authors /<int: author\_id>**

Update an author record.

* **DELETE / authors /<int: author\_id>**

Delete an author record by ID.

**Books API**

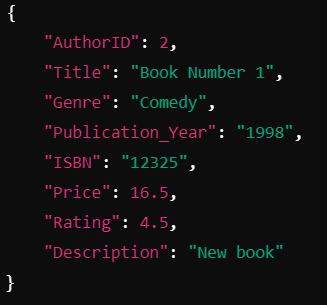
* **GET /books**

Fetch all books records.

* **POST / books**

Add a new book.

**Request Body:**

****

* **GET / books /<int: book\_id>**

Fetch a book record by ID.

* **PUT / books /<int: book\_id>**

Update a book record.

* **DELETE / books /<int: book\_id>**

Delete a book record by ID.

**Books API**

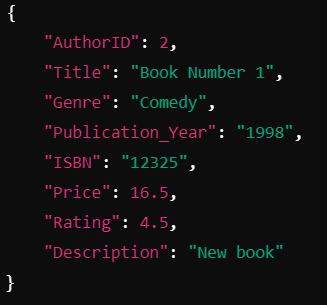
* **GET /books**

Fetch all books records.

* **POST / books**

Add a new book.

**Request Body:**

****

* **GET / books /<int: book\_id>**

Fetch a book record by ID.

* **PUT / books /<int: book\_id>**

Update a book record.

* **DELETE / books /<int: book\_id>**

Delete a book record by ID.

**Country API**

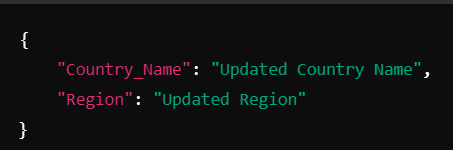
* **GET /countries**

Fetch all countries records.

* **POST / countries**

Add a new country.

**Request Body:**

****

* **GET / countries /<int: country\_id>**

Fetch a book record by ID.

* **PUT / countries /<int: country\_id>**

Update a book record.

* **DELETE / countries /<int: country\_id>**

Delete a book record by ID.

**Discount API**

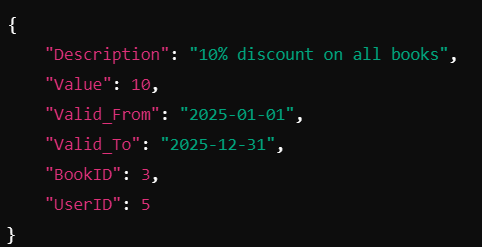
* **GET /discounts**

Fetch all discount records.

* **POST / discounts**

Add a new discount.

**Request Body:**

****

* **GET / discounts /<int: discount\_id>**

Fetch a discount record by ID.

* **PUT / discounts /<int: discount\_id>**

Update a discount record.

* **DELETE / discounts /<int: discount\_id>**

Delete a discount record by ID.

**Key Features**

1. **User Management:** Create, read, update, and delete user profiles.
2. **Book Inventory:** Manage book details, stock levels, and warehouse locations.
3. **Sales Tracking:** Record and retrieve sales data.
4. **Author Management:** Store and update author details.
5. **Dynamic Views:** Responsive templates for an engaging UI.

**Conclusion**

This documentation serves as a guide to understanding the BookBazaar system, its architecture, and its functionalities. The use of the MVC design pattern ensures modularity, scalability, and maintainability of the project.

