Sprints

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.

2. Design Goals

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

Sprints

Part One: SQLite

1. Key Features

1- Book Management
- Add, update, delete and retrieve book information
- List all books in a well-formatted output.

2- Authors Management
- Add new authors
- Retrieve details of all authors.

3- Inventory Management
- Track stock quantities by warehouse.
- Update stock for specific books.
- Retrieve inventory summaries for warehouses.

4- Advanced Queries
- Get inventory summaries by location.
- Retrieve books and inventory data based on custom queries.

5- Error Handling
- Comprehensive error messages for database constraints and connection issues.
- Validations for inputs like BookID, AuthorID, and InventoryID.

Sprints

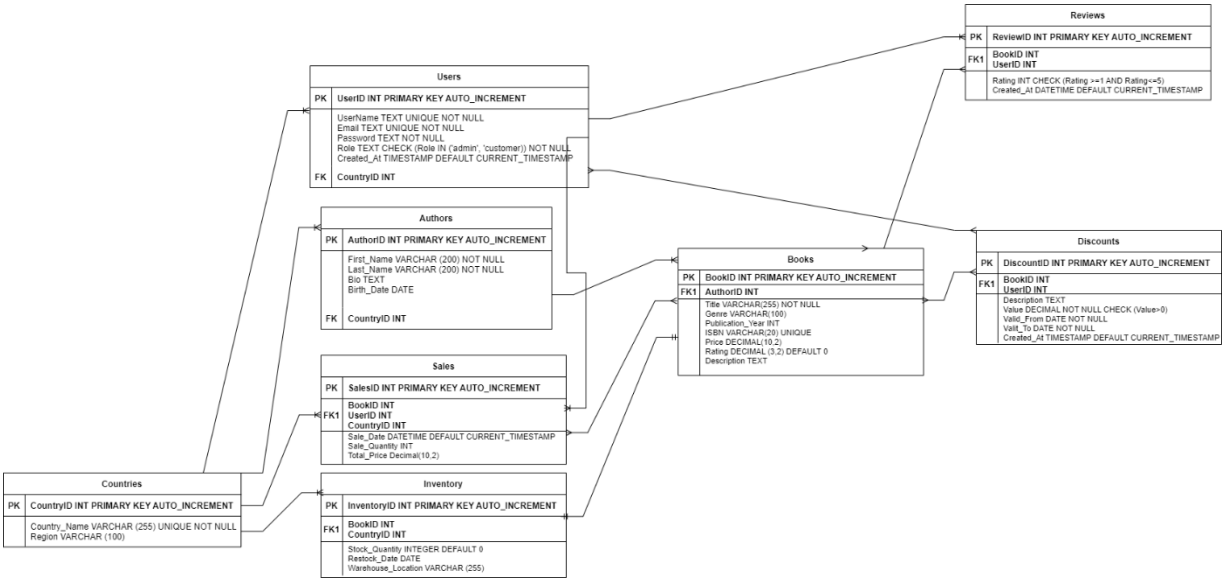
2. Database Setup & Schema Design

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.**

2.2 Task 2: Design the Relational Database Schema

- Data Base Schema



- Entities and Tables

4.1 : Creating Countries Table

Table 1 : Countries

Col_Name	Type	Description
`CountryID`	INT	to store the ID of the country `PRIMARY KEY`
`Country_Name`	VARCHAR	to store the name of the country
`Region`	VARCHAR	to store the regoin where the country belongs to

- **Purpose**:
  - Stores country details for tracking sales and regional performance.

- **Columns Description**:
  - `CountryID` : Unique identifier for each country.
  - `Country_Name` : Name of the country.
  - `Region` : Region to which the country belongs. .

