java-book-library

Overview:

Create a web application using Java 17+ (and Gradle), with one of the following frameworks: Spring, Micronaut, Quarkus; no front-end required. The application should provide OpenAPI documentation and an interactive Swagger-UI page. Use a PostgreSQL database for data storage, and deliver it as a Docker image and/or using Docker Compose. Emphasize the proper implementation of separation of concerns, and well-structured and appropriately visible packages. Use a classical layered approach, hexagonal architecture, or any other appropriate architecture for this project.

Requirements:

1. **Java Application**: Build a Java application that follows best practices with a framework of your choice, providing a CRUD REST API.
2. **Gradle**: For dependency management, use Gradle.
3. **OpenAPI**: Document all possible REST interactions.
4. **PostgreSQL Database**: Use PostgreSQL to store data locally.
5. **Docker**: Deliver the application as a Docker image and/or use Docker Compose.

Context:

- A **book** has a **title**, a **genre**, a **price**, and exactly one **author**. (The library has only one copy of each book.)
- An **author** has a **name** and a **date of birth** and can have multiple books.
- A **member** has a **unique username**, an **email**, an **address**, and a **phone number**.
- Every member can loan up to 5 different books at a time. The loan has a lend and return date (consider only whole days, no timestamps).

Tasks:

1. **Set up the Java Application**:
    - Initialize a new Java project using an initializer, a template, or from scratch.
    - Set up Gradle for dependency management, including necessary dependencies and plugins.
2. **Integrate REST and OpenAPI**:
    - Include OpenAPI and provide a Swagger-UI accessible from the base URL of the application.

    - Include all CRUD endpoints for book, author, member, and loan, ensuring appropriate error responses.
3. **PostgreSQL Database**:
    - Set up a local PostgreSQL database using Docker.
    - Create the necessary tables reflecting the entities and relationships described.
4. **Backend Functionality**:
    - Implement all REST CRUD endpoints for book, author, member, and loan.
    - Pay attention to the context restrictions (e.g., unique constraints, relationships).
5. **Framework Usage and Architecture**:
    - Use the framework (and additional libraries if needed) to your advantage.
    - Think about the software architecture, project layout, and structure your code accordingly.
6. **Dockerization**:
    - Create a Dockerfile to containerize the Java application.
    - Set up Docker Compose to manage the Java application and PostgreSQL database as services.
    - Ensure that the application can be run with a single `docker-compose up` command.

Deliverables:

- The source code for the Java application, including Gradle configurations.
- The PostgreSQL database schema and any necessary SQL scripts (if not embedded into the application with migrations).
- A `Dockerfile` for the Java application.
- A `docker-compose.yml` file to run all services needed.
- Documentation (`README.md`) with instructions on how to build and run the application using Docker.

Evaluation Criteria:

- **Functionality**: Does the application meet all the specified requirements?
- **Code Quality**: Is the code correct, well-structured, and easy to understand?
- **Clean Code**: Are best practices of clean code followed?
- **CRUD Endpoints in REST**: Is the REST API integrated correctly and efficiently?
- **Database Integration**: Is PostgreSQL used appropriately for storing data?
- **Framework Usage**: Are the core utilities and best practices of the chosen framework leveraged?
- **Software Architecture**: Does the chosen architecture benefit the project?
- **Dockerization**: Does the Docker setup work smoothly and as expected; could the container run in a Kubernetes environment?
- **Documentation**: Are the instructions clear and complete?

Additional Notes:

- Provide any necessary environment variables in a `.env` file and ensure they are correctly referenced in the Docker setup if needed.
- Use meaningful commit messages and structure your Git history logically if submitting via a Git repository.
- Consider edge cases and error handling to ensure the robustness of the application.
- If requirements are missing, make documented assumptions to complete the project.