Microservices with ASP.NET CORE Development Guide

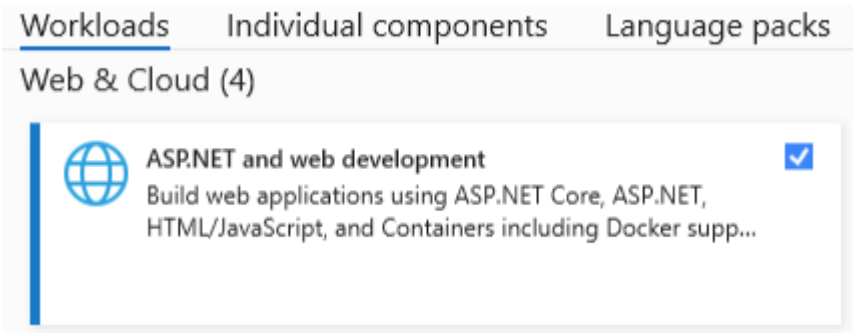
I) Environment and Tools

A) Visual Studio 2022 Community:

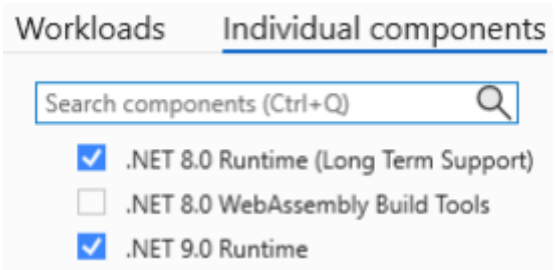
1. Download Visual Studio 2022 Community from: https://need4code.com/DotNet/Home/Index?path=.NET%5C00_Files%5CVisual%20Studio%20Community

2. After running the Visual Studio Installer:

   a) Select only **ASP.NET Web Development** under **Workloads**.



   b) Do not uncheck any component, only select **.NET 8 Runtime (Long Term Support)** under **Individual components**. **.NET 9 Runtime** should have already been selected.



   c) English is recommended for **Language packs**.

d) **Visual Studio IDE** installation is recommended on your SSD drive if you have any. **Download cache** and **Shared components, tools and SDKs** can either be installed on your SSD drive or on your other hard drive if you have any.

B) Rider for MAC:

https://www.jetbrains.com/rider

C) SQLite Database:

https://www.sqlite.org

II) Solution Setup

1. Create the ASP.NET Core Web API project.

2. Give the Project name Users.API. You may change the solution folder in Location. Give the Solution name your project name. Place solution and project in the same directory should not be checked.

3. Select ".NET 8.0" as the "Framework", choose "None" for "Authentication type", check "Configure for HTTPS", do not check "Enable container support", check “Enable OpenAPI support”, do not check "Do not use top-levet statements", check “Use controllers” and do not check "Enlist in .NET Aspire orchestration".

Note: When you run your application in Rider on Mac and if you get the exception below:

Failed to read NuGet.Config due to unauthorized access. Path: '/Users/YourUserFolder/.nuget/NuGet/NuGet.Config'

you should run the following commands in the terminal of Rider:

sudo chown $USER ~/.nuget/NuGet/NuGet.Config

chmod 644 ~/.nuget/NuGet/NuGet.Config

III) Projects Setup for CRUD Operations

Note: A solution is a set of one or more projects.

1. Right-click the solution in Solution Explorer, then Add -> New Project to create a project named CORE as a Class Library with .NET 8.0. You can delete the Class1.cs file after.

2. Right-click the solution in Solution Explorer, then Add -> New Project to create a project named Users.APP as a Class Library with .NET 8.0. You can delete the Class1.cs file after.

3. Set Nullable to Disable for CORE and Users.APP Projects (via project properties or XML).

4. Right-click Users.APP Project then click Add -> Project Reference then select the CORE Project to use the classes of the CORE Project in the Users.APP Project.

5. Right-click Users.API Project then click Add -> Project Reference then select the Users.APP Project to use both the classes of the CORE Project and Users.APP Project in the Users.API Project.

6. If Users.API project name doesn't appear bold in Solution Explorer, right-click Users.API Project then click Set as Startup Project.

7. Right-click CORE Project then click Manage NuGet Packages then in Browse tab search for Microsoft.EntityFrameworkCore latest version starting with 8 and install.

8. To use SQLite Database, right-click Users.APP Project then click Manage NuGet Packages then in Browse tab search for System.Data.SQLite.Core latest version and install then search for Microsoft.EntityFrameworkCore.Sqlite latest version starting with 8 and install.

9. To use Mediator, right-click Users.APP Project then click Manage NuGet Packages then in browse tab search for MediatR and install the latest version.

10. For using Code-First Approach (create entity classes and DbContext class first, database later), right-click Users.API Project then click Manage NuGet Packages then in Browse tab search for Microsoft.EntityFrameworkCore.Tools latest version starting with 8 and install.

A) CORE Project for CRUD Operations

1. Right-click CORE Project then click Add -> New Folder to create a folder named Domain.

2. Right-click Domain folder then click Add -> Class to create a class file named Record.cs as below: Don't forget to change the **internal** to **public** class visibility modifier for all classes and interfaces!

```
namespace CORE.Domain
{
    // Base abstract class for all entity, request and response classes.
    public abstract class Record
    {
        // Property for primary key of entities, unique identifier of
        // requests and responses.
        public int Id { get; set; }

        // Constructor with id parameter to set to Id property value
        // from a derived class constructor with id parameter.
        protected Record(int id)
        {
            Id = id;
        }

        // Default constructor to set the Id property default value 0.
        protected Record()
        {
        }
    }
}
```

3. Right-click CORE Project then click Add -> New Folder to create a folder named Models for base Data Transfer Object (DTO) classes.

4. Right-Click Models folder then create a class file called CommandResponse.cs as below:

```
using CORE.Domain;

namespace CORE.Models
{
    // Concrete class for returning create, update and delete
    // operation results.
    public class CommandResponse : Record
    {
        // Readonly property to return the success status as true or false.
        // Since no setter, the property value can only be set in the below
        // line or in the constructor.
        public bool IsSuccessful { get; }
```

```
        // Readonly property to return additional information such as a
        // success message or an error message containing details.
        public string Message { get; }

        // Constructor with parameters to set the IsSuccessful and Message
        // property values with id default parameter value passed to the
        // base abstract Record class constructor with id parameter.
        // If no value is passed for the id parameter, the default value 0
        // will be used.
        public CommandResponse(bool isSuccessful, string message, int id = 0)
            : base(id)
        {
            IsSuccessful = isSuccessful;
            Message = message;
        }
    }
}
```

5. Right-click CORE Project then click Add -> New Folder to create a folder called Services for base business logic classes.

6. Right-Click Services folder then create a class file called Service.cs as below:

```
using CORE.Models;
using System.Globalization;

namespace CORE.Services
{
    // Abstract base class for culture configuration and returning
    // successful or failure operation result CommandResponse objects.
    public abstract class Service
    {
        // Field to store and retrieve the backing culture value
        // using Encapsulation with the Culture property.
        private string _culture;

        // Property to retrieve the culture value and set the culture value
        // with assigning the service's current thread's culture information
        // to "en-US" for United States English or "tr-TR" for Turkish to
        // ensure consistent formatting and localization.
        protected string Culture
        {
            get
            {
                return _culture;
            }
            set
            {
                _culture = value;
                Thread.CurrentThread.CurrentCulture = new CultureInfo(_culture);
                Thread.CurrentThread.CurrentUICulture =
                    new CultureInfo(_culture);
            }
        }
```

```
        // Default constructor to set the default culture to "en-US" for
        // United States English. "tr-TR" can be assigned to the Culture property
        // in the derived class constructor or methods for Turkish culture.
        protected Service()
        {
            Culture = "en-US";
        }

        // Behavior (method, function) to return a successful CommandResponse
        // object for the operation with the entity's ID and an optional message
        // default parameter.
        // If no message is provided, a default success message is used
        // by using the Null Coalescing Operator meaning that if message is null,
        // return "Operation successful." message, otherwise return
        // the provided message.
        protected CommandResponse Success(int id, string message = default)
            => new CommandResponse(true, message ?? "Operation successful.", id);

        // Behavior to return a failure CommandResponse object for the operation
        // with an optional message default parameter. If no message is provided,
        // a default error message is used by using the Null Coalescing Operator
        // meaning that if message is null, return "Operation failed!" message,
        // otherwise return the provided message.
        protected CommandResponse Error(string message = default)
            => new CommandResponse(false, message ?? "Operation failed!");
    }
}
```

7. Right-click Services folder then create DbService class file as below:

```
using CORE.Domain;
using Microsoft.EntityFrameworkCore;
using System.Linq.Expressions;

namespace CORE.Services
{
    // Abstract base class to perform CRUD (Create, Read, Update and Delete)
    // database operations for a specific entity type inherited from the
    // Record class and has a parameterless constructor.
    // Inherits from the Service class to manage culture and return
    // successful or failure operation result CommandResponse objects
    // after operations.
    // Implements the IDisposable interface to release unmanaged resources.
    public abstract class DbService<TEntity> : Service, IDisposable
        where TEntity : Record, new()
    {
        // Field for Dependency Injection for the DbContext instance to
        // perform database operations in the below methods.
        private readonly DbContext _db;

        // Constructor with an injected DbContext instance parameter
        // for performing database operations in the below methods by
        // using the injected instance assigned _db field.
        protected DbService(DbContext db)
        {
            _db = db;
        }
```

```
// Method to return the entity query such as "select * from table"
// where table is the related entity DbSet.
// Defined as virtual to allow derived classes to override the
// query if needed such as for ordering, filtering, including
// the related entities to the query, etc.
// AsNoTracking is used for not tracking the returned entities to
// improve performance. If not written, the default
// behavior is tracking.
protected virtual IQueryable<TEntity> DbQuery()
    => _db.Set<TEntity>().AsNoTracking();

// Method to return a single tracked (AsTracking) entity for update
// or delete operations by its ID using the base or derived overridden
// DbQuery method. If the entity is not found, returns null since
// OrDefault version of the Single method is used.
protected async Task<TEntity> DbSingleAsync(int id,
    CancellationToken cancellationToken = default)
{
    return await DbQuery().AsTracking()
        .SingleOrDefaultAsync(entity => entity.Id == id,
            cancellationToken);
}

// Method to return a single tracked (AsTracking) entity
// according to a lambda expression filter parameter (predicate)
// which may contain one or more conditions using the base or
// derived overridden DbQuery method. If the entity is not found,
// returns null since OrDefault version of the Single method is used.
protected async Task<TEntity> DbSingleAsync(
    Expression<Func<TEntity, bool>> predicate,
    CancellationToken cancellationToken = default)
    => await DbQuery().AsTracking().SingleOrDefaultAsync(predicate,
        cancellationToken);

// Method to persist changes to the database and return the number
// of effected rows of the table.
// Defined as virtual to allow derived classes to override the behavior
// if needed such as for extra logging, error handling, etc.
protected virtual async Task<int> DbSaveAsync(
    CancellationToken cancellationToken = default)
    => await _db.SaveChangesAsync(cancellationToken);

// Method to insert a new entity to the entity DbSet and persist changes
// to the database if the save default parameter value is true.
// If the save parameter value is false, the changes will not be
// persisted to the database, which can be used for multiple insert
// operations. Then DbSave method can be invoked to persist all changes
// to the database once (Unit of Work) to increase performance.
protected async Task DbAddAsync(TEntity entity,
    CancellationToken cancellationToken,
    bool save = true)
{
    _db.Set<TEntity>().Add(entity);
    if (save)
        await DbSaveAsync(cancellationToken);
}

// Method to update an existing entity retrieved by DbSingle method
// and persist changes to the database if the save default parameter
// value is true. If the save parameter value is false, the changes will
// not be persisted to the database, which can be used for multiple
// update operations. Then DbSave method can be invoked to persist all
```

```
        // changes to the database once (Unit of Work) to increase performance.
        protected async Task DbUpdateAsync(TEntity entity,
            CancellationToken cancellationToken,
            bool save = true)
        {
            _db.Set<TEntity>().Update(entity);
            if (save)
                await DbSaveAsync(cancellationToken);
        }

        // Method to delete an existing entity retrieved by DbSingle method
        // and persist changes to the database if the save default parameter
        // value is true. If the save parameter value is false, the changes will
        // not be persisted to the database, which can be used for multiple
        // delete operations. Then DbSave method can be invoked to persist all
        // changes to the database once (Unit of Work) to increase performance.
        protected async Task DbRemoveAsync(TEntity entity,
            CancellationToken cancellationToken,
            bool save = true)
        {
            _db.Set<TEntity>().Remove(entity);
            if (save)
                await DbSaveAsync(cancellationToken);
        }

        // Method to delete the related navigation entities of an entity.
        // The changes will not be persisted to the database, therefore
        // DbAddAsync, DbUpdateAsync, DbRemoveAsync or DbSaveAsync methods
        // should be invoked after to persist all changes to the database.
        protected void DbRemove<TNavigationEntity>(List<TNavigationEntity>
            navigationEntities) where TNavigationEntity : Record, new()
        {
            _db.Set<TNavigationEntity>().RemoveRange(navigationEntities);
        }

        // Method to dispose the DbContext instance and suppress finalization
        // (don't call the finalizer) of the instance of this class to
        // improve performance and avoid memory leaks.
        public void Dispose()
        {
            _db.Dispose();
            GC.SuppressFinalize(this);
        }
    }
}
```

B) Users.APP Project for CRUD Operations

1. Right-click Users.APP Project then click Add -> New Folder to create a folder called Domain for entity classes and DbContext class (Entity Framework base database operations and configurations class).

2. Right-click Domain folder then click Add -> Class to create a class file called Role.cs as an entity for the Roles database table as below:

```
using CORE.Domain;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace Users.APP.Domain
{
    // Role is a Record.
    public class Role : Record
    {
        // Reference Type: string, array, class and interface
        // are reference types (can be null).
        // Required and StringLength are called attributes
        // (or data annotations when used in an entity or a request).
        // Name can't be null and must contain maximum 25 characters.
        [Required, StringLength(25)]
        public string Name { get; set; } // = "Admin";
                                         // Initial assignments may also
                                         // be done to the properties.
                                         // No need to assign "Admin" here.

        // Reference Type and Navigation Property:
        // Navigation properties are used to navigate from one entity
        // to another related entity to get or set related data.
        // Will be an empty list of UserRole if no assignment
        // or no include to a query.
        // For roles-users many to many relationship.
        public List<UserRole> UserRoles { get; set; }
            = new(); // = new List<UserRole>(); can also be written

        // This property helps to easily manage the UserRoles relational entities
        // by Role entity's User Id values.
        // NotMapped attribute means no column in the Roles table will be created
        // for this property.
        //[NotMapped]
        //public List<int> UserIds
        //{
        //    // Returns the User Id values of the Role entity.
        //    get => UserRoles.Select(userRoleEntity
        //        => userRoleEntity.UserId).ToList();
        //
        //    // Sets the UserRoles relational entities of the Role entity
        //    // by the assigned User Id values.
        //    set => UserRoles = value.Select(userIdValue => new UserRole
        //    {
        //        UserId = userIdValue
        //    }).ToList();
        //}
        // Since we won't update the relational user data (UserRoles)
        // through Role entity, we don't need the UserIds property here.
    }
}
```

3. Right-click Domain folder then click Add -> Class to create a class file called UserRole.cs as an entity for the UserRoles many to many relation database table as below:

```
using CORE.Domain;

namespace Users.APP.Domain
{
    // UserRole is a Record.
    // For users-roles many to many relationship.
    public class UserRole : Record
    {
        // Value Type: Types other than string, array, class and interface
        // are value types (can't be null).
        // UserId can't be null and will have the default value 0
        // if no value is assigned.
        public int UserId { get; set; } // foreign key to the User entity

        // Reference Type and Navigation Property: string, array, class
        // and interface are reference types (can be null).
        // Navigation properties are used to navigate from one entity
        // to another related entity to get or set related data.
        // Will be null if no assignment or no include to a query.
        public User User { get; set; }

        // Value Type:
        // RoleId can't be null and will have the default value 0
        // if no value is assigned.
        public int RoleId { get; set; } // foreign key to the Role entity

        // Reference Type and Navigation Property:
        // Will be null if no assignment or no include to a query.
        public Role Role { get; set; }
    }
}
```

4. Right-click Domain folder then click Add -> Class to create a class file called User.cs as an entity for the Users database table as below:

```
using CORE.Domain;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace Users.APP.Domain
{
    // User is a Record.
    public class User : Record
    {
        // Reference Type: string, array, class and interface
        // are reference types (can be null).
        // UserName can't be null and must contain maximum 30 characters.
        [Required, StringLength(30)]
        public string UserName { get; set; }
```

```
// Reference Type:
// Password can't be null and must contain maximum 15 characters.
[Required, StringLength(15)]
public string Password { get; set; }

// Reference Type:
// FirstName can be null and must contain maximum 50 characters.
// Will be null if no assignment.
[StringLength(50)]
public string FirstName { get; set; }


// Reference Type:
// LastName can be null and must contain maximum 50 characters.
// Will be null if no assignment.
[StringLength(50)]
public string LastName { get; set; }

// Value Type: Types other than string, array, class and interface
// are value types (can't be null).
// Gender can't be null and will have the first value of the Genders
// enum if no value is assigned.
public Genders Gender { get; set; }

// Value Type: ? after the type is used to make the value type act
// like reference type.
// BirthDate can be null since ? is used after the type and
// will be null if no value is assigned.
public DateTime? BirthDate { get; set; }

// Value Type:
// RegistrationDate can't be null and will have the default value
// 01/01/0001 00:00:00 (DateTime.MinValue) if no value is assigned.
public DateTime RegistrationDate { get; set; }

// Value Type:
// Score can't be null and will have the default value 0
// if no value is assigned.
public double Score { get; set; }

// Value Type:
// IsOnline can't be null and will have the default value false
// if no value is assigned.
public bool IsOnline { get; set; }

// Value Type:
// StatusId can't be null and will have the default value 0
// if no value is assigned.
// For users-status one to many relationship.
public int StatusId { get; set; } // foreign key to the Status entity

// Reference Type and Navigation Property:
// Navigation properties are used to navigate from one entity
// to another related entity to get or set related data.
// Will be null if no assignment or no include to a query.
// For users-status one to many relationship.
public Status Status { get; set; }

// Reference Type and Navigation Property:
// Will be an empty list of UserRole if no assignment
// or no include to a query.
// For users-roles many to many relationship.
public List<UserRole> UserRoles { get; set; }
    = new(); // = new List<UserRole>(); can also be written
```

```
        // This property helps to easily manage the UserRoles relational entities
        // by User entity's Role Id values.
        // NotMapped attribute means no column in the Users table will be created
        // for this property.
        [NotMapped]
        public List<int> RoleIds
        {
            // Returns the Role Id values of the User entity.
            get => UserRoles.Select(userRoleEntity
                => userRoleEntity.RoleId).ToList();

            // Sets the UserRoles relational entities of the User entity
            // by the assigned Role Id values.
            set => UserRoles = value.Select(roleIdValue => new UserRole
            {
                RoleId = roleIdValue
            }).ToList();
        }



        // The following properties will be used for JWT (JSON Web Token)
        // based authentication.
        public string RefreshToken { get; set; }

        public DateTime? RefreshTokenExpiration { get; set; }
    }
}
```

5. Right-click Domain folder then click Add -> Class to create a class file called Status.cs as an entity for the Statuses database table as below:

```
using CORE.Domain;
using System.ComponentModel.DataAnnotations;

namespace Users.APP.Domain
{
    // Status is a Record.
    public class Status : Record
    {
        // Reference Type: string, array, class and interface
        // are reference types (can be null).
        // Title can't be null and must contain maximum 5 characters.
        [Required, StringLength(5)]
        public string Title { get; set; }

        // Reference Type and Navigation Property:
        // Navigation properties are used to navigate from one entity
        // to another related entity to get or set related data.
        // Will be an empty list of User if no assignment
        // or no include to a query.
        // For status-users one to many relationship.
        public List<User> Users { get; set; }
            = new(); // = new List<User>(); can also be written
    }
}
```

6. Right-click Domain folder then click Add -> Class to create an enum file named Genders.cs as below:

```
namespace Users.APP.Domain
{
    // Enums are used for predefined values and help not to remember or
    // check each element's value when being used.
    // Getting the value of an enum element:
    // int womanValue = (int)Genders.Woman;
    // Getting the text of an enum element:
    // string manText = Genders.Man.ToString();
    public enum Genders
    {
        Woman, // if no assignment, will start from 0
        Man // will automatically get the next value 1 and other elements
            // will continue to get the next values after 1
    }
}
```

7. Right-click Domain folder then click Add -> Class to create a class file called UsersDb for Entity Framework database configurations and operations as below:

```
using Microsoft.EntityFrameworkCore;

namespace Users.APP.Domain
{
    // Inherits from the Entity Framework's DbContext class for
    // database configurations and operations.
    public class UsersDb : DbContext
    {
        // Represents the Users table in the database.
        public DbSet<User> Users { get; set; }

        // Represents the Roles table in the database.
        public DbSet<Role> Roles { get; set; }

        // Represents the UserRoles table in the database.
        public DbSet<UserRole> UserRoles { get; set; }

        // Represents the Statuses table in the database.
        public DbSet<Status> Statuses { get; set; }

        // Constructor that takes options of type DbContextOptions
        // parameter and passes it to the base DbContext class constructor.
        // The options parameter is used to configure the database
        // connection with such as the connection string.
        public UsersDb(DbContextOptions options) : base(options)
        {
        }
    }
}
```

Database Creation:

8. In the Users.API Project, open appsettings.json, which will contain the application configuration such as the connection string, static information, etc., then add the ConnectionStrings section as below:

```
"ConnectionStrings": {
   "UsersDb": "data source=UsersDB.db"
},
```

UsersDb is the connection string name, UsersDB.db will be the SQLite database file that will be created.

