Technical Task Sample

Problem: Building a Product Management API

Objective

Design and implement a RESTful API for managing products in an e-commerce application. The API should allow clients to perform CRUD (Create, Read, Update, Delete) operations on products while ensuring data integrity and validation.

Requirements

Entities

1. Product
    - Properties:
        - `Id` (int, primary key)
        - `Name` (string, required, max length 100)
        - `Description` (string, optional, max length 500)
        - `Price` (decimal, required, must be greater than 0)
        - `StockQuantity` (int, required, must be greater than or equal to 0)
        - `CreatedAt` (DateTime, auto-generated)
        - `UpdatedAt` (DateTime, auto-generated on update)

API Endpoints

1. GET /api/products
    - Returns a list of all products.
    - Supports optional query parameters for pagination (`page`, `pageSize`) and filtering by `name`.
2. GET /api/products/{id}
    - Returns a single product by its ID.
    - Responds with a 404 status code if the product is not found.
3. POST /api/products
    - Creates a new product.
    - Request body must include `Name`, `Description`, `Price`, and `StockQuantity`.
    - Responds with a 201 status code and the created product object.
    - Validates input data and returns appropriate error messages for invalid data.
4. PUT /api/products/{id}
    - Updates an existing product.

- Request body must include `Name`, `Description`, `Price`, and `StockQuantity`.
- Responds with a 404 status code if the product is not found.
- Responds with a 200 status code and the updated product object.
- Validates input data similar to the POST method.

5. DELETE /api/products/{id}
    - Deletes a product by its ID.
    - Responds with a 204 status code if successful.
    - Responds with a 404 status code if the product is not found.

Data Validation

- Implement custom validation attributes for the properties of the Product entity:
    - Ensure that the `Price` is greater than zero.
    - Ensure that the `StockQuantity` is not negative.

Database Context

- Use Entity Framework Core to manage database interactions.
- Create a DbContext class named `ProductDbContext` that includes a DbSet<Product> for managing products.

Testing

- Write unit tests for each endpoint using an in-memory database to ensure that:
    - All CRUD operations function as expected.
    - Validation rules are enforced correctly.
    - Error handling works properly (e.g., returning appropriate status codes).

Documentation

- Use Swagger/OpenAPI to document your API endpoints.
- Include examples of requests and responses for each endpoint.