- **Relationshios**:
  - **One-to-Many** (`Countries` → `Authors`): One country can have multiple authors. (`Countries.CustomerID` ---> `Authors.CountriesID`)
  - **One-to-Many** (`Countries` → `Users`): One country can have multiple users.(`Countries.CustomerID` ---> `Users.CountriesID`)
  - **One-to-Many** (`Countries` → `Sales`): One country can have multiple sales.(`Countries.CustomerID` ---> `Sales.CountriesID`)
  - **One-to-Many** (`Countries` → `Inventory`): Inventory entries can be linked to warehouses in specific countries. (`Countries.CustomerID` ---> `Inventory.CountriesID`)

Sprints

4.2 : 👤 Creating Users Table

Table 2 : Users

Col_Name	Type	Description
`UserID`	INT	to store the ID of the user `PRIMARY KEY`
`UserName`	TEXT	to store the username of the user
`Email`	TEXT	to store the email of the user
`Password`	TEXT	to store the password of the user
`Role`	TEXT	to store the role of the user `admin` or `customer`
`Created_At`	TIMESTAMP	to store the nationality of the user
`CountryID`	INT	to link between the User and his country `FOREIGN KEY`

- **Purpose**:
  - Stores user account details, differentiating between customers and admins.

- **Columns Description**:
  - `UserID` : Unique identifier for each user.
  - `UserName` : Unique username for login.
  - `Email` : Unique email address for communication.
  - `Password` : Encrypted user password.
  - `Role` : Defines user permissions (`admin` or `customer`).
  - `Created_At` : Timestamp when the account was created.
  - `CountryID` : Links the user to their country. .

- **Relationshios**:
  - **One-to-Many**: A user can write multiple reviews. (`Reviews.UserID` ---> `Users.UserID`)
  - **One-to-Many**: A user can make multiple purchases. (`Sales.UserID` ---> `Users.UserID`)
  - **Many-to-Many**: Discounts can target specific users. (`Discounts.UserID` ---> `Users.UserID`)
  - **Many-to-One**: Each user belongs to one country. (`Users.CountryID` ---> `Countries.CountryID`)

4.3 : ✍️ Creating Authors Table

Table 3 : Authors

Col_Name	Type	Description
`AuthorID`	INT	to store the ID of the author `PRIMARY KEY`
`First_Name`	VARCHAR	to store the first_name of the author
`Last_Name`	VARCHAR	to store the last_name of the author
`Bio`	TEXT	to store the country of the author
`Birth_Date`	DATE	to store the bith_date of the author
`CountryID`	VARCHAR	to link between the author and the Country `FOREIGN KEY`

- **Purpose**:
  - This table is used to store information about book authors, including their first_name, last_name, biography, birth_date and their nationality

- **Columns Description**:
  - `AuthorID` : Unique identifier for each author.
  - `First_Name` : Author's first name .
  - `Last_Name` : Author's last name .
  - `Bio` : A brief biography of the author .
  - `Birth_Date` : Authors bith date .
  - `CountryID` : Author's nationality .

- **Relationshios**:
  - **One-To-Many** : An author can write multiple books, but each book belongs to only one author. (`Books.AuthorID` ---> `Authors.AuthorID`)
  - **Many_To_One** : Many Authors can belong to the same country but each Author is linked to one Country (`Authors.CountryID` ---> `Countries.CountryID`)

Sprints

4.4 📖 Creating Books Table

Table 4 : Books

Col_Name	Type	Description
`BookID`	INT	to store the ID of the book `PRIMARY KEY`
`Title`	VARCHAR	to store the title of the book
`Genre`	VARCHAR	to store the category of the book
`Publication_Year`	INT	to store puplication year of the book
`ISBN`	VARCHAR	to store the ISBN code of the book
`Price`	DECIMAL	to store the price of the book
`Rating`	INT	to store the rating of the book
`Description`	TEXT	to store the description of the book
`AuthorID`	VARCHAR	to link between the book and it's author `FOREIGN KEY`

- **Purpose**:
  - Stores books details, including title, genre, pricing and their authors

- **Columns Description**:
  - `BookID` : Unique identifier for each book.
  - `Title` : Title of the book.
  - `Genre` : Genre or category of the book.
  - `Publication_Year` : Year of publication.
  - `ISBN` : Unique ISBN code.
  - `Price` : Selling price of the book.
  - `Rating` : Average customer rating
  - `Description` : Brief description of the book.
  - `AuthorID` : Reference to the author.