9. In the Users.API Project, open Program.cs and add below in the IoC (Inversion of Control) Container:

```
// Register the application's DbContext dependency injection
// by sending UsersDb instances to the injected service class
// constructors' DbContext or UsersDb parameter, using SQLite
// with the connection string from appsettings.json.
builder.Services.AddDbContext<DbContext, UsersDb>(
    options => options.UseSqlite(
        builder.Configuration.GetConnectionString(nameof(UsersDb))));
          // nameof(UsersDb) = "UsersDb" which is the class name.
```

10. Create your UsersDB.db database using migrations:
    - Open Package Manager Console from Visual Studio menu -> Tools -> NuGet Package Manager -> Package Manager Console
    - Set Users.APP as Default Project in Package Manager Console
    - Run:
        add-migration v1
        update-database
    - For Rider, you can use the UI as described in JetBrains documentation, or from the terminal
      Install:
        dotnet tool install -g dotnet-ef
      Run:
        dotnet ef migrations add v1
        dotnet ef database update -p Users.APP -s Users.API
    - You can see the created UsersDB.db database file in Users.API Project.
    - Optionally in Visual Studio, you can install the SQLite and SQL Server Compact Toolbox extension from Visual Studio menu -> Extensions -> Manage Extensions to connect to the created UsersDB.db SQLite database.

11. In the Users.APP Project under Domain folder, optionally a UsersDbFactory class can be created to be used for Scaffolding (creation of API Controllers automatically using templates) and seeding the database tables with initial data as below:

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Design;
using System.Globalization;

namespace Users.APP.Domain
{
    // Provides a factory for creating UsersDb instances at design time.
    // This is used by Entity Framework (EF) tools such as
    // migrations to construct the database context when the
    // application is not running.
    // This class may need to be created if there are any exceptions
    // during Scaffolding.
    public class UsersDbFactory : IDesignTimeDbContextFactory<UsersDb>
    {
        // The connection string in the configuration file
        // (appsettings.json).
        const string CONNECTIONSTRING = "data source=UsersDB.db";


        // Creates a new instance of the UsersDb type using the connection
        // string. This method is called by EF tooling at design time.
        public UsersDb CreateDbContext(string[] args)
        {
            var optionsBuilder = new DbContextOptionsBuilder<UsersDb>();
            optionsBuilder.UseSqlite(CONNECTIONSTRING);
            return new UsersDb(optionsBuilder.Options);
        }



        // Seeds the database tables with initial data (Optional).
        public void SeedDb()
        {
            // Create a new instance of the UsersDb type:
            var db = CreateDbContext(null);

            // Delete existing data:
            var userRoles = db.UserRoles.ToList();
            db.UserRoles.RemoveRange(userRoles);
            var roles = db.Roles.ToList();
            db.Roles.RemoveRange(roles);
            var users = db.Users.ToList();
            db.Users.RemoveRange(users);
            var statuses = db.Statuses.ToList();
            db.Statuses.RemoveRange(statuses);

            // Reset identity columns (for SQLite):
            // IDs will start from 1
            db.Database.ExecuteSqlRaw("UPDATE SQLITE_SEQUENCE " +
                "SET SEQ=0 WHERE NAME='UserRoles'");
            db.Database.ExecuteSqlRaw("UPDATE SQLITE_SEQUENCE " +
                "SET SEQ=0 WHERE NAME='Roles'");
            db.Database.ExecuteSqlRaw("UPDATE SQLITE_SEQUENCE " +
                "SET SEQ=0 WHERE NAME='Users'");
```

```
            db.Database.ExecuteSqlRaw("UPDATE SQLITE_SEQUENCE " +
                "SET SEQ=0 WHERE NAME='Statuses'");

            // Insert new data:
            // Roles:
            var role = new Role() { Name = "Admin" };
            db.Roles.Add(role);
            // Role may also be written instead of Role()
            role = new Role { Name = "User" };
            db.Roles.Add(role);
            db.SaveChanges(); // commit changes to the database
            // Statuses with Users:
            db.Statuses.Add(new Status
            {
                Title = "Active",
                Users = new List<User>
                {
                    new User
                    {
                        UserName = "admin",
                        Password = "admin",
                        RegistrationDate = new DateTime(
                            2026, 7, 16, 20, 57, 45),
                        UserRoles = new List<UserRole>
                        {
                            new UserRole { RoleId = db.Roles.Single(
                                r => r.Name == "Admin").Id }
                        }
                    },
                    new User
                    {
                        UserName = "user",
                        Password = "user",
                        RegistrationDate = DateTime.Parse(
                            "07/16/2026 20:58:13", new CultureInfo("en-US")),
                        UserRoles = new List<UserRole>
                        {
                            new UserRole { RoleId = db.Roles.Single(
                                r => r.Name == "User").Id }
                        }
                    }
                }
            });
            db.Statuses.Add(new Status { Title = "Inactive" });
            db.SaveChanges();
        }
    }
}
```

12. In the Users.API Project, right-click Controllers folder then Add -> Controller -> Common -> API -> API Controller - Empty to create the UsersDbController for seeding intial data to the database tables as below:

```
using Microsoft.AspNetCore.Mvc;
using Users.APP.Domain;
```

```
namespace Users.API.Controllers
{
    // Database Controller for seeding the database with initial data
    // through the Seed action.
    [Route("api/[controller]")] // route for the controller: api/UsersDb
    [ApiController] // attribute indicates that this is an API controller
    public class UsersDbController : ControllerBase
    {
        [HttpGet, Route("~/api/SeedDb")] // route for the get action
                                         // changed with the Route
                                         // attribute: api/SeedDb
        public IActionResult Seed()
        {
            // Initialize the UsersDbFactory instance.
            var dbFactory = new UsersDbFactory();

            // Seed the database using the SeedDb method.
            dbFactory.SeedDb();

            // Return HTTP OK status result (200) with a success message.
            return Ok("Database seed successful.");
        }
    }
}
```

13. Run the solution by hitting F5 or play icon with https on the top toolbar. If a warning dialog appears about SSL, click Yes for all dialogs. If you have any problems running your solution, change the play icon with https to http then run. Finally, enter /SeedDb after the server name (localhost) and port number in the address bar of the browser or use Swagger to execute the SeedDb get action. For example “https://localhost:7124/api/seeddb” to seed the database tables with initial data.

CRUD Operations Business Logic Implementation:

14. Right-click the Users.APP Project then click Add -> New Folder to create a folder named Features.

15. Right-click the Features folder then click Add -> New Folder to create a folder called Roles.

16. Right-click the Roles folder then click Add -> Class to create a class file named RoleQuery.cs that will include three classes RoleQueryRequest, RoleQueryResponse and RoleQueryHandler as below (Optionally for each class, a seperate class file may be created, therefore there may be RoleQueryRequest.cs, RoleQueryResponse.cs and RoleQueryHandler.cs class files in the Roles folder):

```
using CORE.Domain;
using CORE.Services;
using MediatR;
using Microsoft.EntityFrameworkCore;
using Users.APP.Domain;

namespace Users.APP.Features.Roles
{
    // Request and response model classes are also called
    // Data Transfer Object (DTO) classes.
    // Request properties are created according to the data that
    // will be retrieved from APIs.
    // This class represents a MediatR request for querying roles.
    // Inherits from Record and specifies the expected response
    // type as IQueryable of RoleQueryResponse to be presented
    // as a list or a single item in APIs.
    // This class inherits from Record to include the common
    // identifier property (Id).
    public class RoleQueryRequest : Record,
        IRequest<IQueryable<RoleQueryResponse>>
    {
        // Optionally properties may be defined here for
        // filtering, ordering or pagination.
    }

    // Represents the response model (DTO: Data Trasfer Object)
    // for querying Role entities.
    // The properties of a model are generally copied from the
    // related entity properties which are not navigation properties,
    // or which have the columns in the related database table.
    // Inherits from Record to include the common identifier
    // property (Id).
    public class RoleQueryResponse : Record
    {
        // Copy all the non navigation properties from Role entity.
        public string Name { get; set; }
    }

    // Inherits the DbService class for Role entity type,
    // therefore Role entity database operations of the DbService
    // class can be used in this class.
    // This class also implements the Mediator interface for types
    // RoleQueryRequest and IQueryable of RoleQueryResponse, therefore
    // the Handler method can be implemented for these types
    // for the role query business operation.
    public class RoleQueryHandler : DbService<Role>,
        IRequestHandler<RoleQueryRequest, IQueryable<RoleQueryResponse>>
    {
        // Constructor that passes the injected DbContext instance
        // to the DbService base class constructor.
        public RoleQueryHandler(DbContext db) : base(db)
        {
        }
```

```
        // Gets the Role entity query then projects the Role
        // entity query to the RoleQueryResponse query and returns it.
        // Example query: "select Name from Roles".
        // where Name is the RoleQueryResponse property.
        // The returned query can be executed by invoking ToListAsync,
        // SingleOrDefaultAsync, FirstOrDefaultAsync, AnyAsync, etc. LINQ
        // (Language Integrated Query) methods and then the returned
        // result can be used.
        public Task<IQueryable<RoleQueryResponse>> Handle(
             RoleQueryRequest request, CancellationToken cancellationToken)
        {
            // Way 1:
            //IQueryable<RoleQueryResponse> query =
            // Way 2: var can be used instead of any type if
            // there is an assignment.
            var query =
                DbQuery()// "select * from Roles" entity query.
                .Select(roleEntity => new RoleQueryResponse()
                // () after the class name may not be written.
                {
                    Id = roleEntity.Id,
                    Name = roleEntity.Name
                }); // Map each Role entity property
                    // to RoleQueryResponse property.

            // Return the task to be awaited by the caller to get the
            // IQueryable of type RoleQueryResponse query.
            return Task.FromResult(query);
        }
    }
}
```

17. Right-click the Roles folder then click Add -> Class to create a class file named RoleAdd.cs that will include two classes RoleAddRequest and RoleAddHandler as below (Optionally for each class, a seperate class file may be created, therefore there may be RoleAddRequest.cs and RoleAddHandler.cs class files in the Roles folder):

```
using CORE.Domain;
using CORE.Models;
using CORE.Services;
using MediatR;
using Microsoft.EntityFrameworkCore;
using System.ComponentModel.DataAnnotations;
using Users.APP.Domain;

namespace Users.APP.Features.Roles
{
    // Request properties are created according to the data that
    // will be retrieved from APIs.
    public class RoleAddRequest : Record, IRequest<CommandResponse>
    {
        // Copy all the non navigation properties from Role entity.
        // Name can't be null and can be maximum 25 characters.
        [Required, StringLength(25)]
        public string Name { get; set; }
```

```
        // Modify (1 to many) or add (many to many) the relationship
        // properties. Since we won't update the relational user
        // data (UserRoles) through the request, we don't need the
        // UserIds property here.
        // Required may not be defined if a role may not have a user.
        // For roles-users many to many relationship.
        //[Required]
        //[DisplayName("Users")]
        //public List<int> UserIds { get; set; } = new();
    }

    // RoleAddHandler class injects the DbContext instance for
    // querying and inserting a new role to the Roles database table.
    // The Handle method implemented from the IRequestHandler
    // interface checks whether the role with the same trimmed and
    // case sensitive name exists in the Roles database table first.
    // If it doesn't exist, inserts the Role entity mapped from the
    // RoleAddRequest instance to the Roles database table.
    public class RoleAddHandler : DbService<Role>,
        IRequestHandler<RoleAddRequest, CommandResponse>
    {
        // Constructor that passes the injected DbContext instance
        // to the DbService base class constructor.
        public RoleAddHandler(DbContext db) : base(db)
        {
        }

        // Inserts a new role to the Roles database table if a
        // role with the same name doesn't exist.
        public async Task<CommandResponse> Handle(RoleAddRequest request,
            CancellationToken cancellationToken)
        {
            if (await DbQuery()
                .AnyAsync(roleEntity =>
                    roleEntity.Name == request.Name.Trim(),
                        cancellationToken))
                return Error(request.Name + " role exists!");
            // Trim method removes white space characters in the
            // beginning and at the end.

            // Create a new Role entity instance from the RoleAddRequest
            // instance then insert the entity in the Roles database
            // table (third save default parameter of the DbAddAsync
            // method is true therefore changes of the DbSets are
            // commited to the related database tables).
            var roleEntity = new Role
            {
                Name = request.Name.Trim()
            };
            await DbAddAsync(roleEntity, cancellationToken);

            // Return a successful command response with the new
            // role's ID.
            // Entity's Id value will be updated by the database
            // after the insert operation.
            return Success(roleEntity.Id, roleEntity.Name +
                " role created successfully.");
        }
    }
}
```

18. Right-click the Roles folder then click Add -> Class to create a class file named RoleUpdate.cs that will include two classes RoleUpdateRequest and RoleUpdateHandler as below (Optionally for each class, a seperate class file may be created, therefore there may be RoleUpdateRequest.cs and RoleUpdateHandler.cs class files in the Roles folder):

```
using CORE.Domain;
using CORE.Models;
using CORE.Services;
using MediatR;
using Microsoft.EntityFrameworkCore;
using System.ComponentModel.DataAnnotations;
using Users.APP.Domain;

namespace Users.APP.Features.Roles
{
    // Request properties are created according to the data that
    // will be retrieved from APIs.
    public class RoleUpdateRequest : Record, IRequest<CommandResponse>
    {
        // Copy all the non navigation properties from Role entity.
        // Name can't be null and can be maximum 25 characters.
        [Required, StringLength(25)]
        public string Name { get; set; }

        // Modify (1 to many) or add (many to many) the relationship
        // properties. Since we won't update the relational user
        // data (UserRoles) through the request, we don't need the
        // UserIds property here.
        // Required may not be defined if a role may not have a user.
        // For roles-users many to many relationship.
        //[Required]
        //[DisplayName("Users")]
        //public List<int> UserIds { get; set; } = new();
    }

    // RoleUpdateHandler class injects the DbContext instance for
    // querying and updating an existing role in the Roles database
    // table.
    // The Handle method implemented from the IRequestHandler
    // interface checks whether the role with the same trimmed and
    // case sensitive name other than the role request record
    // exists in the Roles database table first.
    // If it doesn't exist, updates the Role entity mapped from the
    // RoleUpdateRequest instance in the Roles database table.
    public class RoleUpdateHandler : DbService<Role>,
        IRequestHandler<RoleUpdateRequest, CommandResponse>
    {
        // Constructor that passes the injected DbContext instance
        // to the DbService base class constructor.
        public RoleUpdateHandler(DbContext db) : base(db)
        {
        }

        // Updates an existing role in the Roles database table if a
        // role with the same name other than the role request record
        // doesn't exist.
```

```
        public async Task<CommandResponse> Handle(RoleUpdateRequest request,
            CancellationToken cancellationToken)
        {
            if (await DbQuery()
                .AnyAsync(roleEntity => roleEntity.Id != request.Id &&
                    roleEntity.Name == request.Name.Trim(),
                        cancellationToken))
                return Error($"{request.Name} role exists!");
            // $ with strings can be used for concatenation with { and }.

