Documentation: Library Management System using Flask and PostgreSQL

1. Overview

This project is a backend Library Management System built using Flask and PostgreSQL.
The system allows managing books and members and provides functionality for borrowing and returning books.

2. Tech Stack

- Flask
- PostgreSQL
- SQLAlchemy Core
- Pydantic (Validation Layer)
- Pytest (Testing Framework)
- python-dotenv (Environment Configuration)

3. Install Dependencies

- pip install flask
- pip install sqlalchemy
- pip install psycopg2-binary
- pip install python-dotenv
- pip install pydantic
- pip install pytest
- pip install pytest-cov

4. Project Structure (DDD)

The project follows Domain Driven Design (DDD) to keep the code organized and maintainable.

library-management/
app/
domain/
models/
book.py&member.py
services/
book_service.py& member_service.py

helper/
Exceptions & help_fuctions.py
infrastructure/
db.py
repositories/
book_repository.py
member_repository.py
presentation/
routes/
book_routes.py
Member_routes.py

validators/
book&member_validators.py

__init__.py

tests/
config.py
run.py
main.py
.gitignore
README.md
Makefile
pytest.ini
.env

5. Layers Responsibilities

- **Route Layer:** Handles HTTP requests & responses, Validates input data, and calls the service layer
- **Service Layer:** Business Logic & Validations
- **Repository Layer:** Handles Database Operations using SQLAlchemy Core (No business logic here)
- **Database Layer:** PostgreSQL database (Contains Books and Members tables)

6. PostgreSQL Database Configuration

- Install PostgreSQL: Download and install PostgreSQL. (PostgreSQL Page)
- Create Database: CREATE DATABASE library_db;
- Login to PostgreSQL:psql -U postgres
- Install pgAdmin:(Used for managing the Database using a GUI) PGAdmin Page
- Connect pgAdmin to my Database (library_db)

7. Database Configuration using SQLAlchemy Core

- Create Environment Variables (.env file) for database URL
  DATABASE_URL=postgresql://user:password@localhost/library_db
- Database Setup
  - Configure SQLAlchemy create_engine
  - Define metadata
  - Register tables
  - Connect Flask to PostgreSQL

8. Creating Database Tables

- **Books Table**
  - book_id (Integer, Primary Key)
  - title (String)
  - author (String)
  - is_borrowed (Boolean)
  - borrowed_date (DateTime, Nullable)
  - borrowed_by (Foreign Key to Members)

Members Table

  - member_id (UUID, Primary Key)

- name (String)
- email (String, Unique)

9. Repository Layer Implementation

The repository layer contains only SQLAlchemy Core queries (Database operations with **No** Business logic). This layer aims to create SQLAlchemy Core Functions.

- **Examples of functions:**
  - Create Book / Member
  - Update Book / Member
  - Get By ID
  - Delete Book / Member
  - Search and Pagination

- **Testing Repository Layer, Manual Testing, (**Repository tests verify database operations.)
  - **Test Files**
    tests/repository/test_book_repository.py
    tests/repository/test_member_repository.py

  - **Running Repository Tests**
    python3 tests/repository/test_book_repository.py
    python3 tests/repository/test_member_repository.py

10. Service Layer Implementation

- **The service layer contains business rules (logic) such as:**
  - Prevent borrowing borrowed books
  - Ensure member exists
  - Ensure the book exists
  - Validate unique member email
- **Testing Service Layer, Manual Testing:**
  - **Test Files**
    tests/service/test_book_service.py
    tests/service/test_member_service.py

  - **Running Service Tests**
    python tests/service/test_book_service.py
    python tests/service/test_member_service.py

11. API Routes Layer

- Routes are grouped using ***Flask Blueprints*** to separate book and member functionality.

Blueprint is a modular group of routes, instead of putting all routes for books and members together in the same file, we can split them into separate modules.

- Routes communicate only with the Service Layer and do not access the database directly.
- **Member Endpoints**
  POST /members/
  GET /members/
  GET /members/<member_id>
  PUT /members/<member_id>
  DELETE /members/<member_id>

- **Book Endpoints**
  POST /books/
  GET /books/
  GET /books/<book_id>
  PUT /books/<book_id>
  DELETE /books/<book_id>
  POST /books/borrow/<book_id>/<member_id>
  POST /books/return/<book_id>

12. Input validation using Pydantic

- Install Pydantic using (pip install pydantic)
- Folder name: validator
  - book_validator.py
  - member_validator.py
- Pydantic is used here to validate input data before sending it to the service layer.
- Separate validation models are created for:
  - Create Member
  - Update Member
  - Create Book
  - Update Book
- **Important Rule:** Input data must always be validated before passing to the service layer.
  E.g: data = Model(**request.json)

13. Pagination & Search

- Pagination and search logic are implemented in the repository layer because of its database query logic.
- Added for both GET books and Members Endpoint
    - GET /books?limit=5&offset=0
    - GET /members?search=alice
- Pagination Parameters (Both must be positive numbers.)
    - limit
    - offset
- Search Features
    - Books
        - Search by title
        - Search by author
    - Members
        - Search by name
          Search by email

- **Examples**
  GET /books?limit=5&offset=0
  GET /books?limit=5&offset=5
  GET /books?search=flask
  GET /members?search=alice&limit=3&offset=0

14. Global error handling

- The application handles errors using global exception handling.
- Each error returns a proper HTTP status code and message when an invalid operation is attempted.
- Examples:
    - NotFoundError (e.g: Accessing a non-existing member), 404
    - ValidationError (e.g: Invalid input for create member), 400
    - AlreadyExistsError (e.g: Duplicate email registration), 409
    - BorrowError (e.g: Borrowing a borrowed book)
    - Internal Server Error, 500

15. Automated testing using Pytest

- Pytest is a Python testing Framework.
- Pytest is used to test API routes and system behavior.
- Pytest installation: pip install pytest
    - Check version: pytest --version

- Create a tests folder and then create:
    - **Route Test Files**
      tests/routes/test_book_routes.py
      tests/routes/test_member_routes.py

- Pytest Configuration: create a pytest.ini file and add this content (to **d**etect test files automatically).
  [pytest]
  python_files = test_*.py
- Create a ***shared test client Fixture*** (Fixture is usable setup code). This avoids repeating client setup in every test file.
    - Create tests/conftest.py, and add this content:
      import pytest
      from main import app

      @pytest.fixture
      def client():
        return app.test_client()

  @pytest.fixture is to avoid writing client=app.test_client() in each test file.

- Running Tests: To run pytest, we have 3 options:
    - Run all tests: ***pytest from the project root*** (this will automatically search for any files and functions starting with test_ and run them independently).
    - Run specific file: e.g, pytest tests/routes/test_book_routes.py
    - Run single test (run specific function) pytest -k test_borrow_book
- If you have (No Module named ‘main’ error) when pytest runs, this mean it can’t find main.py in python’s module path. This happens because the project is not treated as a package and missing **init**.py file

Note: for the installation steps, refer to the README.md file in the code