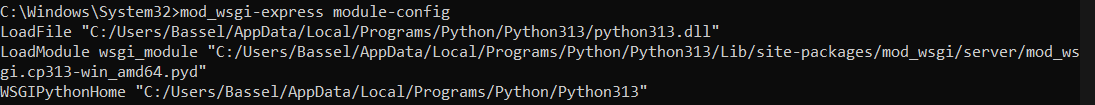
**References**

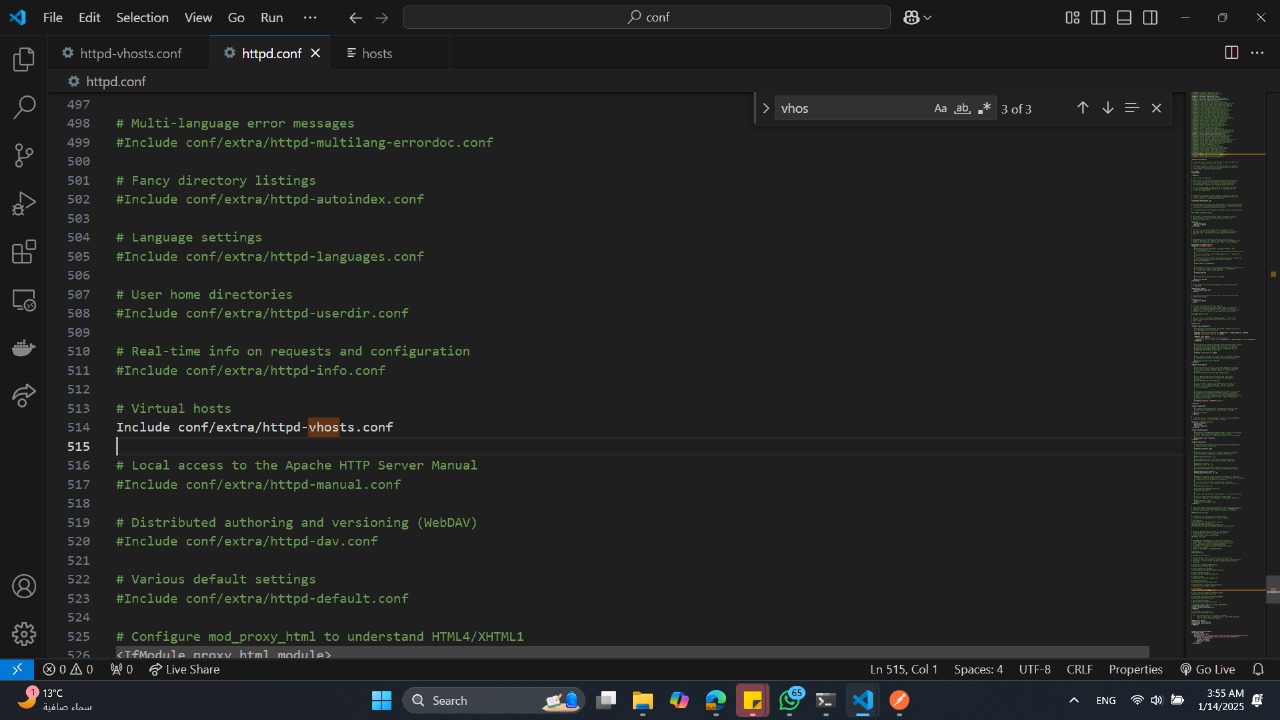
1. Flask Framework  
   Flask Documentation: https://flask.palletsprojects.com/  
   Source for building RESTful APIs and understanding Flask features.
2. SQLite  
   SQLite Documentation: https://sqlite.org/docs.html  
   Reference for creating and managing SQLite databases.
3. MVC Architecture  
   Tutorial: <https://developer.mozilla.org/en-US/docs/Glossary/MVC>  
   Diagram and explanation of the MVC pattern principles.
4. Database Design (Discounts and Countries Tables)  
   Resource on relational database schema design: <https://www.databasejournal.com/>  
   For guidelines and validation constraints applied to table attributes.
5. API Endpoints Design  
   RESTful API Design Best Practices: <https://restfulapi.net/>  
   Source for structuring RESTful API routes and methods.
6. Bootstrap (Optional for Front-End Styling, if used)  
   Bootstrap Documentation: <https://getbootstrap.com/>  
   For UI enhancement in web applications.
7. Diagram Creation Tool  
   MVC Diagram: Created using draw.io (now diagrams.net): https://app.diagrams.net/