            // Get the Role entity by ID from the Roles database table
            // and check whether it is null or not.
            var roleEntity = await DbSingleAsync(request.Id, cancellationToken);
            if (roleEntity is null)
                return Error("Role not found!");

            // Update Role entity properties from RoleUpdateRequest properties.
            // Then update the entity in the Roles database table (third save
            // default parameter of the DbUpdateAsync method is true
            // therefore changes of the DbSets are commited to the related
            // database tables).
            roleEntity.Name = request.Name.Trim();
            await DbUpdateAsync(roleEntity, cancellationToken);

            return Success(roleEntity.Id,
                $"{roleEntity.Name} role updated successfully.");
        }
    }
}
```

19. Right-click the Roles folder then click Add -> Class to create a class file named RoleRemove.cs that will include two classes RoleRemoveRequest and RoleRemoveHandler as below (Optionally for each class, a seperate class file may be created, therefore there may be RoleRemoveRequest.cs and RoleRemoveHandler.cs class files in the Roles folder):

```
using CORE.Domain;
using CORE.Models;
using CORE.Services;
using MediatR;
using Microsoft.EntityFrameworkCore;
using Users.APP.Domain;

namespace Users.APP.Features.Roles
{
    // Delete operation will be performed by the Id property inherited
    // from the Record base class. Therefore, no additional properties
    // are needed to be defined.
    public class RoleRemoveRequest : Record, IRequest<CommandResponse>
    {
    }

    // RoleRemoveHandler class injects the DbContext instance for
    // querying and deleting an existing role in the Roles database
    // table.
    // The Handle method implemented from the IRequestHandler
    // interface gets the Role entity by ID first.
    // If found, deletes the Role entity from the Roles database table.
```

```
    public class RoleRemoveHandler : DbService<Role>,
        IRequestHandler<RoleRemoveRequest, CommandResponse>
    {
        // Constructor that passes the injected DbContext instance
        // to the DbService base class constructor.
        public RoleRemoveHandler(DbContext db) : base(db)
        {
        }

        // Deletes an existing role from the Roles database table if
        // it is found by request's ID.
        public async Task<CommandResponse> Handle(RoleRemoveRequest request,
            CancellationToken cancellationToken)
        {
            // Get the Role entity by ID from the Roles database table
            // and check whether it is null or not.
            var roleEntity = await DbSingleAsync(request.Id, cancellationToken);
            if (roleEntity is null)
                return Error("Role not found!");

            // Delete the Role entity from the Roles database table
            // (third save default parameter of the DbRemoveAsync method is
            // true therefore changes of the DbSets are commited to
            // the related database tables).
            // Since the delete rule of the Roles and UserRoles relation
            // in the database is Cascade, the relational UserRole entities
            // will be deleted automatically (not recommended).
            await DbRemoveAsync(roleEntity, cancellationToken);

            return Success(roleEntity.Id,
                $"{roleEntity.Name} role deleted successfully.");
        }
    }
}
```

20. Right-click the Features folder then click Add -> New Folder to create a folder called Users.

21. Right-click the Users folder then click Add -> Class to create a class file named UserQuery.cs that will include three classes UserQueryRequest, UserQueryResponse and UserQueryHandler as below (Optionally for each class, a seperate class file may be created, therefore there may be UserQueryRequest.cs, UserQueryResponse.cs and UserQueryHandler.cs class files in the Users folder):

```
using CORE.Domain;
using CORE.Services;
using MediatR;
using Microsoft.EntityFrameworkCore;
using Users.APP.Domain;
using Users.APP.Features.Roles;
using Users.APP.Features.Statuses;

namespace Users.APP.Features.Users
{
    // Request properties are created according to the data that
    // will be retrieved from APIs.
```

```
public class UserQueryRequest : Record,
    IRequest<IQueryable<UserQueryResponse>>
{
}

// Response properties are created according to the data to be
// presented in API responses.
public class UserQueryResponse : Record
{
    // Copy all the non navigation properties from User entity.
    public string UserName { get; set; }

    public string Password { get; set; }

    public string FirstName { get; set; }

    public string LastName { get; set; }

    public Genders Gender { get; set; }

    public DateTime? BirthDate { get; set; }

    public DateTime RegistrationDate { get; set; }

    public double Score { get; set; }

    public bool IsOnline { get; set; }

    public int StatusId { get; set; }



    // Add the new properties ending with R declaring Response
    // for custom properties or formatted string value properties.
    public string IsOnlineR { get; set; }

    public string ScoreR { get; set; }

    public string RegistrationDateR { get; set; }

    public string BirthDateR { get; set; }

    public string GenderR { get; set; }

    // FirstName + " " + LastName concatenated value.
    public string FullNameR { get; set; }

    // Password value hidden with * characters.
    public string PasswordR { get; set; }



    // Way 1:
    // Property for the Title value of the
    // related Status entity.
    public string StatusTitleR { get; set; }

    // Way 2:
    // Property for the mapped StatusQueryResponse object
    // from the related Status entity.
    // Either Way 1, Way 2 or both can be used.
    public StatusQueryResponse StatusR { get; set; }



    // Way 1:
    // Property for the concatenated role names
    // of related Role entities.
    public string RoleNamesR { get; set; }
```

```
    // Way 2:
    // Property for the RoleQueryResponse objects projected
    // from the related Role entities.
    // Either Way 1, Way 2 or both can be used.
    public List<RoleQueryResponse> RolesR { get; set; }
}

// UserQueryHandler inherits the DbService class for User entity type,
// therefore User entity database operations of the DbService
// class can be used in this class.
// UserQueryHandler also implements the Mediator IRequestHandler
// interface for UserQueryRequest and IQueryable of UserQueryResponse
// types, therefore IRequestHandler interface's Handle method
// definition can be implemented for types UserQueryRequest and
// IQueryable of UserQueryResponse in this class, which will be used by
// the related controller's action.
public class UserQueryHandler : DbService<User>,
    IRequestHandler<UserQueryRequest,
        IQueryable<UserQueryResponse>>
{
    // Constructor that passes the injected DbContext instance
    // to the DbService base class constructor.
    public UserQueryHandler(DbContext db) : base(db)
    {
    }

    // Overrides the base virtual DbQuery method to include
    // Status, UserRole and Role entities data to the query
    // to be used in the Handle method and orders the query
    // by User entity Score property descending first, then
    // the ordered records are ordered descending by User
    // entity RegistrationDate property. Finally the ordered
    // records are ordered ascending by User entity UserName
    // property.
    // The overridden DbQuery method can be used in any method
    // below by invoking the DbQuery method.
    protected override IQueryable<User> DbQuery()
    {
        // Relationships:
        // User <-> Status.
        // User <-> UserRoles <-> Roles.
        return base.DbQuery()
            .Include(u => u.Status) // Include Status entities
                                   // from User entities.
            .Include(u => u.UserRoles) // Include UserRole entities
                                       // from User entities.
            .ThenInclude(ur => ur.Role) // Then include Role entities
                                        // from UserRole entities.
            .OrderByDescending(u => u.Score)
            .ThenByDescending(u => u.RegistrationDate)
            .ThenBy(u => u.UserName);
        // Only one OrderBy method must be called.
        // The other called ordering methods must be ThenBy.
        // u: User entity delegate, ur: UserRole entity delegate.
    }

    // Gets the updated query from the DbQuery method then projects
    // the User entities to user query responses and returns the
    // UserQueryResponse query.
    // The returned query can be executed by invoking ToListAsync,
    // SingleOrDefaultAsync, FirstOrDefaultAsync, AnyAsync, etc.
    // LINQ (Language Integrated Query) methods and then the
    // returned result can be used.
    public Task<IQueryable<UserQueryResponse>> Handle(
        UserQueryRequest request, CancellationToken cancellationToken)
    {
        return Task.FromResult(
            DbQuery().Select(u => new UserQueryResponse
        {
```

```
            // Map each User entity property to UserQueryResponse property.
            Id = u.Id,
            UserName = u.UserName,
            Password = u.Password,
            FirstName = u.FirstName,
            LastName = u.LastName,
            RegistrationDate = u.RegistrationDate,
            BirthDate = u.BirthDate,
            IsOnline = u.IsOnline,
            Gender = u.Gender,
            Score = u.Score,
            StatusId = u.StatusId,
            // Way 1: Map primitive type data for basic information.
            StatusTitleR = u.Status.Title,
            // Way 2: Map complex type object for more information.
            // Either Way 1, Way 2 or both can be used.
            StatusR = new StatusQueryResponse
            {
                // Map each Status entity property to
                // UserQueryResponse's StatusQueryResponse property.
                Id = u.Status.Id,
                Title = u.Status.Title
            },
            // Way 1: Map primitive type data for basic information.
            RoleNamesR = string.Join(", ", u.UserRoles
                 .OrderBy(ur => ur.Role.Name)
                 .Select(ur => ur.Role.Name)),
            // Way 2: Map complex type object for more information.
            // Either Way 1, Way 2 or both can be used.
            RolesR = u.UserRoles.OrderBy(ur => ur.Role.Name)
                 .Select(ur => new RoleQueryResponse
                 {
                     // Map each Role entity property to
                     // UserQueryResponse's RoleQueryResponse property
                     // through UserRoles.
                     Id = ur.Role.Id,
                     Name = ur.Role.Name
                 }).ToList(),
            // Hidden password with * character.
            PasswordR = new string('*', u.Password.Length),
            // Concatenated first name and list name with white space character.
            FullNameR = u.FirstName + " " + u.LastName,
            // Formatted date and time in month/day/year hour:minute:second
            // format. No need to use the CultureInfo instance for the second
            // parameter of the ToString method since CultureInfo has been
            // assigned in the base abstract Service class by the Culture
            // property assignment. If Culture property of the base abstract
            // Service class is changed within this class constructor,
            // the changed value will be used by the CultureInfo instance and
            // then ToString method.
            RegistrationDateR = u.RegistrationDate
                 .ToString("MM/dd/yyyy HH:mm.ss"),
            // Ternary operator:
            BirthDateR = u.BirthDate.HasValue ? // if (u.BirthDate != null).
                 u.BirthDate.Value.ToShortDateString() : // Assign u.BirthDate's
                                                         // value since u.BirthDate
                                                         // is nullable.
                 string.Empty, // "" string.
            // ToShortDateString returns the date in month/day/year format.
            IsOnlineR = u.IsOnline ? "Yes" : "No",
            GenderR = u.Gender.ToString(), // Assigns "Man" or "Woman".
            ScoreR = u.Score.ToString("N1") // N: Number format.
                                            // 1: 1 decimal.
                                            // C can also be used for currency.
                                            // No need to define the second
                                            // CultureInfo parameter since Culture
                                            // property has been assigned in the
                                            // base abstract Service class.
    })); // u: User entity delegate, ur: UserRole entity delegate.
}
```

```
    }
}
```

22. Right-click the Users folder then click Add -> Class to create a class file named UserAdd.cs that will include two classes UserAddRequest and UserAddHandler as below (Optionally for each class, a seperate class file may be created, therefore there may be UserAddRequest.cs and UserAddHandler.cs class files in the Users folder):

```
using CORE.Domain;
using CORE.Models;
using CORE.Services;
using MediatR;
using Microsoft.EntityFrameworkCore;
using System.ComponentModel.DataAnnotations;
using Users.APP.Domain;

namespace Users.APP.Features.Users
{
    // Request properties are created according to the data that
    // will be retrieved from APIs.
    public class UserAddRequest : Record, IRequest<CommandResponse>
    {
        // Copy all the non navigation properties from User entity.
        /*
        ErrorMessage parameter can be set in all data annotations
        to show custom validation error messages:
        Example 1: [Required(ErrorMessage = "{0} is required!")]
        where {0} is the property name which is "UserName".
        Example 2: [StringLength(30, 3,
        ErrorMessage = "{0} must be minimum {2} maximum {1} characters!")]
        where {0} is the property name which is "UserName", {1} is the
        first parameter which is 30 and {2} is the second parameter
        which is 3.
        */
        // UserName is required and can be minimum 3 maximum 30 characters.
        [Required(ErrorMessage = "{0} is required!")]
        [StringLength(30, MinimumLength = 3,
            ErrorMessage = "{0} must be minimum {2} maximum {1} characters!")]
        public string UserName { get; set; }

        [Required(ErrorMessage = "{0} is required!")]
        [StringLength(15, MinimumLength = 3,
            ErrorMessage = "{0} must be minimum {2} maximum {1} characters!")]
        public string Password { get; set; }

        [StringLength(50, ErrorMessage = "{0} must be maximum {1} characters!")]
        public string FirstName { get; set; }

        [StringLength(50, ErrorMessage = "{0} must be maximum {1} characters!")]
        public string LastName { get; set; }

        public Genders Gender { get; set; }
```

```
    public DateTime? BirthDate { get; set; }

    // We don't need to get the RegistrationDate value from the client
    // since it will be assigned automatically in the handler.

    // Minimum value can be 0, maximum value can be 5.
    /*
    The type changed from double to double? to be able to use [Required]
    attribute. If the type is double, the default value will be 0 and
    the validation will always be successful even if no value is assigned.
    */
    [Required(ErrorMessage = "{0} is required!")]
    [Range(0, 5, ErrorMessage = "{0} must be between {1} and {2}!")]
    public double? Score { get; set; }

    // We don't need to get the IsOnline value from the client
    // since it will be assigned automatically during token
    // authentication.



    // Modify (1 to many) or add (many to many) the relationship properties.
    /*
    The type changed from int to int? to be able to use [Required] attribute.
    If the type is int, the default value will be 0 and the validation will
    always be successful even if no value is assigned.
    */
    // For users-status one to many relationship.
    [Required(ErrorMessage = "{0} is required!")]
    public int? StatusId { get; set; }

    // Required may not be defined if a user may not have a role.
    // For users-roles many to many relationship.
    [Required(ErrorMessage = "At least one role is required!")]
    public List<int> RoleIds { get; set; }
}

// UserAddHandler inherits the DbService class for User entity type,
// therefore User entity database operations of the DbService
// class can be used in this class.
// UserAddHandler also implements the Mediator IRequestHandler
// interface for UserAddRequest and CommandResponse types,
// therefore IRequestHandler interface's Handle method
// definition can be implemented for types UserAddRequest and
// CommandResponse in this class, which will be used by
// the related controller's action.
public class UserAddHandler : DbService<User>,
    IRequestHandler<UserAddRequest, CommandResponse>
{
    // Constructor that passes the injected DbContext instance
    // to the DbService base class constructor.
    public UserAddHandler(DbContext db) : base(db)
    {
    }

    // Checks whether the user with the same trimmed and
    // case sensitive user name exists in the Users database table
    // first. If it doesn't exist, inserts the User entity
    // mapped from the UserAddRequest instance to the Users
    // database table.
    public async Task<CommandResponse> Handle(
        UserAddRequest request, CancellationToken cancellationToken)
    {
```

```
            if (await DbQuery().AnyAsync(u =>
                u.UserName == request.UserName.Trim(), cancellationToken
            //&& u.BirthDate == request.BirthDate
            // One or many conditions can be used by && (and), || (or)
            // and ! (not) operators.
            ))
                return Error("User with the same user name exists!");
            // u: User entity delegate.
            // Trim method removes white space characters in the
            // beginning and at the end.

            // Create a new User entity instance from the
            // UserAddRequest instance then insert the entity in the
            // Users database table (third save default parameter
            // of the DbAddAsync method is true therefore changes of the
            // DbSets are commited to the related database tables).
            var userEntity = new User
            {
                UserName = request.UserName.Trim(),
                Password = request.Password.Trim(),
                FirstName = request.FirstName?.Trim(),
                // Since request.FirstName may be null, ? is used after
                // meaning that if request.FirstName is null, assign
                // null to the User entity's FirstName property else
                // assign the trimmed value of request.FirstName.
                LastName = request.LastName?.Trim(),
                // Since request.LastName may be null, ? is used after
                // meaning that if request.LastName is null, assign
                // null to the User entity's LastName property else
                // assign the trimmed value of request.LastName.
                RegistrationDate = DateTime.Now,
                BirthDate = request.BirthDate,
                Gender = request.Gender,
                Score = request.Score ?? 0,
                StatusId = request.StatusId ?? 0,
                // ??: Null coalescing operator: If request.StatusId is
                // null assign 0 else assign request.StatusId value to the
                // User entity StatusId property. request.StatusId.Value
                // may also be used since request.StatusId is required.
                // Same for request.Score.
                RoleIds = request.RoleIds
            };
            await DbAddAsync(userEntity, cancellationToken);

            // Entity's Id value will be updated by the database after insert.
            return Success(userEntity.Id, "User created successfully.");
        }
    }
}
```

23. Right-click the Users folder then click Add -> Class to create a class file named UserUpdate.cs that will include two classes UserUpdateRequest and UserUpdateHandler as below (Optionally for each class, a seperate class file may be created, therefore there may be UserUpdateRequest.cs and UserUpdateHandler.cs class files in the Users folder):

```
using CORE.Domain;
using CORE.Models;
using CORE.Services;
using MediatR;
using Microsoft.EntityFrameworkCore;
using System.ComponentModel.DataAnnotations;
using Users.APP.Domain;

namespace Users.APP.Features.Users
{
    // Request properties are created according to the data that
    // will be retrieved from APIs.
    public class UserUpdateRequest : Record, IRequest<CommandResponse>
    {
        // Copy all the non navigation properties from User entity.
        // UserName is required and can be minimum 3 maximum 30 characters.
        [Required, StringLength(30, MinimumLength = 3)]
        public string UserName { get; set; }

        [Required, StringLength(15, MinimumLength = 3)]
        public string Password { get; set; }

        [StringLength(50)]
        public string FirstName { get; set; }

        [StringLength(50)]
        public string LastName { get; set; }

        public Genders Gender { get; set; }

        public DateTime? BirthDate { get; set; }

        // We don't need to get the RegistrationDate value from the client
        // since it will be assigned automatically in UserAddHandler class.

        [Required, Range(0, 5)]
        public double? Score { get; set; }

        // We don't need to get the IsOnline value from the client
        // since it will be assigned automatically during token
        // authentication.



        // Modify (1 to many) or add (many to many) the relationship properties.
        /*
        The type changed from int to int? to be able to use [Required] attribute.
        If the type is int, the default value will be 0 and the validation will
        always be successful even if no value is assigned.
        */
        // For users-status one to many relationship.
        [Required]
        public int? StatusId { get; set; }

        // Required may not be defined if a user may not have a role.
        // For users-roles many to many relationship.
        [Required]
        public List<int> RoleIds { get; set; }
    }

    // UserUpdateHandler inherits the DbService class for User entity type,
    // therefore User entity database operations of the DbService
    // class can be used in this class.
```

```
// UserUpdateHandler also implements the Mediator IRequestHandler
// interface for UserUpdateRequest and CommandResponse types,
// therefore IRequestHandler interface's Handle method
// definition can be implemented for types UserUpdateRequest and
// CommandResponse in this class, which will be used by
// the related controller's action.
public class UserUpdateHandler : DbService<User>,
    IRequestHandler<UserUpdateRequest, CommandResponse>
{
    // Constructor that passes the injected DbContext instance
    // to the DbService base class constructor.
    public UserUpdateHandler(DbContext db) : base(db)
    {
    }

    // Override the DbQuery method of the base DbService class to
    // include the UserRole entities to the query. Therefore
    // during update operation, the related UserRole entities
    // will be deleted first and then new related UserRole entities
    // will be inserted through the RoleIds property of the
    // UserUpdateRequest instance.
    protected override IQueryable<User> DbQuery()
    {
        return base.DbQuery().Include(u => u.UserRoles);
        // u: User entity delegate.
    }

    // Checks whether the user with the same trimmed and
    // case sensitive user name other than the request record
    // exists in the Users database table first. If it doesn't exist,
    // gets the User entity by ID and if found updates the User
    // entity mapped from the UserUpdateRequest instance in the Users
    // database table.
    public async Task<CommandResponse> Handle(UserUpdateRequest request,
        CancellationToken cancellationToken)
    {
        if (await DbQuery().AnyAsync(u => u.Id != request.Id &&
            u.UserName == request.UserName.Trim(), cancellationToken))
            return Error("User with the same user name exists!");
        // u: User entity delegate.
        // Trim method removes white space characters in the
        // beginning and at the end.