- **Relationshios**:
  - **Many-to-One**: Each book is written by one author. (`Books.AuthorID` ---> `Authors.AuthorID`)
  - **One-to-One**: Each book has one inventory record. (`Inventory.BookID` ---> `Books.BookID`)
  - **Many-to-Many**: Each book can have multiple reviews. (`Reviews.BookID` ---> `Books.BookID`)
  - **Many-to-Many**: Books can have multiple sales records. (`Sales.BookID` ---> `Books.BookID`)
  - **Many-to-Many**: Many book can have multiple discounts. (`Discounts.BookID` ---> `Books.BookID`)

4.5 📦 Creating Inventory Table

Table 5 : Inventory

Col_Name	Type	Description
`InventoryID`	INT	to store the ID of the inventory `PRIMARY KEY`
`BookID`	VARCHAR	to link between the book and the inventory `FOREIGN KEY`
`Stock_Quantity`	INT	to store the current quantity of the book
`Restock_Date`	DATE	to store the last stock replenishment
`Warehouse_Location`	VARCHAR	to store location of the inventory storing the stock
`CountryID`	INT	to link between the inventory and the country `FORIEGN KEY`

- **Purpose**:
  - Tracks the stock levels and warehouse details of books.

- **Columns Description**:
  - `InventoryID` : Unique identifier for each inventory record.
  - `BookID` : Reference to the book.
  - `Stock_Quantity` : Current stock available.
  - `Restock_Date` : Date of the last stock replenishment.
  - `Warehouse_Location` : Location of the warehouse storing the stock.
  - `CountryID` : Links inventory to a specific country.

- **Relationshios**:
  - **One-to-One**: Each book has one inventory record. (`Inventory.BookID` ---> `Books.BookID`)
  - **Many-to-One** : Inventory is linked to one country. (`Inventory.CountryID` ---> `Countries.CountryID`)

Sprints

4.6 🛒 Creating Sales Table

Table 6 : Sales

Col_Name	Type	Description
SaleID	INT	to store the ID of the sale PRIMARY KEY
Sale_Date	DATETIME	to store the date when the book was sold
Sale_Quantity	INT	to store sold quantity of the book
Total_Price	DECIMAL	to store the total price of the book
BookID	INT	to link between the book and the sales FOREIGN KEY
UserID	INT	to link between the sales and the user FOREIGN KEY
CountryID	INT	to link between the sales and country FOREIGN KEY

- **Purpose**:
  - Tracks sales transactions, including book sold, user details, and country.

- **Columns Description**:
  - `SaleID` : Unique identifier for each sale.
  - `Sale_Date` : Date and time of the sale.
  - `Sale_Quantity` : Number of units sold.
  - `Total_Price` : Total price of the sale.
  - `BookID` : Reference to the book sold.
  - `UserID` : Reference to the user.
  - `CountryID` : Reference to the country.

- **Relationshios**:
  - **Many-to-Many**: Each sale is linked to many book. (`Sales.BookID ---> Books.BookID`)
  - **Many-to-Many**: Each sale is linked to many user. (`Sales.UserID ---> Users.UserID`)
  - **Many-to-One**: Each sale is linked to one country. (`Sales.CountryID ---> Countries.CountryID`)

4.7 💥 Creating Discounts Table

Table 7 : Discount

Col_Name	Type	Description
DiscountID	INT	to store the ID of the discount PRIMARY KEY
Description	TEXT	to store the description of the discount
Value	DECIMAL	to store the value of the discount in percentage
Valid_From	DATE	to store start date of the discount
Valid_To	DATE	to store end date of the discount
Created_At	TIMESTAMP	to store when the discount was made
BookID	INT	to link between the dicsount and the book FOREGIN KEY
UserID	INT	to link between the discount and the user FOREGIN KEY

- **Purpose**:
  - Manages discount campaigns, targeting books and specific users.

