**REST API for Article Management - Project Documentation**

**Introduction**

This project involves the development of a RESTful API for managing articles, built using .NET 8 and following the principles of Clean Architecture. The API provides CRUD operations for articles, along with additional functionalities such as pagination, validation, JWT-based authentication, and custom error handling. The project uses the CQRS pattern with MediatR for handling commands and queries, ensuring a clear separation of responsibilities.

**Project Structure**

**1. Core Layer**

* **Purpose:** The Core layer contains the core business logic and entities of the application. It is independent of other layers and does not depend on any external libraries, ensuring that the core logic remains reusable and testable.
* **Key Components:**
  + **Entities:** The Article entity is defined here with properties such as Id, Title, Content, and PublishedDate. These properties represent the fundamental data that the application manages.
  + **Interfaces:** The IArticleRepository interface is defined here to abstract the data access operations. This ensures that the business logic in the application layer is decoupled from the underlying data storage mechanisms.

**2. Application Layer**

* **Purpose:** The Application layer implements the application logic, including commands, queries, validators, and authentication services. It orchestrates the interactions between the Core layer and the Infrastructure layer using the CQRS pattern.
* **Key Components:**
  + **Commands:**
    - **CreateArticleCommand:** Handles the creation of new articles. This command contains properties like Title and Content and is responsible for encapsulating all data required for creating an article.
    - **UpdateArticleCommand:** Handles the updating of existing articles. This command includes the article's Id along with properties like Title and Content that need to be updated.
    - Both commands are handled by MediatR handlers that contain the business logic for creating and updating articles, respectively.
  + **Queries:**
    - **GetAllArticlesQuery:** Retrieves a paginated list of articles from the database. This query is responsible for handling pagination and filtering logic.
    - **GetArticleByIdQuery:** Retrieves a single article by its ID. This query ensures that the requested article exists and returns the appropriate data.
  + **MediatR Integration:**
    - The project uses MediatR to implement the CQRS pattern. Commands and queries are dispatched through MediatR, which in turn invokes the appropriate handlers to execute the business logic.
    - The MediatR pipeline also includes behaviors for cross-cutting concerns, such as validation and logging, ensuring a clean separation of responsibilities.
  + **Auth Service:**
    - The AuthService is responsible for handling user authentication. It integrates with ASP.NET Core Identity to validate user credentials and generate JWT tokens. The AuthenticateUserAsync method takes a username and password, verifies them, and returns a JWT token if the credentials are valid. This token is then used to authenticate subsequent API requests.
  + **Token Service:**
    - The TokenService is responsible for generating JWT tokens. It uses the JwtSecurityTokenHandler to create tokens based on claims such as UserName and UserId. The tokens are signed with a symmetric security key and have a configurable expiration time.
  + **Mapping:**
    - AutoMapper is used to map requests (from the Business layer) to commands (in the Application layer). For instance, CreateArticleRequest is mapped to CreateArticleCommand, and UpdateArticleRequest is mapped to UpdateArticleCommand. This mapping ensures a clean separation between the API’s input models and the application’s business logic.
  + **Pagination:**
    - The PaginatedResult<T> class is implemented to handle the pagination of results when retrieving large lists of articles. It includes properties like Items, PageIndex, PageSize, and TotalCount to provide metadata about the paginated results. This improves the performance and usability of the API when dealing with large datasets.

**3. Business Layer**

* **Purpose:** The Business layer contains the controllers, request models, and custom middleware for error handling. It is where the application’s API-facing logic resides and where interaction with the Application layer occurs.
* **Key Components:**
  + **Controllers:**
    - The ArticleController implements RESTful endpoints for managing articles. It includes methods for retrieving all articles, retrieving a single article by ID, creating a new article, updating an existing article, and deleting an article by ID. The controller interacts with the MediatR commands and queries to perform these operations.
    - The controller also uses the [Authorize] attribute on certain endpoints to enforce JWT-based authentication, ensuring that only authenticated users can access or modify resources.
  + **Request Models:**
    - **CreateArticleRequest:** Represents the data required to create a new article. This model is sent from the client to the API and is mapped to CreateArticleCommand in the Application layer.
    - **UpdateArticleRequest:** Represents the data required to update an existing article. This model is mapped to UpdateArticleCommand.
    - These request models are simple DTOs that help in transferring data from the API layer to the business logic.
  + **Error Handling Middleware:**
    - Custom middleware is implemented to handle exceptions globally across the application. The ErrorHandlingMiddleware catches unhandled exceptions, logs them, and returns an appropriate HTTP status code with a user-friendly error message. For example, a ValidationException might result in a 400 Bad Request response, while an UnauthorizedAccessException would result in a 401 Unauthorized response.
    - This middleware improves the robustness of the API by providing consistent and meaningful error responses.