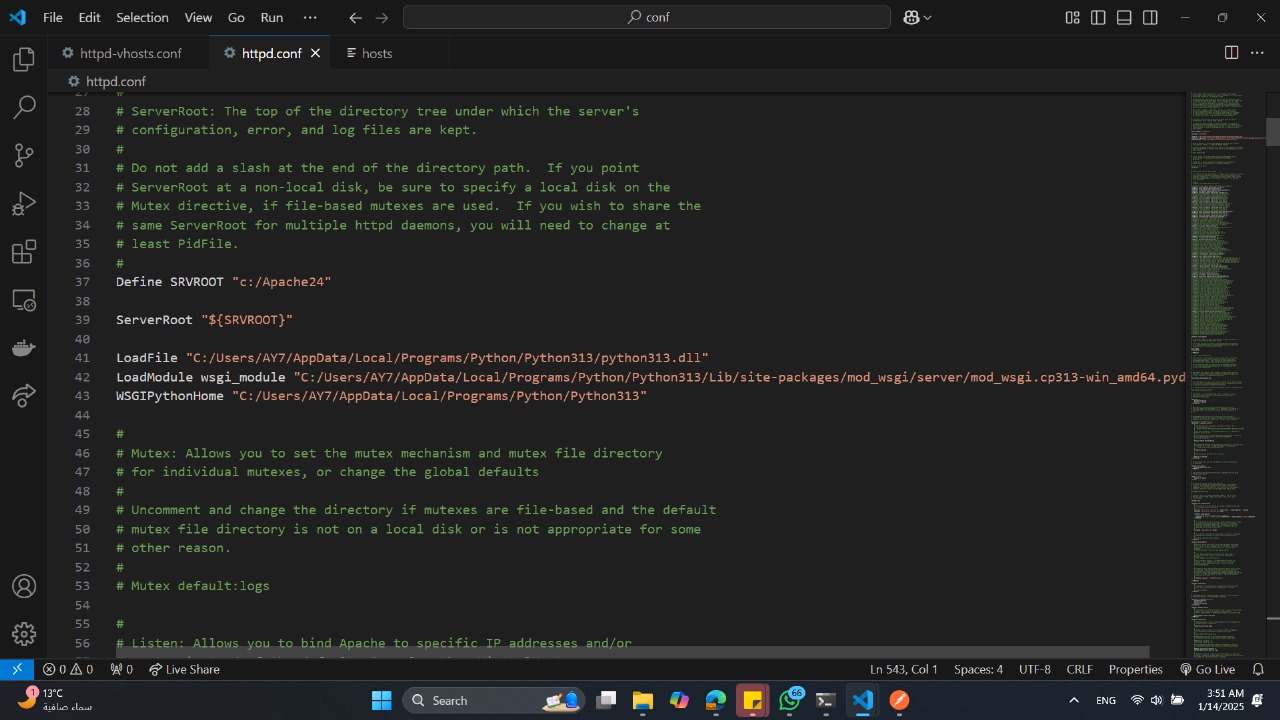
**Part Four: Apache**

**Setting up Flask with Apache using WGSI**

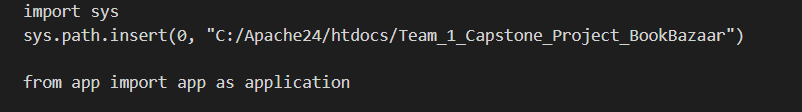
1. **Prerequisites for WSGI Deployment**

* Install Apache for windows
* Install `mod\_wsgi` for Apache
* Verify installation by checking mod\_wsgi version