- **Columns Description**:
  - `DiscountID` : Unique identifier for each discount.
  - `Description` : Description of the discount.
  - `Value` : Discount percentage.
  - `Valid_From` : Start date.
  - `Valid_To` : End date.
  - `Created_At` : Timestamp
  - `BookID` : Reference to the book.
  - `UserID` : Reference to the user.

- **Relationshios**:
  - **Many-to-Many**: Each discount can target many books. (`Books.BookID ---> Discounts.BookID`)
  - **Many-to-Many**: Each discount can target many user (`Discounts.UserID ---> Users.UserID`)

4.8 📝 Creating Reviews Table

Table 5 : Reviews

Col_Name	Type	Description
ReviewID	INT	to store the ID of the Review PRIMARY KEY
Rating	VARCHAR	to store the rating of the book (1 ->5)
Created_At	INT	to store the date when the review was crerated
BookID	VARCHAR	to link between the review and the book FOREGIN KEY
UserID	DECIMAL	to link between the review and the user FOREGIN KEY

- **Purpose**:
  - Stores customer reviews and ratings for books.

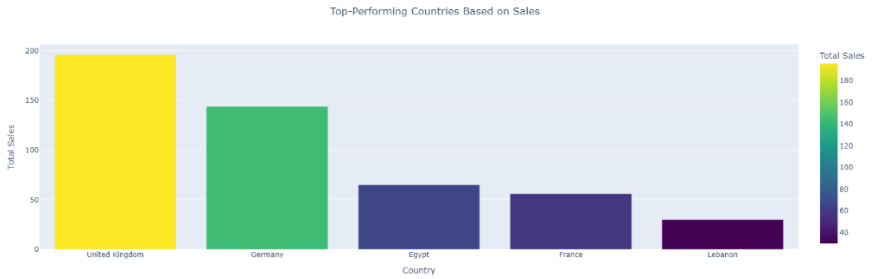
- **Columns Description**:
  - `ReviewID` : Unique identifier for each review.
  - `Rating` : Rating (1-5).
  - `Created_At` : Timestamp.
  - `BookID` : Reference to the reviewed book.
  - `UserID` : Reference to the user

- **Relationshios**:
  - **Many-to-Many**: Each review is linked to one book. (`Reviews.BookID ---> Books.BookID`)
  - **Many-to-One**: Each review is linked to one user. (`Reviews.UserID ---> Users.UserID`)

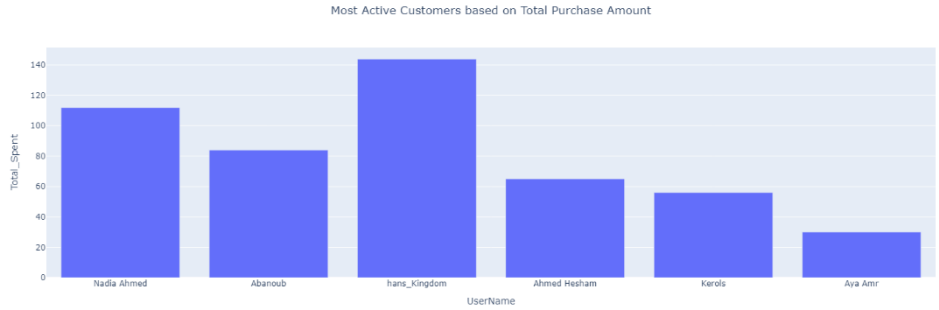
- Triggers:

  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.

