ArtShop

Use Cases:

1. **User Registration:**
    - Actors: User, System
    - User provides registration details (username, email, password, full name).
    - System validates the input data and creates a new user account.
    - System sends a confirmation email for account activation.
2. **User Login:**
    - Actors: User, System
    - User provides login credentials (username, password).
    - System verifies the credentials and issues an authentication token.
    - User uses the token to access protected endpoints.
3. **Create Resource:**
    - Actors: User, System
    - Authenticated user sends a POST request to create a new resource.
    - System validates the request data and creates the resource in the database.
    - System returns a success response with the newly created resource.
4. **Review Resource:**
    - Actors: User, System
    - Authenticated user sends a POST request to leave a review for a resource.
    - System validates the review data and associates it with the respective resource.
    - System returns a success response confirming the review submission.
5. **Order Resource:**
    - Actors: User, System
    - Authenticated user sends a POST request to place an order for a resource.
    - System validates the order details and processes the order, updating the resource's availability.
    - System returns a success response confirming the order placement if resources are available with requested quantities.
6. **Search Resource:**
    - Actors: User, System
    - User sends a GET request with search parameters to search for resources.
    - System processes the search query and returns relevant resources based on the search criteria.
    - System returns a success response with the search results.
7. **Delete Resource:**
    - Actors: User, System
    - User sends a DELETE request with painting id parameter to delete the resource.
    - System verifies the user's ownership and authorisation to delete the resource.
    - System processes the delete query.
    - System returns a success response with the delete results.

Functional Requirements:

1. **User Authentication:**
    - Users should be able to register an account.
    - Users should be able to log in and receive an authentication token.
    - Authenticated users should have access to protected endpoints.
2. **CRUD Operations:**
    - Users should be able to perform CRUD (Create, Read, Update, Delete) operations on resources exposed by the API.
    - Supported resources may include users, products, orders, etc.
3. **Data Validation:**
    - The API should validate incoming data to ensure it meets specified constraints.
    - Proper error messages should be returned for invalid requests.
4. **Search Functionality:**
    - Users should be able to search for resources based on various criteria.
    - Search queries should return relevant results in a structured format.
5. **Security:**
    - Access to sensitive endpoints and data should be restricted based on user roles and permissions.
    - Proper measures should be in place to prevent common security vulnerabilities like SQL injection, CSRF attacks, etc.

Non-functional Requirements:

1. **Performance:**
    - The API should be able to handle a high volume of concurrent requests efficiently.
    - Response times should be within acceptable limits under normal load conditions.
2. **Scalability:**
    - The system should be designed to scale horizontally to handle increased load as the user base grows.
    - New instances of the application should be able to seamlessly join the existing infrastructure.
3. **Reliability:**
    - The API should be highly available and resilient to failures.
    - Measures should be in place to ensure data integrity and consistency.
4. **Documentation:**
    - Comprehensive documentation should be provided for developers to understand how to use the API.
    - Documentation should cover endpoints, request/response formats, authentication methods, etc.
5. **Security:**

- Data transmitted over the API should be encrypted using HTTPS.
- Access tokens should have limited lifetimes and be securely stored.

6. **Testing:**

- Integration tests should be implemented for all endpoints using technologies such as testContainers and MockMvc.
- TestContainers should be used to manage dependencies and spin up containerized instances of external services (e.g., databases) for testing.
- MockMvc should be utilized for simulating HTTP requests and testing the API endpoints in a controlled environment.
- Test coverage should include positive and negative test cases to ensure the reliability and stability of the API.

Objects and their relationships

For entities and the relationships please refer to ER diagram;

Layers Description:

1. **Repository Layer:**
   - Description: The repository layer is responsible for handling data access and persistence. It typically consists of interfaces extending Spring Data JPA repositories or custom repository implementations.
   - Responsibilities:
     - Define CRUD operations for interacting with the database.
     - Execute database queries and transactions.
   - Example: UserRepository interface extends JpaRepository<User, Long> to handle user-related database operations.
2. **Service Layer:**
   - Description: The service layer contains the business logic of the application. It acts as an intermediary between the controller and repository layers, performing operations such as validation, transformation, and orchestration of business processes.
   - Responsibilities:
     - Implement business logic and rules.
     - Coordinate data access and manipulation.
     - Handle transaction management.
   - Example: UserService class encapsulates user-related business logic, such as user registration, authentication, and order processing.
3. **Controller Layer:**
   - Description: The controller layer handles incoming HTTP requests and serves as the entry point to the application. It delegates requests to the appropriate service methods and returns HTTP responses.
   - Responsibilities:

        - Define request mapping endpoints and request/response handling logic.
        - Convert service layer responses into appropriate HTTP responses, often by delegating mapping service layer objects to DTOs.
        - Delegate business operations to the service layer by invoking service methods.
    - Example: UserController class contains methods to handle user registration, login, and other user-related actions, mapping to appropriate service methods.

Additionally

DTO Layer:

1. **Description:**
    - The DTO (Data Transfer Object) layer is responsible for defining data structures used for transferring data between different layers of the application, such as between the controller and service layers. DTOs help in decoupling the internal representation of data from the external representation exposed via API endpoints.
2. **Example DTOs:**
    - **PaintingRequest:** Represents the data structure used for transferring painting-related information from clients to the server when creating or updating a painting.
    - **PaintingResponse:** Represents the data structure used for transferring painting-related information from the server to clients in response to requests, such as when fetching a list of paintings.

Mappers

PaintingMapper:

1. **Description:**
    - The PaintingMapper is responsible for mapping between entity objects (e.g., database entities) and DTOs. It helps in converting data from one representation to another, facilitating communication between the service and controller layers without exposing the internal details of the entities.
2. **Responsibilities:**
    - Map entity objects to DTOs when transferring data from the service layer to the controller layer (e.g., converting Painting entities to PaintingResponse DTOs).
    - Map DTOs to entity objects when transferring data from the controller layer to the service layer (e.g., converting PaintingRequest DTOs to Painting entities).
    - Handle mapping of complex object structures and nested properties as needed.