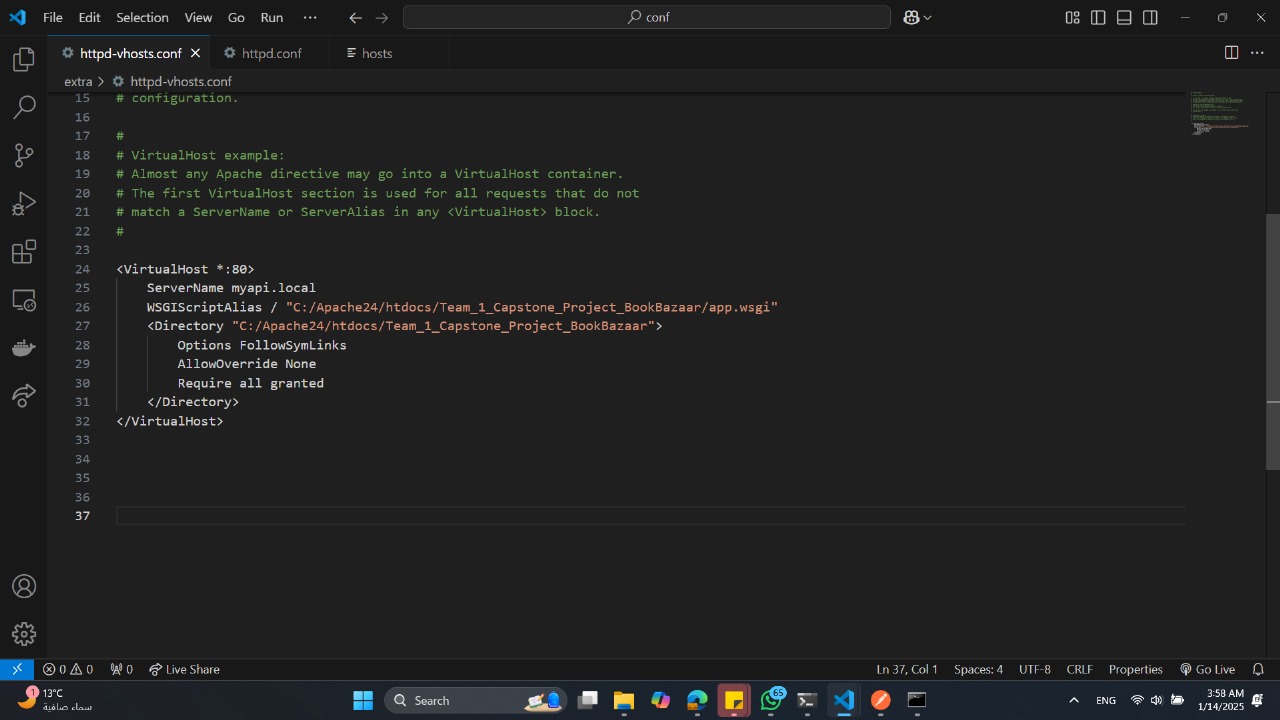
1. **Open the http.conf file located in the conf directory of my Apache installation and add the following lines**

