**28. Employee Management Application .NET Core**

**Create a sample application with following functionality in .Net Core.**

* Need to create Employee Management web application.
* Need to create one .Net Core project, one Web API project (In .Net Core) and one class library for database.
* Hierarchy of request is, cursor will go to MVC controller using AJAX call, from controller to go to Web API and from Web API to class library for save and manage the data.
* Need to create following tables into local database.
  + Users:
    - Id
    - Email
    - Password
    - Created Date
  + Employee Designation
    - Id
    - Designation
  + EmployeeDetails
    - Id
    - Name
    - DesignationId (store designation Id into column)
    - ProfilePicture (Save picture into local machine and store that path into database)
    - Salary
    - Date Of Birth
    - Address
    - Email

**Architecture Overview:**

The Architecture of The Given Project Consists of three main components: a .NET Core project, a Web API Project, and a class library for database operations. The components work together to create an Employee management Web application

**The hierarchy of request flow is as follows**:

* User interface -> Controller ->REST API->Services layer -> Repository ->database

1. **User Interface(UI):**

* The user interacts with the Application through the UI, which typically consists of view, forms, and user input elements.
* In this case, the UI is implemented using ASP.NET MVC and consists of the views and actions defined in the “Employee Controller”.
* Whenever you will debug, your cursor will be going from **UI side to controller** only. Not other place.

1. **Controller:**

* The controller in MVC pattern the receive the user’s request performs necessary action s and determine the appropriate response.
* In this case, the controller handles the HTTP request related to employee management, such as adding, updating, deleting, and retrieving employees.
* Whenever you will debug, your cursor will be going from Controller side to **REST API** side and **Token interceptor** side for Authorize the method Header.

1. **RESTAPI :**

* Rest API project is a responsible for exposing endpoints that allows clients to interact with the applications through HTTP requests.
* Each controller exposes methods that corresponds to different HTTP methods(GET ,POST ,PUT,PATCH,DELETE) and interact with services to perform the required operation
* A Restful API (Representational State transfer) is an architectural style for designing networked applications that provide interoperability between systems over the internet. It follows a set of principles and constraints to create web service that are scalable, stateless, and can be easily consumed by clients.
* The API project may also include authentication and authorization ,mechanism to secure the endpoints
* Whenever you will debug, your cursor will be going from REST API Side To Service layer only. Not other place.

1. **Services Layer:**

* The services layer encapsulates the business logic of the application.
* It receives the request from the REST API controller and processes them and interacts with the repository to perform data access operations
* Whenever you will debug, your cursor will be going from Service layer To Repository Layer only. Not other place.

1. **Repository:**

* The repository is responsible for data access operation, interaction with the database, or any other data storage mechanism.
* It provides a method to perform CRUD operations on employee data.
* In this case, the Employee repository class handles the database interactions using the Dapper to execute the store procedure and retrieve and update employee data
* Whenever you will debug, your cursor will be going to database for process of data only. Not other place.

1. **Database:**

* The actual database stores the employee data persistently.
* In this case, a SQL server database is used, and the connection string is provided in the repository class
* We need to create stored procedure for each and every process. You can’t use native query directly for database process.
* In summary, the request flows start from the UI, where the user interacts with the application the controller receives the user request, and delegates appropriate action to the REST API project. The REST API is exposing end points for external access. The service layer, in turn, interact with the repository to perform data access operations. Finally, the repository communicates with the database to retrieve or update the employee data. The response flows back in the reverse order, from the repository to the service layer then the REST API to the controller, and ultimately to the UI for rendering the appropriate view or response to the user. This layered approach allows for easier maintenance ,testing and scalability of the application

**Project References:**

***You need to create multiple projects for different purposes***

1. **EmployeeManagementMvc**:

* Project will be created as Web application with MVC.
* It is useful for User interface and Controller module.
* Need to create User Interface and MVC controller in this project.
* Need To Create Token Interceptor Class in project form Authorize Header for Access

API Endpoints.

1. **Employee Repositories:**

* Need to add console application into same solution with this name.
* Also, need to create 2 folder. One is for **Implementations** and one is for **interface**.
* In **Implementations**, you need to create your repository methods and in **interface** you need to create interface of all methods.
* Also, inherit that **interface** into repository class.

1. **Employee Services:**

* Need to add console application into same solution with this name.
* Also, need to create 2 folder. One is for **Implementations** and one is for **interface**.
* In services, you need to create your **Service** methods and in **interface** you need to create interface of all methods.
  + - Also, inherit that interface into services class.

1. **EmployeeMangementWebApi :**

* Project will be created as Web application with REST API.
* It is useful for Handling HTTP Request (GET, POST, PUT, and DELETE) and Response.
  + - Need to create Api controller and Authorize and Authenticate Api end points in [program.cs] this project.