3. Insights and Analysis



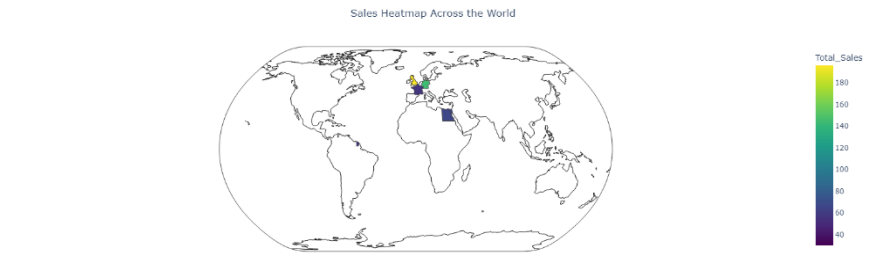
Sprints

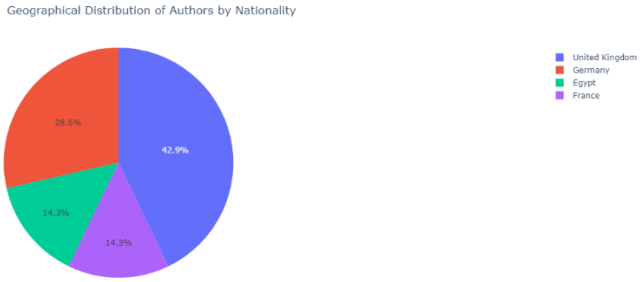

Most Active Customers based on Total Purchase Amount
Total_Spent
0
20
40
60
80
100
120
140
Nadia Ahmed
Abanoub
hans_Kingdom
Ahmed Hesham
Kerols
Aya Amr
UserName

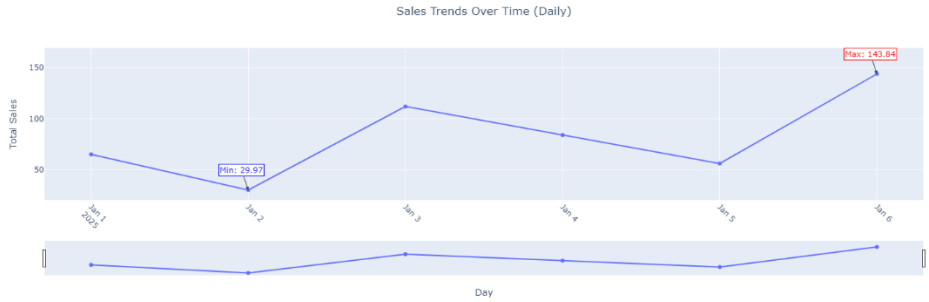

Top-Selling Books (Based on Quantity Sold)
Total Quantity Sold
0
2
4
6
8
10
1984
temp
Animal Farm
El-Ayam
The Plague
Total Quantity Sold
10
8
6
4
Book Name


Most Reviewed Books and Their Average Ratings
Number of Reviews
0
0.5
1
1.5
2
El-Ayam
The Plague
temp
1984
Average_Rating
5
4.5
4
3.5
3
Book Name


Top Performing Authors Based on Sales and Rating
Total Sales
0
20
40
60
80
100
GeorgeOrwell
J.K.Rowling
TahaHussein
NaguibMahfouz
AlbertCamus
Average_Rating
5
4.5
4
3.5
3
Author Name


Sales Heatmap Across the World
Total_Sales
180
160
140
120
100
80
60
40





4. 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.**

Sprints

5. 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.

6. 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.

Sprints

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.

Sprints

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.

