Application Development Exercise: Policy Management

Objectives:

Build a simple policy management application that allows users to create, read, update, and delete (CRUD) policies. This application should use:

- **Latest Angular** for the front-end.
- **Latest .NET (ASP.NET Core)** for the API.
- **Entity Framework Core** for database interactions.
- **Microsoft SQL Server** as the database.
- **Repository Pattern** for data access.
- **Dependency Injection** for managing dependencies.
- **Asynchronous Methods** for database operations.
- **Transaction Handling** to ensure data consistency.
- **API Security** to protect endpoints.
- **Bootstrap** for CSS styling.

Requirements:

1. **Front-End (Angular):**
    - **Create a single-page application (SPA) using Angular.**
    - **Bootstrap should be used for CSS styling and layout. Ensure a responsive design.**
    - **The application should have the following components:**
        - **A form to create and update policies.**
        - **A list view to display policies with options to edit and delete.**
    - **Implement basic validation for policies creation and editing forms.**
    - **Implement routing for different views (e.g., policy List and policy details Details).**
2. **Back-End (.NET API):**
    - **Develop a RESTful API using ASP.NET Core.**
    - **The API should expose endpoints for:**
        - **Getting all policies with pagination (GET /api/policies)**
        - **Getting a specific policy by ID (GET /api/policies/{id})**
        - **Creating a new policy (POST /api/policies)**
        - **Updating an existing policy (PUT /api/policies/{id})**
        - **Deleting a policy (DELETE /api/policies/{id})**
    - **Repository Pattern:**
        - **Implement the repository pattern to abstract the data access layer.**
        - **Create interfaces for repositories and implement them.**
    - **Dependency Injection:**
        - **Use dependency injection to manage repository and service instances.**
    - **Asynchronous Methods:**
        - **Use async and await for asynchronous database operations.**
    - **Transaction Handling:**
        - **Implement transaction handling to roll back changes in case of failure.**
    - **API Security:**
        - **Secure API requests using basic authentication or JWT (JSON Web Token).**
3. **Database (SQL Server with Entity Framework Core):**
    - **Design a database schema for policies. Start with the below table and elaborate more regarding the remaining entities (like: Policy Members, Submitted Claims etc..):**
        - **Id (int, primary key, auto-increment)**
        - **Name (string, required)**

- **Description (string, optional)**
- **Creation Date (DateTime, Required)**
- **EffectiveDate (DateTime, Required)**
- **ExpiryDate (DateTime, Required)**
- **PolicyType (uniqueidentifier, Required)**

  - **Use Entity Framework Core for database operations.**
  - **Configure the application to use SQL Server and apply migrations.**

Deliverables:

1. **Source Code (on Github):**
   - **Provide the source code for both the Angular front-end and the .NET API.**
   - **Ensure that the code is well-organized, follows best practices, and includes comments where necessary.**
2. **Instructions for Running the Application:**
   - **Include a README file with instructions on how to set up and run the application locally.**
   - **Provide details on any required configurations or dependencies.**
3. **Deployment Instructions (Optional but recommended):**
   - **Describe how to deploy the application to a cloud service or a local server if applicable.**

Bonus Points:

1. **Apply multi-tenancy:**
   - **Each tenant has its own database**
2. **Advanced Security:**
   - **Implement advanced security features.**
3. **Error handling and Logging:**
   - **Include a mechanism for logging errors and monitoring application health.**
4. **Documentation:**
   - **Provide comprehensive documentation for both the API and the front-end application.**

Evaluation Criteria:

1. **Functionality:**
   - **Does the application meet all the requirements?**
   - **Are all CRUD operations working as expected?**
2. **Code Quality:**
   - **Is the code clean, well-structured, and maintainable?**
   - **Are proper design patterns, such as the repository pattern, used effectively?**
3. **UI/UX:**
   - **Is the user interface intuitive and user-friendly?**
   - **Is Bootstrap used effectively for styling and responsiveness?**
4. **API Design:**
   - **Are the API endpoints well-defined and RESTful?**
   - **Is there proper error handling, validation, and transaction management?**
5. **Database Design:**
   - **Is the database schema normalized and efficient?**
   - **Are Entity Framework Core conventions and best practices followed?**
6. **Security:**
   - **Are API endpoints secured with authentication and authorization?**

- **Are sensitive data and operations protected against common vulnerabilities?**

7. **Performance:**
   - **Is the application optimized for performance and responsiveness?**
   - **Are asynchronous methods and transactions used appropriately?**
8. **Testing:**
   - **Although not explicitly required, evaluate if the candidate has included any unit or integration tests for the API or front-end.**