        // Get the User entity by ID from the Users database table
        // and check if it is null.
        var userEntity = await DbSingleAsync(request.Id, cancellationToken);
        if (userEntity is null)
            return Error("User not found!");

        // Delete the related UserRole entities first.
        if (request.RoleIds is not null)
            DbRemove(userEntity.UserRoles);

        // Then update the User entity properties from the
        // UserUpdateRequest instance then update the entity in the
        // Users database table (third save default parameter
        // of the DbUpdateAsync method is true therefore changes of the
        // DbSets are commited to the related database tables).
        userEntity.UserName = request.UserName.Trim();
        userEntity.Password = request.Password.Trim();
        userEntity.FirstName = request.FirstName?.Trim();
        // Since request.FirstName may be null, ? is used after
        // meaning that if request.FirstName is null, assign
```

```
            // null to the User entity's FirstName property else
            // assign the trimmed value of request.FirstName.
            userEntity.LastName = request.LastName?.Trim();
            // Since request.LastName may be null, ? is used after
            // meaning that if request.LastName is null, assign
            // null to the User entity's LastName property else
            // assign the trimmed value of request.LastName.
            // We don't need to update the RegistrationDate value
            // since it will be assigned automatically in the
            // UserAddHandler class.
            userEntity.BirthDate = request.BirthDate;
            userEntity.Gender = request.Gender;
            userEntity.Score = request.Score ?? 0;
            userEntity.StatusId = request.StatusId ?? 0;
            // ??: Null coalescing operator: If request.StatusId is
            // null assign 0 else assign request.StatusId value to the
            // User entity StatusId property. request.StatusId.Value
            // may also be used since request.StatusId is required.
            // Same for request.Score.
            userEntity.RoleIds = request.RoleIds;
            await DbUpdateAsync(userEntity, cancellationToken);

            return Success(userEntity.Id, "User updated successfully!");
        }
    }
}
```

24. Right-click the Users folder then click Add -> Class to create a class file named UserRemove.cs that will include two classes UserRemoveRequest and UserRemoveHandler as below (Optionally for each class, a seperate class file may be created, therefore there may be UserRemoveRequest.cs and UserRemoveHandler.cs class files in the Users folder):

```
using CORE.Domain;
using CORE.Models;
using CORE.Services;
using MediatR;
using Microsoft.EntityFrameworkCore;
using Users.APP.Domain;

namespace Users.APP.Features.Users
{
    // Delete operation will be performed by the Id property inherited
    // from the Record base class. Therefore, no additional properties
    // are needed to be defined.
    public class UserRemoveRequest : Record, IRequest<CommandResponse>
    {
    }

    // UserRemoveHandler class injects the DbContext instance for
    // querying and deleting an existing user in the Users database
    // table.
    // The Handle method implemented from the IRequestHandler
    // interface gets the User entity by ID first.
    // If found, deletes the User entity from the Users database table.
```

```
    public class UserRemoveHandler : DbService<User>,
        IRequestHandler<UserRemoveRequest, CommandResponse>
    {
        // Constructor that passes the injected DbContext instance
        // to the DbService base class constructor.
        public UserRemoveHandler(DbContext db) : base(db)
        {
        }

        // Override the DbQuery method of the base DbService class to
        // include the UserRole entities to the query. Therefore
        // during delete operation, the related UserRole entities
        // will be deleted first. Then the user entity will be deleted.
        protected override IQueryable<User> DbQuery()
        {
            return base.DbQuery().Include(u => u.UserRoles);
            // u: User entity delegate.
        }

        // Deletes an existing user from the Users database table
        // with the related UserRole entities if it is found by request's ID.
        public async Task<CommandResponse> Handle(UserRemoveRequest request,
            CancellationToken cancellationToken)
        {
            // Get the User entity by ID from the Users database table
            // and check whether it is null or not.
            var userEntity = await DbSingleAsync(request.Id, cancellationToken);
            if (userEntity is null)
                return Error("User not found!");

            // Delete the related UserRole entities first.
            DbRemove(userEntity.UserRoles);

            // Then delete the User entity from the Users database table.
            await DbRemoveAsync(userEntity, cancellationToken);

            return Success(userEntity.Id, "User deleted successfully.");
        }
    }
}
```

25. Right-click the Features folder then click Add -> New Folder to create a folder called Statuses.

26. Right-click the Statuses folder then click Add -> Class to create a class file named StatusQuery.cs that will include three classes StatusQueryRequest, StatusQueryResponse and StatusQueryHandler as below (Optionally for each class, a seperate class file may be created, therefore there may be StatusQueryRequest.cs, StatusQueryResponse and StatusQueryHandler.cs class files in the Statuses folder):

```
using CORE.Domain;
using CORE.Services;
using MediatR;
```

```
using Microsoft.EntityFrameworkCore;
using Users.APP.Domain;
using Users.APP.Features.Users;

namespace Users.APP.Features.Statuses
{
    // Request properties are created according to the data that
    // will be retrieved from APIs.
    public class StatusQueryRequest : Record,
        IRequest<IQueryable<StatusQueryResponse>>
    {
    }

    // Response properties are created according to the data to be
    // presented in API responses.
    public class StatusQueryResponse : Record
    {
        // Copy all the non navigation properties from Status entity.
        public string Title { get; set; }



        // Add the new properties ending with R declaring Response
        // for custom properties or formatted string value properties.
        // Way 1: Map primitive type properties for basic information.
        public string UserNamesR { get; set; }

        // Property value for the count of the users of the status.
        public int UserCountR { get; set; }

        // Way 2: Map complex type object for more information.
        // Either Way 1, Way 2 or both can be used.
        public List<UserQueryResponse> UsersR { get; set; }
    }

    // StatusQueryHandler inherits the DbService class for Status entity type,
    // therefore Status entity database operations of the DbService
    // class can be used in this class.
    // StatusQueryHandler also implements the Mediator IRequestHandler
    // interface for StatusQueryRequest and IQueryable of StatusQueryResponse
    // types, therefore IRequestHandler interface's Handle method
    // definition can be implemented for types StatusQueryRequest and
    // IQueryable of StatusQueryResponse in this class, which will be used by
    // the related controller's action.
    public class StatusQueryHandler : DbService<Status>,
        IRequestHandler<StatusQueryRequest, IQueryable<StatusQueryResponse>>
    {
        // Constructor that passes the injected DbContext instance
        // to the DbService base class constructor.
        public StatusQueryHandler(DbContext db) : base(db)
        {
        }

        // Overrides the base virtual DbQuery method to include
        // User entities data to the query to be used in the Handle
        // method and orders the query by Status entity Title
        // property ascending.
        // The overridden DbQuery method can be used in any method
        // below by invoking the DbQuery method.
        protected override IQueryable<Status> DbQuery()
        {
            return base.DbQuery() // "select * from Statuses"
                                  // entity query.
```

```
        .Include(s => s.Users) // Includes the relational
                               // User entities data to the query
                               // by using left outer join.
        .OrderBy(s => s.Title); // Orders the query by Title
                                // property of the Status entity
                                // in ascending order.
                                // OrderByDescending can be used
                                // for descending order.
    // s: Status entity delegate.
}

// Gets the updated query from the DbQuery method then projects
// the Status entities to status query responses and returns the
// StatusQueryResponse query.
// The returned query can be executed by invoking ToListAsync,
// SingleOrDefaultAsync, FirstOrDefaultAsync, AnyAsync, etc.
// LINQ (Language Integrated Query) methods and then the
// returned result can be used.
public Task<IQueryable<StatusQueryResponse>> Handle(
    StatusQueryRequest request, CancellationToken cancellationToken)
{
    var query = DbQuery().Select(s => new StatusQueryResponse
    {
        // Map each Status entity property to
        // StatusQueryResponse property.
        Id = s.Id,
        Title = s.Title,
        // Way 1: Map primitive type data for basic information.
        UserNamesR = string.Join(", ", s.Users.Select(u => u.UserName)),
        // string type's Join method gets a seperator as the first
        // parameter and gets a collection of strings as the second
        // parameter then concatenates each string item in collection
        // with the seperator and returns a string.
        UserCountR = s.Users.Count,
        // Count property of the List type returns the item count
        // of the collection.
        // Way 2: Map complex type object for more information.
        // Either Way 1, Way 2 or both can be used.
        UsersR = s.Users.Select(u => new UserQueryResponse
        {
            // Map each User entity property to
            // StatusQueryResponse's UserQueryResponse property.
            Id = u.Id,
            UserName = u.UserName,
            PasswordR = new string('*', u.Password.Length),
            // Let's hide the password with *.
            FirstName = u.FirstName,
            LastName = u.LastName,
            FullNameR = u.FirstName + " " + u.LastName,
            // Concatenate the first name and last name
            // with a white space for the full name.
            RegistrationDate = u.RegistrationDate,
            BirthDate = u.BirthDate,
            Gender = u.Gender,
            IsOnline = u.IsOnline,
            Score = u.Score,
            StatusId = u.StatusId
        }).ToList()
    });
    // s: Status entity delegate, u: User entity delegate.

    return Task.FromResult(query);
}
```

```
    }
}
```

27. Right-click the Statuses folder then click Add -> Class to create a class file named StatusAdd.cs that will include two classes StatusAddRequest and StatusAddHandler as below (Optionally for each class, a seperate class file may be created, therefore there may be StatusAddRequest.cs and StatusAddHandler.cs class files in the Statuses folder):

```
using CORE.Domain;
using CORE.Models;
using CORE.Services;
using MediatR;
using Microsoft.EntityFrameworkCore;
using System.ComponentModel.DataAnnotations;
using Users.APP.Domain;

namespace Users.APP.Features.Statuses
{
    // Request properties are created according to the data that
    // will be retrieved from APIs.
    public class StatusAddRequest : Record, IRequest<CommandResponse>
    {
        // Copy all the non navigation properties from Status entity.
        [Required, StringLength(5)]
        public string Title { get; set; }
    }

    // StatusAddHandler class injects the DbContext instance for
    // querying and inserting a new status to the Statuses database table.
    // The Handle method implemented from the IRequestHandler
    // interface checks whether the status with the same trimmed and
    // case sensitive title exists in the Statuses database table first.
    // If it doesn't exist, inserts the Status entity mapped from the
    // StatusAddRequest instance to the Statuses database table.
    public class StatusAddHandler : DbService<Status>,
        IRequestHandler<StatusAddRequest, CommandResponse>
    {
        // Constructor that passes the injected DbContext instance
        // to the DbService base class constructor.
        public StatusAddHandler(DbContext db) : base(db)
        {
        }

        // Inserts a new status to the Statuses database table if a
        // status with the same title doesn't exist.
        public async Task<CommandResponse> Handle(StatusAddRequest request,
            CancellationToken cancellationToken)
        {
            if (await DbQuery()
                .AnyAsync(statusEntity =>
                    statusEntity.Title == request.Title.Trim(),
                        cancellationToken))
                return Error("Status with the same title exists!");
            // Trim method removes white space characters in the
            // beginning and at the end.
```

```
            // Create a new Status entity instance from the StatusAddRequest
            // instance then insert the entity in the Statuses database
            // table (third save default parameter of the DbAddAsync
            // method is true therefore changes of the DbSets are
            // commited to the related database tables).
            var statusEntity = new Status
            {
                Title = request.Title.Trim()
            };
            await DbAddAsync(statusEntity, cancellationToken);

            // Return a successful command response with the new
            // status ID.
            // Entity's Id value will be updated by the database
            // after the insert operation.
            return Success(statusEntity.Id, "Status created successfully.");
        }
    }
}
```

28. Right-click the Statuses folder then click Add -> Class to create a class file named StatusUpdate.cs that will include two classes StatusUpdateRequest and StatusUpdateHandler as below (Optionally for each class, a seperate class file may be created, therefore there may be StatusUpdateRequest.cs and StatusUpdateHandler.cs class files in the Statuses folder):

```
using CORE.Domain;
using CORE.Models;
using CORE.Services;
using MediatR;
using Microsoft.EntityFrameworkCore;
using System.ComponentModel.DataAnnotations;
using Users.APP.Domain;

namespace Users.APP.Features.Statuses
{
    // Request properties are created according to the data that
    // will be retrieved from APIs.
    public class StatusUpdateRequest : Record, IRequest<CommandResponse>
    {
        // Copy all the non navigation properties from Status entity.
        // Title can't be null and can be maximum 5 characters.
        [Required, StringLength(5)]
        public string Title { get; set; }
    }

    // StatusUpdateHandler class injects the DbContext instance for
    // querying and updating an existing status in the Statuses database
    // table.
    // The Handle method implemented from the IRequestHandler
    // interface checks whether the status with the same trimmed and
    // case sensitive title other than the status request record
    // exists in the Statuses database table first.
    // If it doesn't exist, updates the Status entity mapped from the
    // StatusUpdateRequest instance in the Statuses database table.
    public class StatusUpdateHandler : DbService<Status>,
        IRequestHandler<StatusUpdateRequest, CommandResponse>
```

```
    {
        // Constructor that passes the injected DbContext instance
        // to the DbService base class constructor.
        public StatusUpdateHandler(DbContext db) : base(db)
        {
        }

        // Updates an existing status in the Statuses database table if a
        // status with the same title other than the status request record
        // doesn't exist.
        public async Task<CommandResponse> Handle(StatusUpdateRequest request,
            CancellationToken cancellationToken)
        {
            if (await DbQuery()
                .AnyAsync(statusEntity => statusEntity.Id != request.Id &&
                    statusEntity.Title == request.Title.Trim(),
                        cancellationToken))
                return Error("Status with the same title exists!");

            // Get the Status entity by ID from the Statuses database table
            // and check whether it is null or not.
            var statusEntity = await DbSingleAsync(request.Id, cancellationToken);
            if (statusEntity is null)
                return Error("Status not found!");

            // Update Status entity properties from StatusUpdateRequest properties.
            // Then update the entity in the Statuses database table (third save
            // default parameter of the DbUpdateAsync method is true
            // therefore changes of the DbSets are commited to the related
            // database tables).
            statusEntity.Title = request.Title.Trim();
            await DbUpdateAsync(statusEntity, cancellationToken);

            return Success(statusEntity.Id, "Status updated successfully.");
        }
    }
}
```

29. Right-click the Statuses folder then click Add -> Class to create a class file named StatusRemove.cs that will include two classes StatusRemoveRequest and StatusRemoveHandler as below (Optionally for each class, a seperate class file may be created, therefore there may be StatusRemoveRequest.cs and StatusRemoveHandler.cs class files in the Statuses folder):

```
using CORE.Domain;
using CORE.Models;
using CORE.Services;
using MediatR;
using Microsoft.EntityFrameworkCore;
using Users.APP.Domain;

namespace Users.APP.Features.Statuses
{
    // Delete operation will be performed by the Id property inherited
    // from the Record base class. Therefore, no additional properties
    // are needed to be defined.
    public class StatusRemoveRequest : Record, IRequest<CommandResponse>
    {
    }
```

```
    // StatusRemoveHandler class injects the DbContext instance for
    // querying and deleting an existing status in the Statuses database
    // table.
    // The Handle method implemented from the IRequestHandler
    // interface gets the Status entity by ID first.
    // If found, first checks if there are any related Users and
    // if none, deletes the Status entity from the Statuses database table.
    public class StatusRemoveHandler : DbService<Status>,
        IRequestHandler<StatusRemoveRequest, CommandResponse>
    {
        // Constructor that passes the injected DbContext instance
        // to the DbService base class constructor.
        public StatusRemoveHandler(DbContext db) : base(db)
        {
        }

        // Override the DbQuery method to include the related Users
        // to the Status entity query.
        protected override IQueryable<Status> DbQuery()
        {
            return base.DbQuery().Include(statusEntity => statusEntity.Users);
        }

        // Deletes an existing status from the Statuses database table if
        // it is found by request's ID and it has no related users.
        public async Task<CommandResponse> Handle(StatusRemoveRequest request,
            CancellationToken cancellationToken)
        {
            // Get the Status entity by ID from the Statuses database table
            // and check whether it is null or not.
            var statusEntity = await DbSingleAsync(request.Id, cancellationToken);
            if (statusEntity is null)
                return Error("Status not found!");

            // Check if there are any related Users for the Status entity.
            if (statusEntity.Users.Any()) // if (statusEntity.Users.Count > 0)
                                          // can also be written.
                return Error("Status has relational users!");

            // Delete the Status entity from the Statuses database table
            // (third save default parameter of the DbRemoveAsync method is
            // true therefore changes of the DbSets are commited to
            // the related database tables).
            await DbRemoveAsync(statusEntity, cancellationToken);

            return Success(statusEntity.Id, "Status deleted successfully.");
        }
    }
}
```

C) Users.API Project for CRUD Operations

1. In Program.cs IoC (Inversion of Control) Container section, add the dependency registration for using Mediator under the DbContext dependency registration as below (The complete implementation of the Program.cs can be found in the Additional Information of this document (1)):

```
// Way 1:
// Registers MediatR services with the dependency injection
// container.
// MediatR is a popular .NET library that implements the
// mediator pattern, enabling decoupled communication
// between components by sending requests (commands, queries,
// events) to handlers without direct dependencies.
// The configuration below scans the assembly containing the
// 'UsersDb' type for any classes that implement MediatR handler
// interfaces (such as IRequestHandler, INotificationHandler,
// etc.).
// This allows automatic discovery and registration of all
// MediatR handlers in the specified assembly.
// As a result, you can inject the IMediator interface into
// controllers and use it to send requests or publish notifications,
// which will be routed to the appropriate handlers.
//builder.Services.AddMediatR(
//  config => config.RegisterServicesFromAssembly(
//      typeof(UsersDb).Assembly));
// Way 2:
// Iterates through all assemblies currently loaded in the
// application's app domain.
// For each assembly, registers all MediatR handlers (such as
// IRequestHandler, INotificationHandler, etc.) found within
// that assembly with the dependency injection container.
// This enables MediatR to automatically discover and wire up
// handlers from any loaded assembly, allowing for modular handler
// organization and dynamic handler loading (e.g., from plugins
// or feature assemblies).
// Note: Registering handlers from all assemblies can be useful
// in large or modular applications, but may introduce duplicate
// registrations or performance overhead if not managed carefully.
foreach (var assembly in AppDomain.CurrentDomain.GetAssemblies())
{
    builder.Services.AddMediatR(
        config => config.RegisterServicesFromAssemblies(assembly));
}
```

2. A Controller class has one or more Actions (methods) which HTTP requests can be sent.
   Generally each Action returns an IActionResult instance for HTTP responses after executed.
   In Web API, generally HTTP responses with status codes 200 (OK), 201 (Created), 204 (No Content), 400 (Bad Request), 404 (Not Found) and 500 (Internal Server Error) are returned.
   You don't need to modify the WeatherForecastController.cs in the Controllers folder, it is just an example and good for testing if your project is running properly.