****

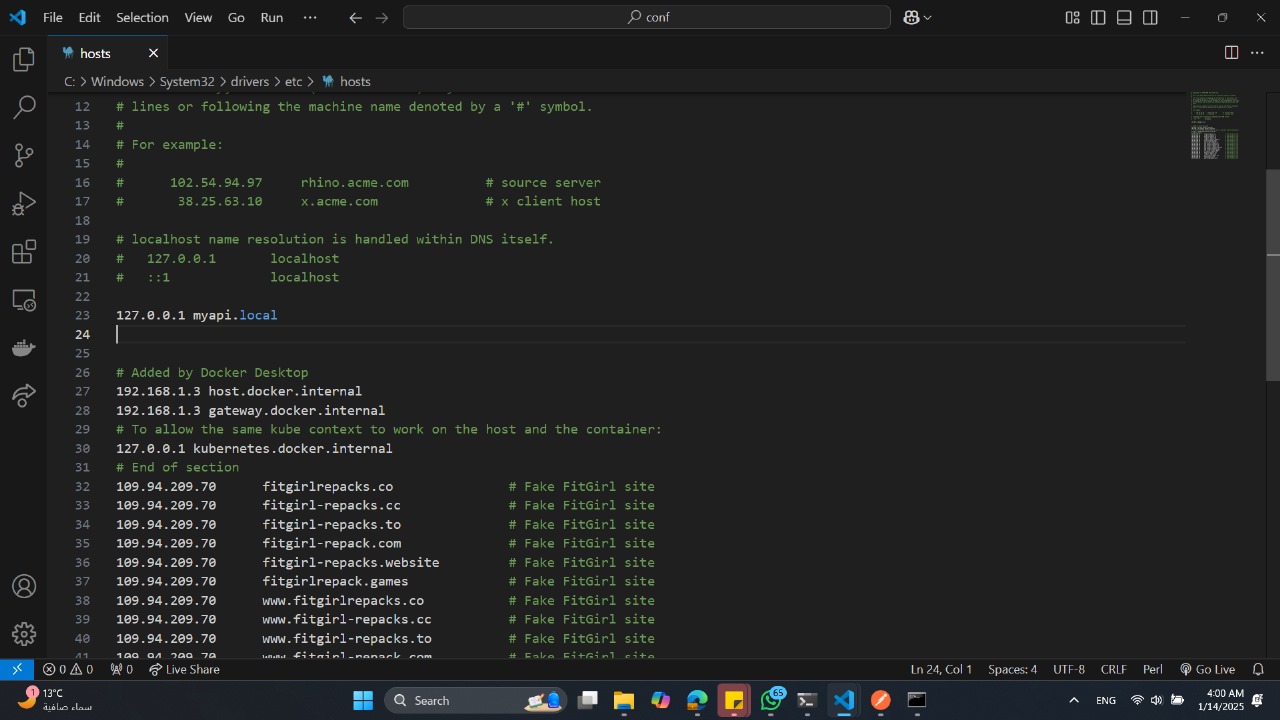
1. **Create a WSGI File**

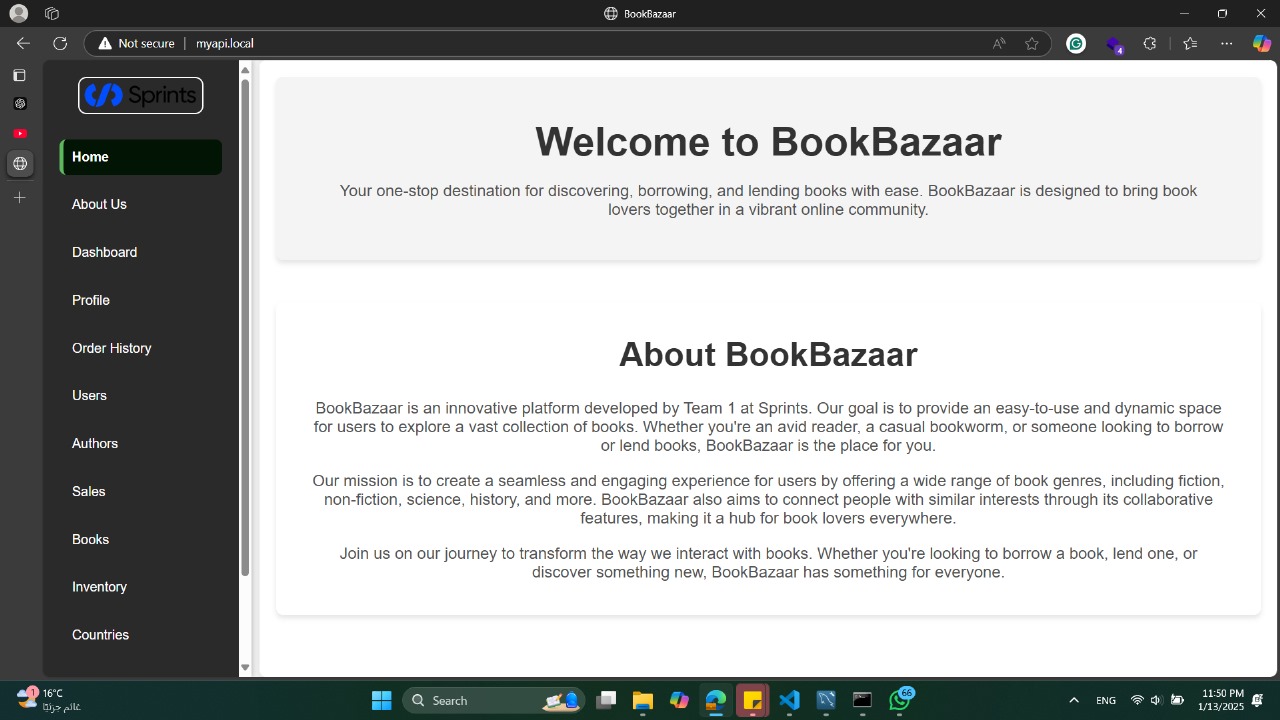
* Create a file named app.wsgi in the same directory as our api.py file
* Add the following content

1. **Setting Up Apache Virtual Host**

* Open the httpd-vhost-conf file located in the conf/extra directory of our Apache installation
* Add the following content

1. **Updating the host file**

* Open the hosts file with administrative privilege’s
* Add a line to map our hostname to myapi.local

1. **Test the hostname by opening the browser and navigation to http://myflaskapp.local**