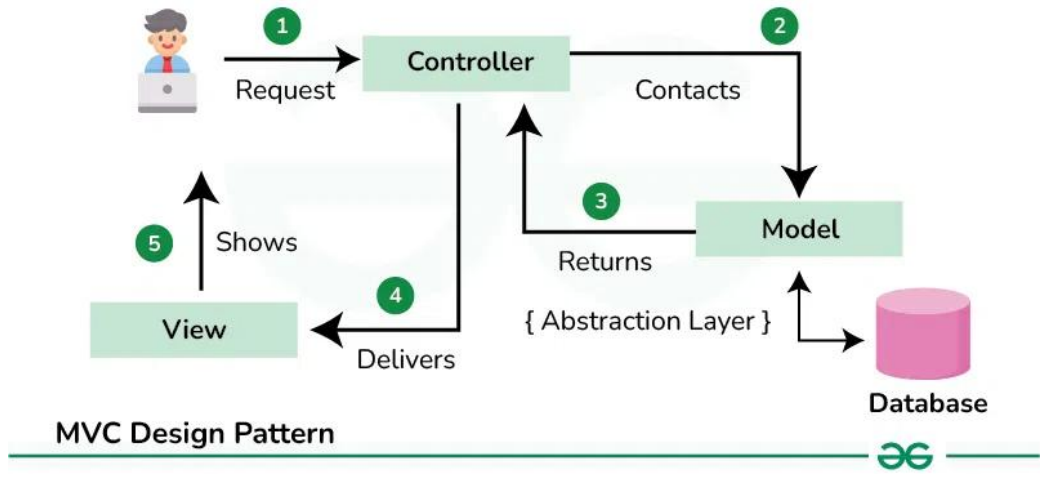
MVC Design Pattern:

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.



Sprints

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.

Sprints

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.

Sprints

API Documentation

Users API

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

```
{
    "UserName": "John Doe",
    "Email": "john@gmail.com",
    "Password": "user1223",
    "Role": "admin",
    "Created_At": "1998-12-12",
    "CountryID": 1
}
```

- **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.

Sprints

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.

Sprints

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.

Sprints

Country API

- **GET /countries**
  Fetch all countries records.
- **POST / countries**
  Add a new country.
  Request Body:

```
{
    "Country_Name": "Updated Country Name",
    "Region": "Updated Region"
}
```

- **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:

```
{
    "Description": "10% discount on all books",
    "Value": 10,
    "Valid_From": "2025-01-01",
    "Valid_To": "2025-12-31",
    "BookID": 3,
    "UserID": 5
}
```

- **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.

Sprints

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/

Sprints

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

```
C:\Windows\System32>mod_wsgi-express module-config
LoadFile "C:/Users/Bassel/AppData/Local/Programs/Python/Python313/python313.dll"
LoadModule wsgi_module "C:/Users/Bassel/AppData/Local/Programs/Python/Python313/Lib/site-packages/mod_wsgi/server/mod_ws
gi.cp313-win_amd64.pyd"
WSGIPythonHome "C:/Users/Bassel/AppData/Local/Programs/Python/Python313"
```

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

```
37  Define SRVROOT "c:/Apache24"
38
39  ServerRoot "${SRVROOT}"
40
41  LoadFile "C:/Users/AY7/AppData/Local/Programs/Python/Python313/python313.dll"
42  LoadModule wsgi_module "C:/Users/AY7/AppData/Local/Programs/Python/Python313/Lib/site-packages/mod_wsgi/server/mod_wsgi.cp313-win_amd64.pyd
43  WSGIPythonHome "C:/Users/AY7/AppData/Local/Programs/Python/Python313"
44
45  #
```

```
512
513  # Virtual hosts
514  Include conf/extra/httpd-vhosts.conf
515
```

3. Create a WSGI File

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

```
import sys
sys.path.insert(0, "C:/Apache24/htdocs/Team_1_Capstone_Project_BookBazaar")

from app import app as application
```

Sprints

4. 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

```
#
# VirtualHost example:
# Almost any Apache directive may go into a VirtualHost container.
# The first VirtualHost section is used for all requests that do not
# match a ServerName or ServerAlias in any <VirtualHost> block.
#

<VirtualHost *:80>
    ServerName myapi.local
    WSGIScriptAlias / "C:/Apache24/htdocs/Team_1_Capstone_Project_BookBazaar/app.wsgi"
    <Directory "C:/Apache24/htdocs/Team_1_Capstone_Project_BookBazaar">
        Options FollowSymLinks
        AllowOverride None
        Require all granted
    </Directory>
</VirtualHost>
```

5. Updating the host file

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

```
127.0.0.1 myapi.local
```

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