```
using Microsoft.AspNetCore.Mvc;

namespace Users.API.Controllers
{
    /// <summary>
    /// API controller for a weather forecast demonstration.
    /// Inherits from the ControllerBase class for using
    /// some of the base methods such as Ok, NoContent,
    /// BadRequest, etc.
    /// </summary>
    [ApiController] // Attribute declaring WeatherForecastController
                    // is an API controller.
    [Route("[controller]")] // The route of the controller is:
                            // ~/WeatherForecast
    public class WeatherForecastController : ControllerBase
    {
        // Private static field for storing weather statuses array.
        private static readonly string[] Summaries = new[]
        {
            "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm",
            "Balmy", "Hot", "Sweltering", "Scorching"
        };

        // Optional private read only field for ILogger instance
        // Dependency Injection through the constructor.
        // Not used in any action but can be used for logging.
        private readonly ILogger<WeatherForecastController> _logger;

        public WeatherForecastController(
            ILogger<WeatherForecastController> logger)
        {
            _logger = logger;
        }

        [HttpGet(Name = "GetWeatherForecast")] // Declares that the action
                                               // is a HTTP GET action.
        // Name property specifies a unique identifier for the specific route
        // (~/WeatherForecast) which allows to generate links to this endpoint
        // from other parts of the application. It behaves as a route tag and
        // does not change the actual public URL path.
        public IEnumerable<WeatherForecast> Get()
        {
            // Returns an array (collection) of random dates, temperatures
            // and statuses through the WeatherForecast response model
            // as JSON (JavaScript Object Notation).
            return Enumerable.Range(1, 5).Select(index => new WeatherForecast
            {
                Date = DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
                TemperatureC = Random.Shared.Next(-20, 55),
                Summary = Summaries[Random.Shared.Next(Summaries.Length)]
            })
            .ToArray();
        }
    }
}
```

3. Download the Scaffolding Templates that will generate the code for the controllers automatically from:
   https://need4code.com/DotNet/Home/Index?path=.NET%5C00_Files%5CScaffolding%20Templates%5CTemplates.7z

   Extract the Templates folder in the compressed file to your Users.API Project.

   Right-click the Templates folder then click Exclude From Project for the template codes not being compiled and published.

   If you want to see the excluded folders or files of your project, you can click the fifth icon from left in top of the Solution Explorer with description Show All Files.

   The excluded files or folders are seen as dashed points in Solution Explorer.

   If you want to include a folder or file in your project, right-click the file or folder with dashed points then click Include In Project, therefore the codes will be compiled and published.

   You don't have to use these templates, however if you choose not to, you need to write or modify the dependency injections and actions.

   Note: If you get any exceptions during scaffolding, create the UsersDbFactory class in the Users.APP/Domain folder.

   Note: For Rider run:
   "dotnet aspnet-codegenerator controller -name {ModelName}Controller -m {Namespace}.{ModelName} -dc {Namespace}.{DbContextName} --relativeFolderPath Controllers"
   in the terminal for scaffolding with the templates under the Templates folder.

4. Right-click the Controllers folder then Add -> Controller -> Common -> API -> API Controller with actions, using Entity Framework, and select Role entity as Model class, select UsersDb as DbContext class and give the name RolesController as Controller name. You can see the implementation below:

```
#nullable disable
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using MediatR;
using CORE.Models;
using Users.APP.Features.Roles;

//Generated from Custom Microservices Template.
namespace Users.API.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class RolesController : ControllerBase
    {
        private readonly ILogger<RolesController> _logger;
        private readonly IMediator _mediator;

        // Constructor: injects logger to log the errors to
        // Kestrel Console or Output Window and mediator
        public RolesController(ILogger<RolesController> logger,
            IMediator mediator)
        {
            _logger = logger;
            _mediator = mediator;
        }

        // GET: api/Roles
        [HttpGet]
        public async Task<IActionResult> Get()
        {
            try
            {
                // Send a query request to get query response
                var response = await _mediator
                    .Send(new RoleQueryRequest());
                // Convert the query response to a list
                var list = await response.ToListAsync();
                // If there are items, return them with 200 OK
                if (list.Any())
                    return Ok(list);
                // If no items found, return 204 No Content
                return NoContent();
            }
            catch (Exception exception)
            {
                // Log the exception
                _logger.LogError("RolesGet Exception: " +
                    exception.Message);
                // Return 500 Internal Server Error with an error
                // command response with message
                return StatusCode(StatusCodes.Status500InternalServerError,
                    new CommandResponse(false,
                        "An exception occured during RolesGet."));
            }
        }

        // GET: api/Roles/5
        [HttpGet("{id}")]
        public async Task<IActionResult> Get(int id)
        {
            try
```

```
    {
        // Send a query request to get query response
        var response = await _mediator
            .Send(new RoleQueryRequest());
        // Find the item with the given id
        var item = await response
            .SingleOrDefaultAsync(r => r.Id == id);
        // If item found, return it with 200 OK
        if (item is not null)
            return Ok(item);
        // If item not found, return 204 No Content
        return NoContent();
    }
    catch (Exception exception)
    {
        // Log the exception
        _logger.LogError("RolesGetById Exception: " +
            exception.Message);
        // Return 500 Internal Server Error with an error
        // command response with message
        return StatusCode(StatusCodes.Status500InternalServerError,
            new CommandResponse(false,
                "An exception occured during RolesGetById."));
    }
}

// POST: api/Roles
[HttpPost]
public async Task<IActionResult> Post(RoleAddRequest request)
{
    try
    {
        // Check if the request model is valid through
        // data annotations
        if (ModelState.IsValid)
        {
            // Send the create request
            var response = await _mediator.Send(request);
            // If creation is successful, return 200 OK with
            // success command response
            if (response.IsSuccessful)
            {
                //return CreatedAtAction(nameof(Get),
                //  new { id = response.Id }, response);
                return Ok(response);
            }
            // If creation failed, add error command
            // response message to model state
            ModelState.AddModelError("RolesPost", response.Message);
        }
        // Return 400 Bad Request with all data annotation
        // validation error messages and the error command
        // response message if added seperated by |
        return BadRequest(new CommandResponse(false,
            string.Join("|", ModelState.Values
                .SelectMany(v => v.Errors)
                .Select(e => e.ErrorMessage))));
    }
    catch (Exception exception)
    {
        // Log the exception
        _logger.LogError("RolesPost Exception: " +
            exception.Message);
```

```
            // Return 500 Internal Server Error with an error
            // command response with message
            return StatusCode(StatusCodes.Status500InternalServerError,
                new CommandResponse(false,
                    "An exception occured during RolesPost."));
        }
    }

    // PUT: api/Roles
    [HttpPut]
    public async Task<IActionResult> Put(RoleUpdateRequest request)
    {
        try
        {
            // Check if the request model is valid through
            // data annotations
            if (ModelState.IsValid)
            {
                // Send the update request
                var response = await _mediator.Send(request);
                // If update is successful, return 200 OK
                // with success command response
                if (response.IsSuccessful)
                {
                    //return NoContent();
                    return Ok(response);
                }
                // If update failed, add error command response
                // message to model state
                ModelState.AddModelError("RolesPut", response.Message);
            }
            // Return 400 Bad Request with all data annotation
            // validation error messages and the error command
            // response message if added seperated by |
            return BadRequest(new CommandResponse(
                false, string.Join("|", ModelState.Values
                    .SelectMany(v => v.Errors)
                    .Select(e => e.ErrorMessage))));
        }
        catch (Exception exception)
        {
            // Log the exception
            _logger.LogError("RolesPut Exception: " +
                exception.Message);
            // Return 500 Internal Server Error with an error
            // command response with message
            return StatusCode(StatusCodes.Status500InternalServerError,
                new CommandResponse(false,
                    "An exception occured during RolesPut."));
        }
    }

    // DELETE: api/Roles/5
    [HttpDelete("{id}")]
    public async Task<IActionResult> Delete(int id)
    {
        try
        {
            // Send the delete request
            var response = await _mediator
                .Send(new RoleRemoveRequest() { Id = id });
            // If delete is successful, return 200 OK with
            // success command response
```

```
                if (response.IsSuccessful)
                {
                    //return NoContent();
                    return Ok(response);
                }
                // If delete failed, add error command response
                // message to model state
                ModelState.AddModelError("RolesDelete", response.Message);
                // Return 400 Bad Request with the error
                // command response message
                return BadRequest(new CommandResponse(false,
                    string.Join("|", ModelState.Values
                        .SelectMany(v => v.Errors)
                        .Select(e => e.ErrorMessage))));
            }
            catch (Exception exception)
            {
                // Log the exception
                _logger.LogError("RolesDelete Exception: " +
                    exception.Message);
                // Return 500 Internal Server Error with an error
                // command response with message
                return StatusCode(StatusCodes.Status500InternalServerError,
                    new CommandResponse(false,
                        "An exception occured during RolesDelete."));
            }
        }
    }
}
```

5. Right-click the Controllers folder then Add -> Controller -> Common -> API -> API Controller with actions, using Entity Framework, and select Status entity as Model class, select UsersDb as DbContext class and give the name StatusesController as Controller name.

6. Right-click the Controllers folder then Add -> Controller -> Common -> API -> API Controller with actions, using Entity Framework, and select User entity as Model class, select UsersDb as DbContext class and give the name UsersController as Controller name.

IV) Projects Setup for Authentication and Authorization

Note: Authentication: Who are you? Authroization: What can you do?

A) CORE Project for Authentication and Authorization

1. Right-click the CORE Project then click Manage NuGet Packages then in Browse tab search for Microsoft.AspNetCore.Authentication.JwtBearer latest version starting with 8 and install.

2. Add a new folder named Authentication in the CORE Project.

3. Add a new folder called Models in the Authentication folder.

4. Add a model class named JwtRequest (JWT: Json Web Token) in the Models folder of the Authentication folder as below:

```
using System.ComponentModel.DataAnnotations;
using System.Text.Json.Serialization;

namespace CORE.Authentication.Models
{
    /// <summary>
    /// Represents a request to generate a JWT (Json Web Token)
    /// response including JWT access token string containing
    /// user credentials and refresh token string.
    /// </summary>
    public class JwtRequest
    {
        /// <summary>
        /// The username of the user requesting the token response.
        /// </summary>
        [Required]
        public string UserName { get; set; }

        /// <summary>
        /// The password of the user requesting the token response.
        /// </summary>
        [Required]
        public string Password { get; set; }

        /// <summary>
        /// Gets or sets the security key used for signing or
        /// validating JWT.
        /// This property is ignored during JSON serialization and
        /// deserialization.
        /// The [JsonIgnore] attribute ensures that the security key
        /// is not exposed in JSON payloads sent to or received from
        /// clients.
        /// Typically, this value is assigned from application's
        /// configuration through IConfiguration instance injection
        /// and used for cryptographic operations in JWT handling.
        /// </summary>
        [JsonIgnore]
        public string SecurityKey { get; set; }
```

```
        /// <summary>
        /// Gets or sets the issuer (iss) claim for the JWT.
        /// This value identifies the principal (generally API server
        /// application) that issued the JWT.
        /// The [JsonIgnore] attribute ensures that the issuer is not
        /// exposed in JSON payloads sent to or received from clients.
        /// Typically, this value is assigned from application's
        /// configuration through IConfiguration instance injection
        /// and used during JWT generation and validation.
        /// </summary>
        [JsonIgnore]
        public string Issuer { get; set; }

        /// <summary>
        /// Gets or sets the audience (aud) claim for the JWT.
        /// This value identifies the recipients (generally client
        /// applications) that the JWT is intended for.
        /// The [JsonIgnore] attribute ensures that the audience
        /// is not exposed in JSON payloads sent to or received from clients.
        /// Typically, this value is assigned from application's
        /// configuration through IConfiguration instance injection
        /// and used during JWT generation and validation.
        /// </summary>
        [JsonIgnore]
        public string Audience { get; set; }
    }
}
```

5. Add a model class named JwtResponse in the Models folder of the Authentication folder as below:

```
namespace CORE.Authentication.Models
{
    /// <summary>
    /// Represents the response to a JwtRequest or JwtRefreshRequest,
    /// including the JWT (JSON Web Token) and refresh token.
    /// </summary>
    public class JwtResponse
    {
        /// <summary>
        /// The generated JWT (access token).
        /// </summary>
        public string Jwt { get; set; }

        /// <summary>
        /// Gets or sets the refresh token assigned to the user.
        /// This token is used to request a new JWT without
        /// requiring re-authentication, typically after the
        /// original JWT has expired.
        /// </summary>
        public string RefreshToken { get; set; }
    }
}
```

6. Add a model class named JwtRefreshRequest in the Models folder of the Authentication folder as below:

```
using System.ComponentModel.DataAnnotations;
using System.Text.Json.Serialization;

namespace CORE.Authentication.Models
{
    /// <summary>
    /// Represents the request for refreshing a JWT
    /// (JSON Web Token). Contains both the JWT and
    /// the refresh token to obtain a new JWT.
    /// </summary>
    public class JwtRefreshRequest
    {
        /// <summary>
        /// The expired or soon-to-expire JWT that
        /// needs refreshing.
        /// </summary>
        [Required]
        public string Jwt { get; set; }

        /// <summary>
        /// The refresh token used to validate the
        /// user and issue a new JWT.
        /// </summary>
        [Required]
        public string RefreshToken { get; set; }

        /// <summary>
        /// Gets or sets the security key used for signing
        /// or validating JWT.
        /// This property is ignored during JSON serialization
        /// and deserialization.
        /// The [JsonIgnore] attribute ensures that the security
        /// key is not exposed in JSON payloads sent to or
        /// received from clients.
        /// Typically, this value is assigned from application's
        /// configuration through IConfiguration instance injection
        /// and used for cryptographic operations in JWT handling.
        /// </summary>
        [JsonIgnore]
        public string SecurityKey { get; set; }

        /// <summary>
        /// Gets or sets the issuer (iss) claim for the JWT.
        /// This value identifies the principal (generally API
        /// server application) that issued the JWT.
        /// The [JsonIgnore] attribute ensures that the issuer
        /// is not exposed in JSON payloads sent to or received
        /// from clients.
        /// Typically, this value is assigned from application's
        /// configuration through IConfiguration instance injection
        /// and used during JWT generation and validation.
        /// </summary>
        [JsonIgnore]
        public string Issuer { get; set; }
```

```
        /// <summary>
        /// Gets or sets the audience (aud) claim for the JWT.
        /// This value identifies the recipients (generally client
        /// applications) that the JWT is intended for.
        /// The [JsonIgnore] attribute ensures that the audience
        /// is not exposed in JSON payloads sent to or received
        /// from clients.
        /// Typically, this value is assigned from application's
        /// configuration through IConfiguration instance injection
        /// and used during JWT generation and validation.
        /// </summary>
        [JsonIgnore]
        public string Audience { get; set; }
    }
}
```

7. Add a new folder called Services in the Authentication folder.

8. Add an interface named IJwtAuthService in the Services folder of the Authentication folder as below:

```
using CORE.Authentication.Models;
using System.Security.Claims;

namespace CORE.Authentication.Services
{
    /// <summary>
    /// Provides JWT (JSON Web Token) based authentication operations,
    /// including access token (JWT) and refresh token generation and
    /// claim extraction.
    /// </summary>
    public interface IJwtAuthService
    {
        /// <summary>
        /// Returns a JWT response including JWT (access token) and
        /// refresh token.
        /// </summary>
        /// <param name="userId">The unique ID of the user.</param>
        /// <param name="userName">The username of the user.</param>
        /// <param name="userRoleNames">A collection of role names
        /// assigned to the user.</param>
        /// <param name="expiration">The expiration date and time for
        /// the JWT.</param>
        /// <param name="securityKey">The security key used to sign
        /// the JWT.</param>
        /// <param name="issuer">The issuer of the JWT, generally the
        /// server API application's domain.</param>
        /// <param name="audience">The intended audience for the JWT,
        /// generally the client application's domain.</param>
        /// <param name="refreshToken">Refresh token to be included
        /// in the returned JwtResponse object.</param>
        /// <returns>A JwtResponse object containing the created JWT
        /// and provided refresh token.</returns>
```

```
        public JwtResponse GetJwtResponse(int userId, string userName,
            IEnumerable<string> userRoleNames, DateTime expiration,
            string securityKey, string issuer,
            string audience, string refreshToken);
        // public may not be written

        /// <summary>
        /// Returns a new generated refresh token, which is a secure,
        /// random string used to obtain new JWT without
        /// re-authenticating the user.
        /// </summary>
        /// <returns>
        /// A string representing the newly generated refresh token.
        /// </returns>
        public string GetRefreshToken();

        /// <summary>
        /// Extracts and returns a collection of claims from the
        /// specified access token (JWT) using the provided security key.
        /// </summary>
        /// <param name="jwt">The JWT containing encoded claims.</param>
        /// <param name="securityKey">The security key used to validate
        /// and decode the JWT.</param>
        /// <returns>
        /// A claim collection containing the claims extracted from the JWT.
        /// </returns>
        public IEnumerable<Claim> GetClaims(string jwt, string securityKey);
    }
}
```

9. Add a concrete class called JwtAuthService in the Services folder of the Authentication folder as below:

```
using CORE.Authentication.Models;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.IdentityModel.Tokens;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Security.Cryptography;
using System.Text;


namespace CORE.Authentication.Services
{
    /// <summary>
    /// Provides concrete implementations for JWT (JSON Web Token)
    /// based authentication operations, including generating JWT
    /// response with JWT (access token) and refresh token,
    /// generating refresh token, and extracting claims from JWT.
    /// This service is responsible for securely creating and
    /// validating JWT used in authentication flows.
    /// </summary>
    public class JwtAuthService : IJwtAuthService
    {
```

```
/// <summary>
/// Returns a JWT response including JWT (access token) and
/// refresh token.
/// </summary>
/// <param name="userId">The unique ID of the user.</param>
/// <param name="userName">The username of the user.</param>
/// <param name="userRoleNames">A collection of role names
/// assigned to the user.</param>
/// <param name="expiration">The expiration date and time for
/// the JWT.</param>
/// <param name="securityKey">The security key used to sign
/// the JWT.</param>
/// <param name="issuer">The issuer of the JWT, generally the
/// server API application's domain.</param>
/// <param name="audience">The intended audience for the JWT,
/// generally the client application's domain.</param>
/// <param name="refreshToken">Refresh token to be included
/// in the returned JwtResponse object.</param>
/// <returns>A JwtResponse object containing the created JWT
/// and provided refresh token.</returns>
public JwtResponse GetJwtResponse(int userId, string userName,
    IEnumerable<string> userRoleNames, DateTime expiration,
    string securityKey, string issuer,
    string audience, string refreshToken)
{
    // Create claims for user ID and username,
    // then add claims for each user role.
    var claims = new List<Claim>
    {
        new Claim("Id", userId.ToString()),
        // custom claim with key Id and value user ID
        new Claim(ClaimTypes.Name, userName)
    };
    foreach (var userRoleName in userRoleNames)
    {
        claims.Add(new Claim(ClaimTypes.Role, userRoleName));
    }

    // Create signing credentials using the provided security key
    // and 256-bit hash.
    var signingKey = new SymmetricSecurityKey(
        Encoding.UTF8.GetBytes(securityKey));
    var signingCredentials = new SigningCredentials(
        signingKey, SecurityAlgorithms.HmacSha256);

    // Build the JWT with claims, issuer, audience, and expiration.
    var jwtSecurityToken = new JwtSecurityToken(
        issuer, audience, claims, DateTime.Now, expiration,
        signingCredentials);
    var jwtSecurityTokenHandler = new JwtSecurityTokenHandler();

    // Serialize the JWT to a string.
    var jwt = jwtSecurityTokenHandler.WriteToken(jwtSecurityToken);

    // Return the JWT response with the serialized JWT value
    // and the refresh token parameter value.
    return new JwtResponse
    {
        Jwt = $"{JwtBearerDefaults.AuthenticationScheme} {jwt}",
        // JwtBearerDefaults.AuthenticationScheme: "Bearer"
        RefreshToken = refreshToken
    };
}

/// <summary>
/// Returns a new generated refresh token, which is a secure,
```

```
        /// random string used to obtain new JWT without
        /// re-authenticating the user.
        /// </summary>
        /// <returns>
        /// A string representing the newly generated refresh token.
        /// </returns>
        public string GetRefreshToken()
        {
            // Generate a cryptographically secure random
            // 32-byte refresh token.
            var bytes = new byte[32];
            using (var generator = RandomNumberGenerator.Create())
            {
                generator.GetBytes(bytes);
            }
            return Convert.ToBase64String(bytes);
        }

        /// <summary>
        /// Extracts and returns a collection of claims from the
        /// specified access token (JWT) using the provided security key.
        /// </summary>
        /// <param name="jwt">The JWT containing encoded claims.</param>
        /// <param name="securityKey">The security key used to validate
        /// and decode the JWT.</param>
        /// <returns>
        /// A claim collection containing the claims extracted from the JWT.
        /// </returns>
        public IEnumerable<Claim> GetClaims(string jwt, string securityKey)
        {
            // IEnumerable is an interface that the List class implements.
            // LINQ methods can also be used with IEnumerable.
            // An IEnumerable collection can be converted to a List collection
            // by invoking ToList method when needed, or ToArray method
            // to convert the collection to an array.