**4. Infrastructure Layer**

* **Purpose:** The Infrastructure layer is responsible for data access, dependency injection, and any external services required by the application. It implements the interfaces defined in the Core layer and integrates with external libraries and frameworks.
* **Key Components:**
  + **Repositories:**
    - The ArticleRepository class implements the IArticleRepository interface. It interacts with the in-memory database to perform CRUD operations on articles. By following the repository pattern, data access logic is separated from the business logic, making the application more maintainable and testable.
  + **Data Seeding:**
    - The DataSeeder class seeds the in-memory database with initial data for articles and users. This is useful for development and testing, ensuring that the application has predefined data to work with.
  + **Dependency Injection:**
    - The Infrastructure layer configures services for dependency injection, which is a core feature of ASP.NET Core. Services like ArticleRepository, AuthService, and TokenService are registered in the DI container, ensuring that dependencies are resolved automatically at runtime.

**Authentication and Authorization**

**JWT Authentication**

* The API uses JWT (JSON Web Tokens) for authentication. Upon successful login, the AuthService generates a JWT token containing user-specific claims. This token is then included in the Authorization header of subsequent requests to secure endpoints.
* The TokenService is responsible for generating the token, which includes claims such as UserName and UserId, and is signed using a symmetric key. The token has an expiration time (e.g., 1 hour) and is validated on each request to ensure it has not expired or been tampered with.
* Endpoints that require authentication are decorated with the [Authorize] attribute, ensuring that only authenticated users can access them.

**Authorization**

* Authorization is enforced at the endpoint level using the [Authorize] attribute. The API checks the validity of the JWT token before allowing access to protected resources. If the token is invalid or expired, the API returns a 401 Unauthorized response.

**Pagination**

**Paginated Result**

* The PaginatedResult<T> class is a utility that helps manage pagination for API responses. When fetching large datasets, the API can return a subset of results along with metadata such as the current page, page size, and total item count. This is especially useful for endpoints like GET /api/articles where the number of articles could be large.
* The class encapsulates both the data (Items) and the pagination metadata (PageIndex, PageSize, TotalCount), making it easy to include pagination in API responses.

**Example Usage**

* The GetAllArticlesQueryHandler uses this class to return paginated results when fetching all articles. Clients can specify pageIndex and pageSize as query parameters to control pagination, enhancing performance and user experience.

**Error Handling**

**Custom Middleware**

* The ErrorHandlingMiddleware is a custom middleware that intercepts exceptions thrown during request processing. It catches unhandled exceptions, logs them for debugging purposes, and returns a structured JSON response with an appropriate HTTP status code and error message.
* The middleware handles different types of exceptions differently. For example, validation errors might return a 400 Bad Request, while authentication errors return a 401 Unauthorized. This ensures that clients receive meaningful feedback when something goes wrong.

**Example Response**

* A typical error response might look like this:

{

"statusCode": 400,

"message": "Validation error: Title is required.",

"details": "Please provide a valid title for the article."

}

**Clean Architecture**

**Principles**

* The project follows Clean Architecture principles, which emphasize separation of concerns, testability, and maintainability. The architecture is divided into layers, with each layer having a specific responsibility.
* **Core Layer:** Contains the business entities and interfaces, which are independent of other layers. This layer is the most abstract and least likely to change.
* **Application Layer:** Implements the use cases and business logic. It depends on the Core layer but is independent of the Infrastructure layer.
* **Business Layer:** Contains the web-facing aspects of the application, such as controllers and middleware. It interacts with the Application layer to execute business logic.
* **Infrastructure Layer:** Contains the implementation details such as data access, dependency injection, and external services. This layer depends on both the Core and Application layers.

**Benefits**

* **Maintainability:** The separation of concerns allows developers to make changes in one part of the system without affecting other parts.
* **Testability:** The clear separation between layers makes it easier to write unit tests, as dependencies can be mocked or stubbed.
* **Scalability:** The architecture allows for easy addition of new features or changes in implementation details, such as switching from an in-memory database to a real database, without impacting the core business logic.