1. **Common Model:**

* Need to add console application into same solutions with this name.
* Need to create 2 folder. One is for DB Model, one is for View model.
* In **Common** Model folder, you need to create DB Model and add all properties which is in database.
* In view model folder, you need to create view model and add all properties which is useful in UI.

**Dapper:**

1. **What is dapper and how to dapper in ASP.NET MVC?**

Dapper usually refers to the Dapper ORM (Object-Relational-Mapper) framework for database access in .NET applications it is the same as the entity framework.

Add dapper in your project install Nuget package=Dapper.

2. **Dependency Injection:**

In .NET Core, a dependency refers to an external component or service that your application relies on to perform certain tasks or provide specific functionality. Dependencies can include libraries, frameworks, packages, or even other modules within your own application.

Dependency management in .NET Core is handled through a dependency injection (DI) container. The DI container is responsible for resolving and injecting dependencies into the classes or components that need them. It helps decouple the components of your application, making them more modular and easier to maintain.

***In ASP.NET Core Dependency injection is inbuilt Feature we Not Use external Framework.***

**Authentication and Authorization**

1. **Authentication:**

Authentication is the process of verifying the identity of users or clients accessing your application. It ensures that users are who they claim to be. In .NET Core, you can implement authentication using various authentication schemes, such as:

1. Cookie-based Authentication

2. Token-based-Authentication

3. External-Authentication provider

*In* ***this we are use Token Based Authentication***

* **What is Token-Based Authentication:**

Token-based authentication involves issuing and validating tokens to authenticate users.

JSON Web Tokens (**JWT**) are commonly used in token-based authentication.

The Add Authentication method can be used to enable token-based authentication, and

You can configure authentication options such as token validation parameters, issuer, audience,

And signing keys.

1. **Authorization:**

Authorization determines what actions or resources a user is allowed to access within an application. It controls user permissions and protects sensitive functionality or data. In .NET Core, you can implement authorization using various approaches:

* Role-based Authorization:
* Policy-based Authorization:
* Claims-based Authorization:
* Resource-based Authorization:

These authentication and authorization mechanisms can be configured in the builder. Services method of the Program class in your .NET Core application. Additionally, you can apply authorization attributes to controllers and actions to control access at the granularity level required by your application's security requirements.

1. **JWT Token:**

JWT (JSON Web Token) is a widely used standard for securely transmitting information between parties as a JSON object. It is commonly used for authentication and authorization in web applications. In .NET Core, you can easily work with JWT tokens using various libraries and frameworks. Here's an overview of using JWT tokens in .NET Core:

**Generating JWT Tokens:**

To generate JWT tokens, you typically need a secret key and some claims (payload data) to include in the token.

You can use libraries like System.IdentityModel.Tokens.Jwt or popular third-party libraries like Microsoft.AspNetCore.Authentication.JwtBearer to generate JWT tokens.

When generating tokens, you can set the issuer, audience, expiration time, and include any additional claims relevant to your application.

**Authentication using JWT Tokens**:

In .NET Core, you can authenticate requests using JWT tokens using the Microsoft.AspNetCore.Authentication.JwtBearer library.

Configure the authentication middleware in the builder. Services method of the Start-up class using the Add Authentication and AddJwtBearer methods.

Specify the token validation parameters, such as issuer, audience, signing key, and whether to validate the token's expiration, issuer, and audience.

Enable the authentication middleware using the Use Authentication method in the request pipeline.

**Authorization using JWT Tokens:**

Once authentication is set up, you can use JWT tokens for authorization.

Apply the [Authorize] attribute to controllers or actions to restrict access to authenticated users with valid JWT tokens.

Optionally, you can apply additional policies or role-based authorization alongside JWT-based authentication for more fine-grained control over access.

**Token Validation and Security:**

When receiving a request with a JWT token, the authentication middleware validates the token's signature and checks its validity based on the configured parameters.

It's important to securely store and manage the secret key used for signing the tokens. Avoid exposing the key publicly or committing it to source control.

You can refresh tokens by issuing new tokens when the existing token expires, reducing the need for repeated authentication

Overall, working with JWT tokens in .NET Core involves generating tokens When User login successfully, configuring authentication and authorization middleware, validating tokens, and leveraging the claims within the tokens to make authorization decisions. The provided libraries and frameworks simplify the process and provide a secure and scalable solution for authentication and authorization in your .NET Core applications

**Project on GitHub:**

[GitHub - jigar-prajapati1/EmployeeManagementWebApplicationauth](https://github.com/jigar-prajapati1/EmployeeManagementWebApplicationauth)