            // Remove the "Bearer" prefix if exists in the JWT.
            jwt = jwt.StartsWith(JwtBearerDefaults.AuthenticationScheme) ?
                jwt.Remove(0, JwtBearerDefaults.AuthenticationScheme.Length + 1) :
                jwt;

            // Prepare the signing key and validation parameters.
            var signingKey = new SymmetricSecurityKey(
                Encoding.UTF8.GetBytes(securityKey));
            var tokenValidationParameters = new TokenValidationParameters
            {
                ValidateIssuer = false,
                ValidateAudience = false,
                ValidateLifetime = false,
                ValidateIssuerSigningKey = true,
                IssuerSigningKey = signingKey
            };

            // Validate the JWT and extract claims then return the claims.
            var jwtSecurityTokenHandler = new JwtSecurityTokenHandler();
            SecurityToken securityToken;
            var principal = jwtSecurityTokenHandler.ValidateToken(
                jwt, tokenValidationParameters, out securityToken);
            // out and ref are used to pass arguments by reference,
            // allowing the method to modify the value of the argument
            // and return it to the caller through the variable (securityToken).
            // Mostly used with value types.
            return securityToken is null ? null : principal.Claims;
        }
    }
}
```

B) Users.APP Project for Authentication and Authorization

1. Add a new folder named Authentication in the Features folder of the Users.APP Project.

2. Add a class file named Token in the Authentication folder of the Features folder including TokenRequest and TokenHandler classes as below (TokenRequest and TokenHandler class files may also be created seperately):

```
using CORE.Authentication.Models;
using CORE.Authentication.Services;
using CORE.Services;
using MediatR;
using Microsoft.EntityFrameworkCore;
using Users.APP.Domain;

namespace Users.APP.Features.Authentication
{
    /// <summary>
    /// Represents a token request to obtain a JWT response
    /// including a JWT (access token) and refresh token.
    /// Inherits from JwtRequest and implements IRequest of
    /// type JwtResponse for MediatR pipeline integration.
    /// </summary>
    public class TokenRequest : JwtRequest, IRequest<JwtResponse>
    {
        // Inherits user credentials (UserName and Password)
        // with SecurityKey, Issuer and Audience properties
        // from the JwtRequest base class.
    }

    /// <summary>
    /// Handles a TokenRequest by validating user credentials and
    /// generating a JwtResponse including JWT and refresh token.
    /// </summary>
    public class TokenHandler : DbService<User>,
        IRequestHandler<TokenRequest, JwtResponse>
    {
        // The JWT authentication service that will provide JWT
        // operations in the methods of this class.
        private readonly IJwtAuthService _jwtAuthService;

        /// <summary>
        /// Initializes a new instance of the TokenHandler class.
        /// </summary>
        /// <param name="db">The injected application's user
        /// database context through the IoC Container.</param>
        /// <param name="jwtAuthService">The injected JWT
        /// authentication service instance through the
        /// IoC Container.</param>
        public TokenHandler(DbContext db,
            IJwtAuthService jwtAuthService) : base(db)
```

```
{
    _jwtAuthService = jwtAuthService;
}

/// <summary>
/// Returns a queryable collection of User entities with
/// their associated UserRole and Role navigation properties
/// eagerly included.
/// Overrides the base method to apply eager loading.
/// </summary>
/// <returns>
/// An IQueryable of type User with the UserRole and Role
/// navigation properties eagerly loaded.
/// </returns>
protected override IQueryable<User> DbQuery()
{
    return base.DbQuery()
        .Include(u => u.UserRoles)
        .ThenInclude(ur => ur.Role);
    // u: User entity delegate, ur: UserRole entity delegate
}

/// <summary>
/// Handles the token request by authenticating the user and
/// returning a JWT response including JWT and refresh token.
/// </summary>
/// <param name="request">The token request containing username,
/// password, security key, audience and issuer.</param>
/// <param name="cancellationToken">Asynchronous method's token
/// to cancel the operation.</param>
/// <returns>A JwtResponse including the JWT and refresh token
/// if successful, otherwise null.</returns>
public async Task<JwtResponse> Handle(TokenRequest request,
    CancellationToken cancellationToken)
{
    // Attempt to get the active user by user name and password
    var userEntity = await DbSingleAsync(
        u => u.UserName == request.UserName &&
        u.Password == request.Password && u.StatusId == 1,
        cancellationToken);
        // "Active" status ID value is 1 in the Statuses table.
        // Instead of u.StatusId == 1, u.Status.Title == "Active"
        // may also be written.
    // u: User entity delegate, ur: UserRole entity delegate.

    // If user entity is not found, return null
    if (userEntity is null)
        return null;

    // Generate refresh token and save it to the Users table
    // for the retrieved user entity with expiration date and time,
    // Also update the user entity's online status
    userEntity.RefreshToken = _jwtAuthService.GetRefreshToken();
    userEntity.RefreshTokenExpiration = DateTime.Now.AddDays(7);
    userEntity.IsOnline = true;
    // the refresh token will expire after 7 days from
    // DateTime.Now's execution value
    await DbUpdateAsync(userEntity, cancellationToken);


    // Return a JWT response according to the expiration
    // including the JWT and refresh token
    var expiration = DateTime.Now.AddMinutes(5);
```

```
            // the JWT will expire after 5 minutes from DateTime.Now's
            // execution value
            return _jwtAuthService.GetJwtResponse(userEntity.Id,
                userEntity.UserName,
                userEntity.UserRoles.Select(ur => ur.Role.Name),
                expiration, request.SecurityKey, request.Issuer,
                request.Audience, userEntity.RefreshToken);
        }
    }
}
```

3. Add a class file named RefreshToken in the Authentication folder of the Features folder including RefreshTokenRequest and RefreshTokenHandler classes as below (RefreshTokenRequest and RefreshTokenHandler class files may also be created seperately):

```
using CORE.Authentication.Models;
using CORE.Authentication.Services;
using CORE.Services;
using MediatR;
using Microsoft.EntityFrameworkCore;
using Users.APP.Domain;

namespace Users.APP.Features.Authentication
{
    /// <summary>
    /// Represents the request for returning a JWT response
    /// including JWT and refresh token.
    /// Inherits from the JwtRefreshRequest base class to have
    /// Jwt, RefreshToken, SecurityKey, Issuer and Audience
    /// properties, and implements IRequest of type JwtResponse
    /// for MediatR pipeline integration.
    /// </summary>
    public class RefreshTokenRequest : JwtRefreshRequest,
        IRequest<JwtResponse>
    {
    }

    /// <summary>
    /// Handles the logic for processing a refresh token request
    /// to return a JWT response.
    /// Validates if the refresh token is expired or not through
    /// a User entity query and returns a new JWT response including
    /// the new JWT and new refresh token if valid,
    /// otherwise returns null.
    /// </summary>
    public class RefreshTokenHandler : DbService<User>,
        IRequestHandler<RefreshTokenRequest, JwtResponse>
    {
        // The JWT authentication service that will provide JWT
        // operations in the methods of this class.
        private readonly IJwtAuthService _jwtAuthService;

        /// <summary>
        /// Initializes a new instance of the RefreshTokenHandler
        /// class.
        /// </summary>
```

```
/// <param name="db">The injected application's user
/// database context through the IoC Container.</param>
/// <param name="jwtAuthService">The injected JWT
/// authentication service instance through the
/// IoC Container.</param>

public RefreshTokenHandler(DbContext db,
    IJwtAuthService jwtAuthService) : base(db)
{
    _jwtAuthService = jwtAuthService;
}

/// <summary>
/// Returns a queryable collection of User entities with
/// their associated UserRole and Role navigation properties
/// eagerly included.
/// Overrides the base method to apply eager loading.
/// </summary>
/// <returns>
/// An IQueryable of type User with the UserRole and Role
/// navigation properties eagerly loaded.
/// </returns>
protected override IQueryable<User> DbQuery()
{
    return base.DbQuery()
        .Include(u => u.UserRoles)
        .ThenInclude(ur => ur.Role);
    // u: User entity delegate, ur: UserRole entity delegate
}

/// <summary>
/// Handles the refresh token logic: verifies the user with
/// refresh token expiration, then returns a new JWT response
/// including the JWT and refresh token if verified.
/// Otherwise returns null.
/// </summary>
/// <param name="request">The refresh token request object
/// containing the JWT, refresh token, security key, issuer
/// and audience.</param>
/// <param name="cancellationToken">Asynchronous method's token
/// to cancel the operation.</param>
/// <returns>A JWT response containing the result of the
/// operation.</returns>
public async Task<JwtResponse> Handle(RefreshTokenRequest request,
    CancellationToken cancellationToken)
{
    // Extract the user's claims from request's expired JWT
    // and security key
    var claims = _jwtAuthService.GetClaims(request.Jwt,
        request.SecurityKey);

    // Extract the user ID from claims
    var userId = Convert.ToInt32(claims
        .SingleOrDefault(claim => claim.Type == "Id").Value);

    // Find user entity in the Users database table that matches
    // the ID and has a non expired refresh token
    var userEntity = await DbSingleAsync(user =>
        user.Id == userId &&
        user.RefreshToken == request.RefreshToken &&
        user.RefreshTokenExpiration >= DateTime.Now,
        cancellationToken);
```

```
            // If user entity is not found, return null
            if (userEntity is null)
                return null;

            // Generate a new refresh token (for added security)
            userEntity.RefreshToken = _jwtAuthService.GetRefreshToken();

            // Optional: Enable sliding expiration for the refresh token
            // userEntity.RefreshTokenExpiration = DateTime.Now.AddDays(7);

            // Save the updated user entity state to the database
            await DbUpdateAsync(userEntity, cancellationToken);

            // Return a JWT response according to the expiration including
            // the new JWT and new refresh token
            var expiration = DateTime.Now.AddMinutes(5);
            return _jwtAuthService.GetJwtResponse(userEntity.Id,
                userEntity.UserName,
                userEntity.UserRoles.Select(ur => ur.Role.Name),
                expiration, request.SecurityKey, request.Issuer,
                request.Audience, userEntity.RefreshToken);
        }
    }
}
```

C) Users.API Project for Authentication and Authorization

Authentication:

1. Add the following sections in appsettings.json under the ConnectionStrings section.

```
"Issuer": "https://usersmicroservices.com",
"Audience": "https://usersmicroservices.com",
"TokenMessage": {
    "NotFound": "User not found!",
    "BadRequest": "Invalid token request!"
},
```

2. In the IoC Container of Program.cs add the sections marked as Authentication under the MediatR dependency registration as below:

```
// --------------
// Authentication
// --------------
// Registers the JwtAuthService as a scoped service for the
// IJwtAuthService interface.
// Lifetime: Scoped (a new instance is created for each
// HTTP request).
// Usage: All dependencies requesting IJwtAuthService will
// receive the same JwtAuthService instance.
builder.Services.AddScoped<IJwtAuthService, JwtAuthService>();



// --------------
// Authentication
// --------------
// For getting the value for the key SecurityKey from
// appsettings.json in any class injected with IConfiguration
// instance to be used for JWT.
builder.Configuration["SecurityKey"] =
    "users_microservices_security_key_2026=";
    // must be minimum 256 bits
// Enable JWT Bearer authentication as the default scheme.
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(config =>
    {
        // Define rules for validating JWT.
        config.TokenValidationParameters = new TokenValidationParameters
        {
            // Use the builder configuration's security key to create
            // a new symmetric security key for verifying the JWT's signature.
            IssuerSigningKey = new SymmetricSecurityKey(
                Encoding.UTF8.GetBytes(
                    builder.Configuration["SecurityKey"] ?? string.Empty)),

            ValidIssuer = builder.Configuration["Issuer"],
            // get Issuer section's value from appsettings.json
            ValidAudience = builder.Configuration["Audience"],
            // get Audience section's value from appsettings.json

            // These flags ensure the validation of the JWT.
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateIssuerSigningKey = true,
            ValidateLifetime = true
        };
    });



// --------------------------
// Authentication and Swagger
// --------------------------
// Configure Swagger/OpenAPI documentation, including JWT authentication support
// in the UI.
builder.Services.AddSwaggerGen(c =>
{
    // Define the basic information for your API.
    c.SwaggerDoc("v1", new OpenApiInfo
    {
        Title = "Users API",
        Version = "v1"
    });

    // Add the JWT Bearer scheme to the Swagger UI so JWT can be tested in requests.
    c.AddSecurityDefinition(JwtBearerDefaults.AuthenticationScheme,
```

```
        new OpenApiSecurityScheme
        {
            Name = "Authorization",
            Type = SecuritySchemeType.ApiKey,
            Scheme = JwtBearerDefaults.AuthenticationScheme,
            BearerFormat = "JWT",
            In = ParameterLocation.Header,
            Description = """
                JWT Authorization header using the Bearer scheme.
                Enter your JWT as: "Bearer jwt"
                Example: "Bearer a1b2c3"
            """
    });

    // Add the security requirement globally so all endpoints are secured unless
    // specified otherwise.
    c.AddSecurityRequirement(new OpenApiSecurityRequirement
    {
        {
            new OpenApiSecurityScheme
            {
                Reference = new OpenApiReference
                {
                    Type = ReferenceType.SecurityScheme,
                    Id = JwtBearerDefaults.AuthenticationScheme
                }
            },
            Array.Empty<string>()
        }
    });
});



// --------------------------
// Authentication and Swagger
// --------------------------
// Configure the HTTP request pipeline.
// Way 1: Enable Swagger for only the development environment.
//if (app.Environment.IsDevelopment())
//{
//    app.UseSwagger();
//    app.UseSwaggerUI();
//}
// Way 2: Enable Swagger for both development and production environments.
app.UseSwagger();
app.UseSwaggerUI();


// --------------
// Authentication
// --------------
// Enable authentication middleware so that [Authorize] works.
app.UseAuthentication();
```

Note: You need to use the same security key (users_microservices_security_key_2026=), audience and issuer implemented in the Program.cs in your other project too for JWT Authentication to work.

3. Right-click Users.API Controllers folder then Add -> Controller -> Common -> API -> API Controller - Empty to create the TokensController, implement the Mediator and Configuration injections then implement the Token and RefreshToken actions as below:

```
using MediatR;
using Microsoft.AspNetCore.Mvc;
using Users.APP.Features.Authentication;

namespace Users.API.Controllers
{
    /// <summary>
    /// API controller for handling token-related operations
    /// such as generating new JWT and refreshing JWT.
    /// </summary>
    [Route("api/[controller]")] // Sets the base route for
                                // this controller to "api/tokens"
                                // (controller name is replaced at
                                // runtime).
    [ApiController] // Indicates that this class is an API controller
                    // and enables automatic model validation and
                    // binding.
    public class TokensController : ControllerBase
    {
        private readonly IMediator _mediator;
        // instance of type implementing IMediator will be injected
        // to this variable in the constructor
        private readonly IConfiguration _configuration;
        // instance of type implementing IConfiguration will be
        // injected to this variable in the constructor

        /// <summary>
        /// Injects the mediator instance and application's configuration
        /// instance for getting configuration values from such as
        /// appsettings.json.
        /// </summary>
        /// <param name="mediator">The mediator instance for
        /// sending requests.</param>
        /// <param name="configuration">The application's configuration
        /// settings instance for getting configuration values.</param>
        public TokensController(IMediator mediator,
            IConfiguration configuration)
        {
            _mediator = mediator;
            _configuration = configuration;
        }


        /// <summary>
        /// Handles HTTP POST requests to generate a new JWT and
        /// refresh token for a user.
        /// </summary>
        /// <param name="request">The token request containing
        /// user credentials (user name and password).</param>
        /// <returns>
        /// An IActionResult containing the JWT response if
        /// authentication operation is successful, or an error message
        /// if authentication operation fails or the request is invalid.
        /// </returns>
```

```
[HttpPost] // Specifies that this action responds to
           // HTTP POST requests.
[Route("~/api/[action]")] // Overrides the controller's base route.
                          // The route becomes "api/Token"
                          // (action name is replaced at runtime).
public async Task<IActionResult> Token(TokenRequest request)
{
    request.SecurityKey = _configuration["SecurityKey"];
    // get the SecurityKey section value from the previously added
    // section in Program.cs
    request.Audience = _configuration["Audience"];
    // get the Audience section value from appsettings.json
    request.Issuer = _configuration["Issuer"];
    // get the Issuer section value from appsettings.json
    if (ModelState.IsValid)
    {
        var response = await _mediator.Send(request);
        if (response is not null)
            return Ok(response);
        return NotFound(_configuration["TokenMessage:NotFound"]);
        // return the NotFound section value of the TokenMessage section
        // from appsettings.json as a HTTP 404 NotFound response
    }
    return BadRequest(_configuration["TokenMessage:BadRequest"]);
    // return the BadRequest section value of the TokenMessage section
    // from appsettings.json as a HTTP 400 BadRequest response
}

/// <summary>
/// Handles HTTP POST requests to refresh the JWT and the refresh token
/// for a user.
/// </summary>
/// <param name="request">The refresh token request containing the
/// previously generated expired JWT and refresh token.</param>
/// <returns>
/// An IActionResult containing the new JWT response if the refresh
/// operation is successful, or an error message if the refresh operation
/// fails or the request is invalid.
/// </returns>
[HttpPost] // Specifies that this action responds to
           // HTTP POST requests.
[Route("~/api/[action]")] // Overrides the controller's base route.
                          // The route becomes "api/RefreshToken"
                          // (action name is replaced at runtime).
public async Task<IActionResult> RefreshToken(RefreshTokenRequest request)
{
    request.SecurityKey = _configuration["SecurityKey"];
    // get the SecurityKey section value from the previously added
    // section in Program.cs
    request.Audience = _configuration["Audience"];
    // get the Audience section value from appsettings.json
    request.Issuer = _configuration["Issuer"];
    // get the Issuer section value from appsettings.json
    if (ModelState.IsValid)
    {
        var response = await _mediator.Send(request);
        if (response is not null)
            return Ok(response);
        return NotFound(_configuration["TokenMessage:NotFound"]);
        // return the NotFound section value of the TokenMessage section
        // from appsettings.json as a HTTP 404 NotFound response
    }
    return BadRequest(_configuration["TokenMessage:BadRequest"]);
    // return the BadRequest section value of the TokenMessage section
    // from appsettings.json as a HTTP 400 BadRequest response
}
```

```
    }
}
```

Notes:

- JWT (JSON Web Token) existance can be checked by User.Identity.IsAuthenticated in controller actions returning the result of whether the user is authenticated or not.

- The auhenticated user's user name can be reached by User.Identity.Name.

- The authenticated user's role claim value can be checked by User.IsInRole method which gets the role name string as parameter, or the role claim can be accessed by User.Claims.SingleOrDefault method which gets the predicate checking claim type equals role claim type as parameter.

- Examples:

a) if (User.IsInRole("Admin")) { ... }

b) Claim roleClaim = User.Claims.SingleOrDefault(
claim => claim.Type == ClaimTypes.Role);

c) if (roleClaim is not null && roleClaim.Value == "User") { ... }

d) Custom claim types' values such as Id can be accessed by:

```
int id = Convert.ToInt32(User.Claims?.Single(claim => claim.Type == "Id").Value);
```

Authorization:

1. Add the [Authorize(Roles = "Admin")] attribute for Admin role on top of the RolesController class so that all of the actions can be executed by only authenticated users with role Admin as below:

```
[Authorize(Roles = "Admin")]
// Only authenticated users with role Admin can execute all
// of the actions of this controller.
public class RolesController : ControllerBase
```

Notes:

- The [Authorize] attribute in ASP.NET Core is used to control access to a controller's actions by requiring that the user is authenticated and, optionally, meets certain authorization requirements.

- When you decorate a controller or action with [Authorize], only authenticated users who provides the Authentication Cookie or Json Web Token (used in Web APIs) can access it. Unauthenticated requests will receive a 401 Unauthorized response.

- You can specify roles or policies to further restrict access. For example, [Authorize(Roles = "Admin")]
  allows only authenticated users in the "Admin" role.

- You can apply [Authorize] at the controller level (to protect all actions) or at the action level.

Examples:

a) [Authorize]
   Any authenticated user can access.

b) [Authorize(Roles = "User")]
Only authenticated users with role "User" can access.

c) [Authorize(Roles = "Admin,User")]
Only authenticated users with the "Admin" or "User" role can access.

d) [AllowAnonymous]
Can be used to override [Authorize] at the action level and allows public access, therefore gives permission to everyone for executing specific actions.

2. Add the [Authorize] attribute on top of the StatusesController class.

   For example, if the Get action is wanted to be executed by authorized and unauthorized users (eveyone), [AllowAnonymous] attribute can be defined.

   The Get by id action can be executed by only authenticated users since Authorize is defined at controller level.

   Add [Authorize(Roles = "Admin")] attribute on top of the Post, Put and Delete actions so that only authenticated users with role Admin can execute these actions.

3. Add the [Authorize] attribute on top of the Get action of the UsersController class so that authenticated users with all roles can execute the Get action.

   Then add [Authorize] attribute on top of the Get by id, Put and Delete actions to allow only authenticated users to execute. These actions also include if the user is trying to make an operation on his/her own account through IsOwnAccount method at the bottom of the UsersController class, or for users with role Admin if the authenticated user is in Admin role check.

IsOwnAccount Method:

```
/// <summary>
/// For regular users can only make operations on their own accounts.
/// </summary>
/// <param name="id">User ID.</param>
/// <returns>bool</returns>
bool IsOwnAccount(int id) // private is default if not written
{
    // getting the authenticated user ID value for the claim type
    // "Id" from the user's claims and checking if it matches
    // the provided id parameter of a user
    return id.ToString() == (User.Claims.SingleOrDefault(
        claim => claim.Type == "Id")?.Value ?? string.Empty);
}
```

Get by id Action:

```
// GET: api/Users/5
[HttpGet("{id}")]
[Authorize] // Only authenticated users can execute this action.
public async Task<IActionResult> Get(int id)
{
    try
    {
        // Check if the user is in Admin role or trying to make
        // the operation on his/her own account.
        if (!IsOwnAccount(id) && !User.IsInRole("Admin"))
            return BadRequest("You are not authorized for this operation!");

        // Send a query request to get query response
        var response = await _mediator.Send(new UserQueryRequest());
        // Find the item with the given id
        var item = await response.SingleOrDefaultAsync(r => r.Id == id);
        // If item found, return it with 200 OK
        if (item is not null)
            return Ok(item);
        // If item not found, return 204 No Content
        return NoContent();
    }
    catch (Exception exception)
    {
        // Log the exception
        _logger.LogError("UsersGetById Exception: " + exception.Message);
        // Return 500 Internal Server Error with an error command response
        // with message
        return StatusCode(StatusCodes.Status500InternalServerError,
            new CommandResponse(false, "An exception occured during UsersGetById."));
    }
}
```

Put Action:

```
// PUT: api/Users
[HttpPut]
[Authorize] // Only authenticated users can execute this action.
public async Task<IActionResult> Put(UserUpdateRequest request)
```

```
{
    try
    {
        // Check if the user is in Admin role or trying to make
        // the operation on his/her own account.
        if (!IsOwnAccount(request.Id) && !User.IsInRole("Admin"))
            return BadRequest("You are not authorized for this operation!");

        // Check if the request model is valid through data annotations
        if (ModelState.IsValid)
        {
            // Send the update request
            var response = await _mediator.Send(request);
            // If update is successful, return 200 OK with success command response
            if (response.IsSuccessful)
            {
                //return NoContent();
                return Ok(response);
            }
            // If update failed, add error command response message to model state
            ModelState.AddModelError("UsersPut", response.Message);
        }
        // Return 400 Bad Request with all data annotation validation
        // error messages and the error command response message if added
        // seperated by |
        return BadRequest(new CommandResponse(false, string.Join("|",
            ModelState.Values.SelectMany(v => v.Errors).Select(e => e.ErrorMessage))));
    }
    catch (Exception exception)
    {
        // Log the exception
        _logger.LogError("UsersPut Exception: " + exception.Message);
        // Return 500 Internal Server Error with an error command response
        // with message
        return StatusCode(StatusCodes.Status500InternalServerError,
            new CommandResponse(false, "An exception occured during UsersPut."));
    }
}
```

Delete Action:

```
// DELETE: api/Users/5
[HttpDelete("{id}")]
[Authorize] // Only authenticated users can execute this action.
public async Task<IActionResult> Delete(int id)
{
    try
    {
        // Check if the user is in Admin role or trying to make
        // the operation on his/her own account.
        if (!IsOwnAccount(id) && !User.IsInRole("Admin"))
            return BadRequest("You are not authorized for this operation!");

        // Send the delete request
        var response = await _mediator.Send(new UserRemoveRequest() { Id = id });
        // If delete is successful, return 200 OK with success command response
        if (response.IsSuccessful)
        {
            //return NoContent();
            return Ok(response);
        }
        // If delete failed, add error command response message to model state
        ModelState.AddModelError("UsersDelete", response.Message);
        // Return 400 Bad Request with the error command response message
        return BadRequest(new CommandResponse(false, string.Join("|",
            ModelState.Values.SelectMany(v => v.Errors).Select(e => e.ErrorMessage))));
```

```
    }
    catch (Exception exception)
    {
        // Log the exception
        _logger.LogError("UsersDelete Exception: " + exception.Message);
        // Return 500 Internal Server Error with an error command response
        // with message
        return StatusCode(StatusCodes.Status500InternalServerError,
            new CommandResponse(false, "An exception occured during UsersDelete."));
    }
}
```

Finally add [Authorize(Roles = "Admin")] attribute on top of the Post action so that authenticated users with role Admin can execute this action.

Cross-Origin Resource Sharing (CORS):

1. To increase the security of APIs for the Production Environment and to use the optionally developed front-end application with the backend APIs, CORS configuration can be added in the Program.cs of the Users.API Project as below:

```
// ------------------------------------------------------------
// CORS (Cross-Origin Resource Sharing) for Production Environment
// ------------------------------------------------------------
// Registers and configures CORS services for the application.
// CORS is a security feature implemented by browsers to restrict
// cross-origin HTTP requests initiated from scripts running in the browser.
// By default, web applications are not allowed to make requests to a domain
// different from the one that served the web page.
// The configuration below adds a default CORS policy that allows requests
// from any origin, with any HTTP header, and any HTTP method.
// This is useful during development or for public APIs, but should be
// restricted in production environments to specific origins for better security.
// Usage:
// - The policy is applied globally if app.UseCors() is called without parameters
// in the middleware pipeline.
// - To restrict CORS, replace AllowAnyOrigin(), AllowAnyHeader(), and
// AllowAnyMethod() with more specific rules.
builder.Services.AddCors(options =>
{
    options.AddDefaultPolicy(builder => builder
        .AllowAnyOrigin()   // Allows requests from any domain.
        .AllowAnyHeader()   // Allows any HTTP headers in the request.
        .AllowAnyMethod()); // Allows any HTTP method (GET, POST, PUT, DELETE, etc.).
});


// ------------------------------------------------------------
// CORS (Cross-Origin Resource Sharing) for Production Environment
// ------------------------------------------------------------
app.UseCors();
```

V) Running Multiple API Projects

In order to run multiple API Projects together in Visual Studio, right-click any API Project, click “Configure Startup Projects”, click “Multiple startup projects” then select Action as Start for the API Projects to run in the right window.

For Rider, you need to create a multi-launch configuration as below from JetBrains Rider documentation:

https://www.jetbrains.com/help/rider/Run_Debug_Multiple.html

- Select the desired projects in the solution explorer.
- Right-click the selection and choose Run Multiple Projects.
- JetBrains Rider will create run configurations for each project as well as a new Multi-Launch configuration. will be created and displayed in the Run/Debug Configurations dialog.
- By default, when you start a Multi-Launch configuration, the start signals are sent to all configurations and tasks immediately.
- If you need configurations and tasks to start in a specific order, first rearrange the configurations using the buttons on the left, then use the selectors in the When to launch column to change launch conditions of configurations and tasks.
- Often when you need to launch multiple projects, you may want to debug only some of them, and others are started for auxiliary functions. To avoid attaching the debugger to all projects, select checkboxes in the Do not debug column next to the projects that should not be debugged.
- Click OK to save the configuration.
- The newly created Multi-Launch configuration will be displayed as the default configuration in the toolbar selector, and you will be able to run or debug it when necessary.

VI) Gateway.API Project

A Gateway API Project may be developed to consume multiple API projects with different IP addresses and port numbers from a single IP address and port number.

The configuration in the Gateway API Project directs the requests to the different APIs.

A gateway helps to develop client applications using a single IP address and port number to consume APIs.

1. Create a new project under the solution named “Gateway.API”, search for web api and select “ASP.NET Core Web API” template, select .NET 8 as the Framework, Authentication type as “None”, check “Configure for HTTPS”, do not check “Enable container support”, do not check “Enable OpenAPI support”, do not check “Do not use top-level statements”, do not check “Use controllers” and do not check “Enlist in .NET Aspire orchestration”.

2. Right click on the Gateway.API project then click Manage NuGet Packages. Under the Browse tab search for Ocelot and install the latest version.

3. Create the ocelot.json file in the Gateway.API project by right-clicking the Gateway.API Project then clicking Add -> New Item as below:

   Note: You can get the Host and Port of DownstreamHostAndPorts section from the launchSettings.json files which are under the Properties folder of the API Projects. If you are running your solution with https, look for the https profile and use the first url defined at applicationUrl section which is for our solution "localhost:7124" for Users.API. If you are running your solution with http, look for the http profile and use the url defined at applicationUrl section which is for our solution "localhost:5223" for Users.API. Now you can reach all of the users endpoints defined in the ocelot.json file through this gateway.

```
{
  "Routes": [
    {
      // for testing the gateway to send a request to the
      // WeatherForecastController of Users.API
      "DownstreamPathTemplate": "/weatherforecast",
      "DownstreamScheme": "https",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
          "Port": 7124
        }
      ],
      "UpstreamPathTemplate": "/weatherforecast",
      "UpstreamHttpMethod": [ "Get" ]
    },
    {
      "DownstreamPathTemplate": "/api/users/{id}",
      "DownstreamScheme": "https",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
          "Port": 7124
        }
      ],
      "UpstreamPathTemplate": "/api/users/{id}",
      "UpstreamHttpMethod": [ "Get", "Post", "Put", "Delete" ]
    },
    {
      "DownstreamPathTemplate": "/api/roles/{id}",
      "DownstreamScheme": "https",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
          "Port": 7124
        }
      ],
      "UpstreamPathTemplate": "/api/roles/{id}",
      "UpstreamHttpMethod": [ "Get", "Post", "Put", "Delete" ]
    },
    {
      "DownstreamPathTemplate": "/api/statuses/{id}",
      "DownstreamScheme": "https",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
          "Port": 7124
        }
      ],
      "UpstreamPathTemplate": "/api/statuses/{id}",
      "UpstreamHttpMethod": [ "Get", "Post", "Put", "Delete" ]
    }
  ]
}
```

4. Replace all of the code in the Program.cs file of the Gateway.API Project with the code below:

```
using Ocelot.DependencyInjection;
using Ocelot.Middleware;

var builder = WebApplication.CreateBuilder(args);

// Add services to the IoC container.
// Load the Ocelot configuration file, which defines routes and
// downstream services.
builder.Configuration.AddJsonFile("ocelot.json");

// Register Ocelot services into the DI container.
builder.Services.AddOcelot();

var app = builder.Build();

// Enables Ocelot's middleware to intercept and forward incoming
// HTTP requests.
await app.UseOcelot();

app.UseHttpsRedirection();

var summaries = new[]
{
    "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy",
    "Hot", "Sweltering", "Scorching"
};

app.MapGet("/weatherforecast", () =>
{
    var forecast = Enumerable.Range(1, 5).Select(index =>
        new WeatherForecast
        (
            DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
            Random.Shared.Next(-20, 55),
            summaries[Random.Shared.Next(summaries.Length)]
        ))
        .ToArray();
    return forecast;
});

app.Run();

internal record WeatherForecast(DateOnly Date, int TemperatureC,
    string? Summary)
{
    public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);
}
```

5. Set Gateway.API and Users.API as multiple startup projects.

   After you run your solution and if you get a successful response for weather forecast, your gateway is working properly. You may also test the statuses GET response, since the GET action of the Statuses controller was defined allow anonymous, by entering “/api/statuses” at the end of Gateway.API Project’s IP address and port number in the address bar of your browser (Example for this solution: https://localhost:7123/api/statuses).

Now you can use the controller actions of your API projects through your client applications or Postman to consume your APIs by only using the Gateway.API Project's IP address and port number then the route of your controller actions such as: https://localhost:7123/api/users and https://localhost:7123/api/roles.
The IP address and port number of your Gateway.API Project can be found in the launchSettings.json file in the Properties folder which is https://localhost:7123 for the https profile and http://localhost:5225 for the http profile in our solution.

You may optionally develop a client application with React, Angular, ASP.NET Core MVC, HTMX, etc. if you want.

VII) Additional Information

1. The complete implementation of the Program.cs in the Users.API Project:

```
using CORE.Authentication.Services;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.EntityFrameworkCore;
using Microsoft.IdentityModel.Tokens;
using Microsoft.OpenApi.Models;
using System.Text;
using Users.APP.Domain;

// Create a builder for the Web API and initialize
// configuration, logging, etc.
var builder = WebApplication.CreateBuilder(args);

// ------------------------------------------------------
// Add services to the IoC (Inversion of Control) container
// for Dependency Injections.
// ------------------------------------------------------
// Register the application's DbContext dependency injection
// by sending UsersDb instances to the injected service class
// constructors' DbContext or UsersDb parameter, using SQLite
// with the connection string from appsettings.json.
builder.Services.AddDbContext<DbContext, UsersDb>(
    options => options.UseSqlite(
        builder.Configuration.GetConnectionString(nameof(UsersDb))));
// nameof(UsersDb) = "UsersDb" which is the class name.

// Way 1:
// Registers MediatR services with the dependency injection
// container.
// MediatR is a popular .NET library that implements the
// mediator pattern, enabling decoupled communication
// between components by sending requests (commands, queries,
// events) to handlers without direct dependencies.
// The configuration below scans the assembly containing the
```

```
// 'UsersDb' type for any classes that implement MediatR handler
// interfaces (such as IRequestHandler, INotificationHandler,
// etc.).
// This allows automatic discovery and registration of all
// MediatR handlers in the specified assembly.
// As a result, you can inject the IMediator interface into
// controllers and use it to send requests or publish notifications,
// which will be routed to the appropriate handlers.
//builder.Services.AddMediatR(
//  config => config.RegisterServicesFromAssembly(
//      typeof(UsersDb).Assembly));
// Way 2:
// Iterates through all assemblies currently loaded in the
// application's app domain.
// For each assembly, registers all MediatR handlers (such as
// IRequestHandler, INotificationHandler, etc.) found within
// that assembly with the dependency injection container.
// This enables MediatR to automatically discover and wire up
// handlers from any loaded assembly, allowing for modular handler
// organization and dynamic handler loading (e.g., from plugins
// or feature assemblies).
// Note: Registering handlers from all assemblies can be useful
// in large or modular applications, but may introduce duplicate
// registrations or performance overhead if not managed carefully.
foreach (var assembly in AppDomain.CurrentDomain.GetAssemblies())
{
    builder.Services.AddMediatR(
        config => config.RegisterServicesFromAssemblies(assembly));
}

// --------------
// Authentication
// --------------
// Registers the JwtAuthService as a scoped service for the
// IJwtAuthService interface.
// Lifetime: Scoped (a new instance is created for each
// HTTP request).
// Usage: All dependencies requesting IJwtAuthService will
// receive the same JwtAuthService instance.
builder.Services.AddScoped<IJwtAuthService, JwtAuthService>();

// --------------
// Authentication
// --------------
// For getting the value for the key SecurityKey from
// appsettings.json in any class injected with IConfiguration
// instance to be used for JWT.
builder.Configuration["SecurityKey"] =
    "users_microservices_security_key_2026=";
    // must be minimum 256 bits
// Enable JWT Bearer authentication as the default scheme.
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(config =>
    {
        // Define rules for validating JWT.
        config.TokenValidationParameters = new TokenValidationParameters
        {
            // Use the builder configuration's security key to create
            // a new symmetric security key for verifying the JWT's signature.
            IssuerSigningKey = new SymmetricSecurityKey(
                Encoding.UTF8.GetBytes(
                    builder.Configuration["SecurityKey"] ?? string.Empty)),

            ValidIssuer = builder.Configuration["Issuer"],
            // get Issuer section's value from appsettings.json
            ValidAudience = builder.Configuration["Audience"],
```

```
            // get Audience section's value from appsettings.json

            // These flags ensure the validation of the JWT.
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateIssuerSigningKey = true,
            ValidateLifetime = true
        };
    });

// --------------------------
// Authentication and Swagger
// --------------------------
// Configure Swagger/OpenAPI documentation, including JWT authentication support
// in the UI.
builder.Services.AddSwaggerGen(c =>
{
    // Define the basic information for your API.
    c.SwaggerDoc("v1", new OpenApiInfo
    {
        Title = "Users API",
        Version = "v1"
    });

    // Add the JWT Bearer scheme to the Swagger UI so JWT can be tested in requests.
    c.AddSecurityDefinition(JwtBearerDefaults.AuthenticationScheme,
        new OpenApiSecurityScheme
        {
            Name = "Authorization",
            Type = SecuritySchemeType.ApiKey,
            Scheme = JwtBearerDefaults.AuthenticationScheme,
            BearerFormat = "JWT",
            In = ParameterLocation.Header,
            Description = """
                JWT Authorization header using the Bearer scheme.
                Enter your JWT as: "Bearer jwt"
                Example: "Bearer a1b2c3"
            """
    });

    // Add the security requirement globally so all endpoints are secured unless
    // specified otherwise.
    c.AddSecurityRequirement(new OpenApiSecurityRequirement
    {
        {
            new OpenApiSecurityScheme
            {
                Reference = new OpenApiReference
                {
                    Type = ReferenceType.SecurityScheme,
                    Id = JwtBearerDefaults.AuthenticationScheme
                }
            },
            Array.Empty<string>()
        }
    });
});

// Add support for controllers.
builder.Services.AddControllers();

// Add Swagger support.
// Learn more about configuring Swagger/OpenAPI
// at https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
```

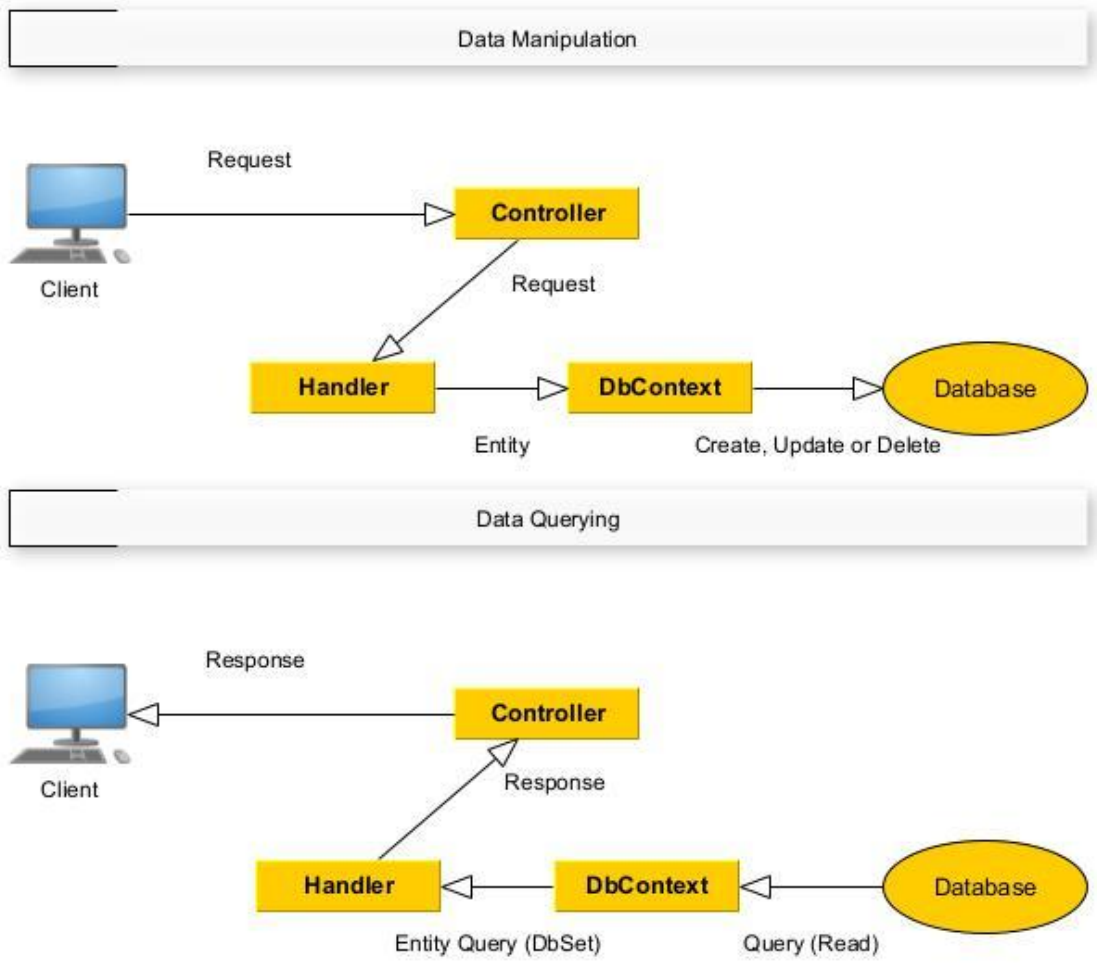
```
// ------------------------------------------------------------
// CORS (Cross-Origin Resource Sharing) for Production Environment
// ------------------------------------------------------------
// Registers and configures CORS services for the application.
// CORS is a security feature implemented by browsers to restrict
// cross-origin HTTP requests initiated from scripts running in the browser.
// By default, web applications are not allowed to make requests to a domain
// different from the one that served the web page.
// The configuration below adds a default CORS policy that allows requests
// from any origin, with any HTTP header, and any HTTP method.
// This is useful during development or for public APIs, but should be
// restricted in production environments to specific origins for better security.
// Usage:
// - The policy is applied globally if app.UseCors() is called without parameters
// in the middleware pipeline.
// - To restrict CORS, replace AllowAnyOrigin(), AllowAnyHeader(), and
// AllowAnyMethod() with more specific rules.
builder.Services.AddCors(options =>
{
    options.AddDefaultPolicy(builder => builder
        .AllowAnyOrigin()   // Allows requests from any domain.
        .AllowAnyHeader()   // Allows any HTTP headers in the request.
        .AllowAnyMethod()); // Allows any HTTP method (GET, POST, PUT, DELETE, etc.).
});

// Build the application.
var app = builder.Build();

// ---------------------------
// Authentication and Swagger
// ---------------------------
// Configure the HTTP request pipeline.
// Way 1: Enable Swagger for only the development environment.
//if (app.Environment.IsDevelopment())
//{
//    app.UseSwagger();
//    app.UseSwaggerUI();
//}
// Way 2: Enable Swagger for both development and production environments.
app.UseSwagger();
app.UseSwaggerUI();

// Redirect HTTP requests to HTTPS.
app.UseHttpsRedirection();

// --------------
// Authentication
// --------------
// Enable authentication middleware so that [Authorize] works.
app.UseAuthentication();

// Enable authorization middleware.
app.UseAuthorization();

// Maps controller endpoints to handle incoming HTTP requests.
app.MapControllers();

// ------------------------------------------------------------------
// CORS (Cross-Origin Resource Sharing) for Production Environment
// ------------------------------------------------------------------
app.UseCors();

// Runs the application and starts listening for incoming HTTP requests.
app.Run();
```

2. A simplified version of the Clean Architecture and Repository Pattern that include the basic concepts and structures are applied to the project. More information about Clean Architecture and other architectures can be found at:

   https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures

3. Domain is for data access from a database, features are for business logic and API is for presentation. Mediator and CQRS (Command Query Response Segregation) Patterns are also applied to this project.

4. Web API data querying and manipulation:



5. Controller classes have action methods that handle the incoming HTTP get, post, put and delete requests, generally interact with entity data in the database through handler classes using model (DTO: Data Transfer Object) classes and generally return JSON (JavaScript Object Notation) or XML serialized model class data.

6. ActionResult inheritance:

   IActionResult : general return type of actions in a controller
   |
   ActionResult: base class that implements IActionResult
   |
   OkObjectResult (returned by Ok method) - NoContentObjectResult (returned by NoContent method) - BadRequestObjectResult (returned by BadRequest method) - etc.

7. The default route to send the request to controller actions is:

   controller/id?

   where id value is optional.

   The action will be executed by the HTTP method of the request.

   For example:

   If GET is used as the HTTP method, Get action will be executed.

   If GET HTTP method is used with an id value, Get action with the id parameter will be executed.

   If POST HTTP method is used, Post action will be executed.

   If PUT HTTP method is used, Put action will be executed.

   If DELETE HTTP method is used with an id value, Delete action with the id parameter will be executed.

8. Getters and Setters in C#:

```
public abstract class Record
{
    // Way 1:
    // Field to store the record's ID.
    //private int id; // variable, field

    // Method to get the record's ID.
    //public int GetId() // function, method, behavior
    //{
    //    return id;
    //}

    // Method to set the record's ID.
    //public void SetId(int id)
    //{
    //    this.id = id;
    //}
```

```
    // Way 2:
    public int Id { get; set; }
}
```

9. Enums in C#:

```
namespace Users.APP.Domain
{
    // Enums are used for predefined values and help not to remember or
    // check each element's value when being used.
    // Getting the value of an enum element:
    // int womanValue = (int)Genders.Woman;
    // Getting the text of an enum element:
    // string manText = Genders.Man.ToString();
    // Way 1:
    //public enum Genders
    //{
    //    Woman = 1, // will start from 1
    //    Man = 2 // will get the value 2 and other elements will continue
    //            // to get the next values after 2
    //}


    // Way 2:
    //public enum Genders
    //{
    //    Woman = 1, // will start from 1
    //    Man // will automatically get the next value 2 and other elements
    //        // will continue to get the next values after 2
    //}


    // Way 3:
    public enum Genders
    {
        Woman, // if no assignment, will start from 0
        Man // will automatically get the next value 1 and other elements
            // will continue to get the next values after 1
    }
}
```

10. Handler classes are business logic classes that first get entity data from the database through the database class (UsersDb), which inherits Entity Framework Core's DbContext class for data access, convert the data to the response model object and return the response model object to the controller action for presentation (query).
    Secondly, handler classes get request model data from the controller action, convert the data to the entity object for create and update operations, or use unique identifier (ID) for delete operation, in the database. Then they return a response model object to the controller action to present the result of the operation (create, update and delete).

11. Request and response model classes are also called Data Transfer Object (DTO) classes.

12. Synchronous methods execute tasks one after another. Each operation must complete before the next one starts. The calling thread waits (or "blocks") until the method finishes.
    Asynchronous methods allow tasks to run in the background. The calling thread does not wait for the operation to finish and can continue executing other code.
    In C#, asynchronous methods often use the async and await keywords, enabling non-blocking operations (such as I/O or database calls) and improving application responsiveness.

13. Service Lifetimes in ASP.NET Core Dependency Injection:

    1) AddScoped:
    - Lifetime: Scoped to a single HTTP request (or scope).
    - Behavior: Creates one instance of the service per HTTP request.
    - Use case: Use when you want to maintain state or dependencies that last only during a single request.
    - Example: DbContext, which should be shared across operations within a request, generally added with AddDbContext method.

    2) AddSingleton:
    - Lifetime: Singleton for the entire application lifetime.
    - Behavior: Creates only one instance of the service for the whole app lifecycle.
    - Use case: Use for stateless services or global shared data/services.
    - Example: Caching services, configuration providers, logging services.

    3) AddTransient:
    - Lifetime: Transient (short-lived).
    - Behavior: Creates a new instance every time the service is requested.
    - Use case: Use for lightweight, stateless services that are cheap to create.
    - Example: Utility/helper classes without state.

    Notes:
    - Injecting a Scoped service into a Singleton can cause issues due to lifetime mismatch.
    - ASP.NET Core DI container will warn about such mismatches.

```
Summary:
| Method        | Lifetime                | Instance Created              | Typical Use Case                   |
|---------------|-------------------------|-------------------------------|------------------------------------|
| AddScoped     | Per HTTP request        | One instance per request      | DbContext, per-request services    |
| AddSingleton  | Application-wide        | One instance for app lifetime | Caching, config, logging           |
| AddTransient  | Every time requested    | New instance each time        | Lightweight stateless helpers      |
```

14. SOLID Principles:
    1) Single Responsibility Principle (SRP):
       A class should have only one reason to change, meaning it should have only one job or responsibility.
    2) Open/Closed Principle (OCP):
       Software entities (classes, modules, functions) should be open for extension but closed for modification. You should be able to add new functionality without changing existing code.
    3) Liskov Substitution Principle (LSP):
       Subtypes must be substitutable for their base types. Derived classes should extend base classes without changing their behavior.
    4) Interface Segregation Principle (ISP):
       No client should be forced to depend on methods it does not use. Prefer small, specific interfaces over large, general-purpose ones.
    5) Dependency Inversion Principle (DIP):
       High-level modules should not depend on low-level modules; both should depend on abstractions (e.g., interfaces). This is commonly implemented in ASP.NET Core using dependency injection, as seen in Program.cs.

       Note: Concrete type object injection is not suitable for SOLID Principles (D: Dependency Inversion) and Unit Testing.

15. ASP.NET Core Environments:

    The environment is development when the Users.API is run from Visual Studio choosing a development profile from the launchSettings.json file in the Properties folder of the Users.API Project.
    When the Users.API is run on a server or from Visual Studio choosing a production profile from the launchSettings.json file, the environment is production.
    In launchSettings.json, http profile was defined as Development through ASPNETCORE_ENVIRONMENT section, https profile's ASPNETCORE_ENVIRONMENT value was changed to Production from Development for using the production environment.
    The environments can be changed from the drop down list (selected as https) near the run button under the Visual Studio top menu when running the application.

Before, Users.API Project must be set as startup project from the drop down list at left of the run button.
If https (production environment) is selected, sections in appsettings.json will be used for configuration, if http (development environment) is selected, sections in appsettings.Development.json will be used for configuration.
Therefore, same sections with some different values (such as connection string) must present in both files.
Environment configurations and usage is not a must for the development of projects.

IWebHostEnvironment injected instance can be used to get the web application's running environment information such as development environment or not by using IsDevelopment method.

16. var in C#:
    var can be used instead of types if there is an assignment.

```
// Way 1:
// User user = new User();
// Way 2:
var user = new User();

// Way 1:
//int number = 17;
// Way 2:
var number = 17;

// Way 1:
//string role = "Admin";
// Way 2:
var role = "Admin";
```

17. Different Generic Type implementations for classes (can also be applied to interfaces):
    Multiple different types (T1, T2, T3, ...) can also be used for example:
    public abstract class DbService<T1, T2> where T1 : class, new() where T2 : class, new()

```
// Way 1: T can be any type
public abstract class DbService<T>

// Way 2: T must be only a reference type
public abstract class DbService<T> where T : class

// Way 3: T must be only a reference type
// with a parameterless constructor.
public abstract class DbService<T> where T : class, new()

// Way 4: T must be only a type inherited from the Record class
// with a parameterless constructor.
public abstract class DbService<T> where T : Record, new()
```

18. Attributes in C#:

Attributes gain new features to the fields, properties, methods or classes. When they are used in entities or requests, they are also called data annotations which provide data validations.

Some commonly used data annotation attributes in C#:

[Required] // Ensures the property must have a value.

[StringLength] // Sets maximum (and optionally minimum) length for strings.

[Length] // Sets maximum and minimum length for strings.

[MinLength] // Specifies the minimum length for strings or collections.

[MaxLength] // Specifies the maximum length for strings or collections.

[Range] // Defines the allowed range for numeric values.

[RegularExpression] // Validates the property value against a regex pattern.

[EmailAddress] // Validates that the property is a valid email address.

[Phone] // Validates that the property is a valid phone number.

[Url] // Validates that the property is a valid URL.

[Compare] // Compares two properties for equality (e.g., password confirmation).

[DisplayName] // Sets a friendly name for the property (used in error messages/UI).

[DataType] // Specifies the data type (e.g., DateTime) for formatting/UI hints.

ErrorMessage parameter can be set in all data annotations to show custom validation error messages:

Example 1: [Required(ErrorMessage = "{0} is required!")]

where {0} is the DisplayName (used in MVC) if defined otherwise property name.

Example 2: [StringLength(100, 3, ErrorMessage = "{0} must be minimum {2} maximum {1} characters!")]

where {0} is the DisplayName (used in MVC) if defined otherwise property name, {1} is the first parameter which is 100 and {2} is the second parameter which is 3.

19. Some LINQ (Language Integrated Query) methods for querying data (async versions already exists):

Find: Finds an entity with the given primary key value. Returns null if not found.
Uses the database context's cache before querying the database.
Example: var group = _db.Groups.Find(5);
Single: Returns the only element that matches the specified condition(s).
Throws an exception if no element or more than one element is found.
Example: var group = _db.Groups.Single(groupEntity => groupEntity.Id == 5);

SingleOrDefault: Returns the only element that matches the specified condition(s), or null if no such element exists.
Throws an exception if more than one element is found.
Example: var group = _db.Groups.SingleOrDefault(groupEntity => groupEntity.Id == 5);

First: Returns the first element that matches the specified condition(s).
Throws an exception if no element is found.
Example: var group = _db.Groups.First();
Example: var group = _db.Groups.First(groupEntity => groupEntity.Id > 5 && groupEntity.Title.StartsWith("Jun");

FirstOrDefault: Returns the first element that matches the specified condition(s), or null if no such element exists.
Example: var group = _db.Groups.FirstOrDefault();
Example: var group = _db.Groups.FirstOrDefault(groupEntity => groupEntity.Id < 5 || groupEntity.Title == "Senior");

Last: Returns the last element that matches the specified condition(s).
Throws an exception if no element is found. Usually requires an OrderBy or OrderByDescending clause.
Example: var group = _db.Groups.OrderByDescending(groupEntity => groupEntity.Id).Last(); gets the first group from the groups descending ordered by Id.
Example: var group = _db.Groups.OrderBy(groupEntity => groupEntity.Id).Last(); gets the last group from the groups ordered by Id.

LastOrDefault: Returns the last element that matches the specified condition(s), or null if no such element exists.
Usually requires an OrderBy or OrderByDescending clause.
Example: var group = _db.Groups.OrderBy(groupEntity => groupEntity.Id).LastOrDefault();
Example: var group = _db.Groups.OrderBy(groupEntity => groupEntity.Id).LastOrDefault(groupEntity.Title.Contains("io"));

Where: Returns the filtered query that matches the specified condition(s). Tolist, SingleOrDefault or FirstOrDefault methods are invoked to get the filtered data.
Example: var groups = _db.Groups.Where(groupEntity => groupEntity.Id > 5).ToList();

Note: SingleOrDefault is generally preferred to get single data.
Note: These LINQ methods can also be used with collections such as lists and arrays.

20. Eager Loading and Lazy Loading with Entity Framework:

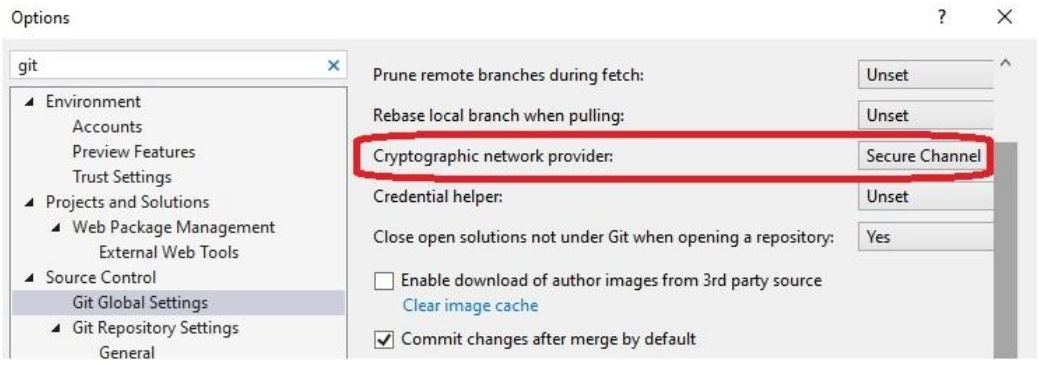
    Relational entity data can be included to the query by using the Include method (Entity Framework Eager Loading).

    If the included relational entity data has a relation with other entity data, ThenInclude method is used.

    If you want to automatically include all relational data without using Include / ThenInclude methods (Entity Framework Lazy Loading), you need to make the necessary configuration in the class inheriting from DbContext class (Db) to enable Lazy Loading (not recommended).

21. Visual Studio 2022 Community GitHub push problem solution:

From Visual Studio Menu
Tools -> Options -> Source Control -> Git Global Settings
Change Cryptographic network provider to Secure Channel



22. If you change the database table structures (such as adding or removing entity properties, changing entity property data types or data annotations, adding or removing entities and db sets) or relationships (such as changing Cascade delete rule to No Action) through the entities or the DbContext inheriting class, you need to update the database as below:

    - Open Package Manager Console from Visual Studio menu -> Tools -> NuGet Package Manager -> Package Manager Console

- Right-click Users.API Project and click Set as Startup Project
- Set Users.APP as Default Project in Package Manager Console
- Run:
  add-migration v2
  update-database
- For Rider, you can use the UI as described in JetBrains documentation, or from the terminal
  Run:
  dotnet ef migrations add v2
  dotnet ef database update -p Users.APP -s Users.API

23. Optionally database configurations such as using no action (will not allow to delete the records from the relational table when a record is deleted from the main table) instead of cascade (which will automatically delete the records from the relational table when a record is deleted from the main table) between tables may be done by overriding the OnModelCreating method in the UsersDb database context class. Default is cascade in Entity Framework.
    Changing column configurations instead of using data annotations in entities (e.g. making a column required or setting maximum length), changing table names etc. can also be done in the OnModelCreating method among other configurations.