**Api Development**

Here, we have layers. We have one solution for many projects. When we name our controllers in Api we normally add “s” to the end of the name. 

And in the controller we specify the followings 🡪



**Localhost:portNumber/api/Students** for instance.

It is also important to specify the request with an attribute 🡪



In api applications, our route doesn’t include the action name, instead it includes the name that is written in the Http attribute for example here it will be **Localhost:portNumber/api/Students/list**

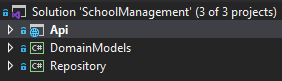
If we don’t have the name for the attribute then others actions must have the name otherwise there will be a conflict.

Here our controllers inherit from ControllerBase. We don’t need to inherit from the Controller class since we don’t use views here.



**Layers**

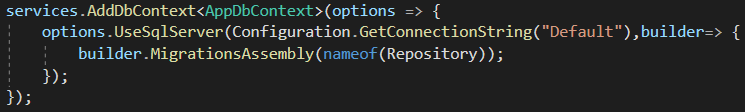
In api apps, we have a repository , DomainModels and Api project itself 🡪



DomainModels and Repository are class libraries. In the DomainModels we have our Models and Dtos (data transfer objects).

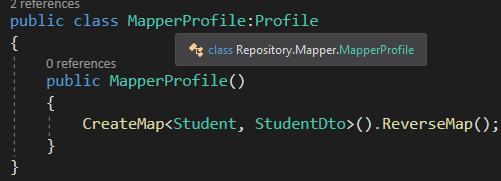
In the Repository, we have got our DAL which means we also need the following packages: EntityFrameworkCore and EntityFramework. SqlServer. EntityFrameworkCore is to inherit from DbContext and EntityFramework.SqlServer is for migrations. We will place the migrations folder in the repository project!

In order for migrations to be in the Repository project we need to explicitly tell it in the Startup 🡪



**Mapper**

We place our mappers in the repository project. We also need to add AutoMapper package to the repository project so that we can do the followings 🡪



The naming conventions is to add “Profile” to the end of our mapper name. we inherit from Profile and in the constructor we create our mappers. The first object is the source and the second one is the destination.

In order to use our mappers, in the api project we need the AutoMapper.Extensions.Microsoft.DependencyInjection package so that we can inject our mapper to controllers.

**Mapper basically creates a new instance of a specified object.** 🡪



For example, here this mapper creates a list of studentdtoes, and it maps StudentDto list insance from students list

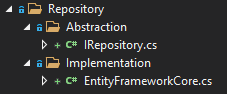
Also in the Startup we need to add the following 🡪



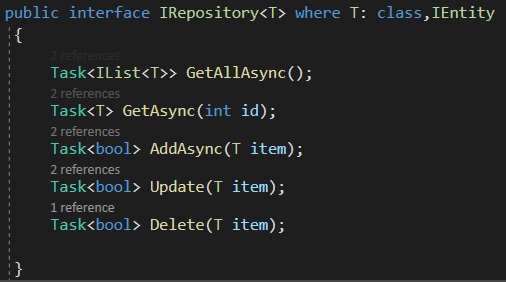
**Repository pattern**

The repository and unit of work patterns are intended to create an abstraction layer between the data access layer and the business logic layer of an application.

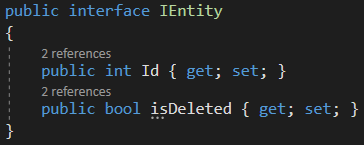
To implement repository pattern, first we create our abstaction and implementation folders in the repository folder. 🡪

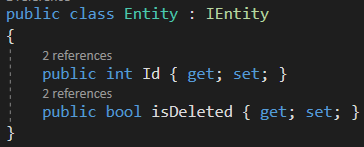


Our IRepository interface will have all the abstarct methods of our database 🡪



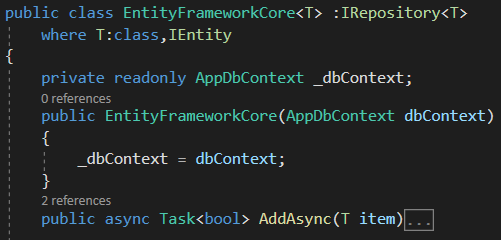
TEntity is our base class that all the models inherit from 🡪



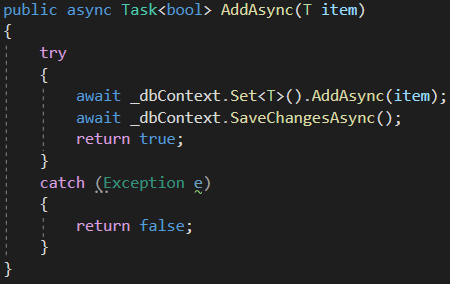


All of our models inherit from this Entity class.

Then we create our EntityFrameworkCore class that inherits from IRepository. It will have all the implementations of IRepository interface.



Just note that when we interact with the database through the T parameter type we use the following Set() method 🡪



After this we need to add AddScoped sevice for IRepository 🡪



Then in our controller, for example Student controller we inject IRepository through the constructor to our controller 🡪



**Route**

When we send a request we need to be careful about the route. First of all, in api apps we name our controllers with s suffix for example, StudentsController and so on. In api, We don’t use action names for the routing. However we can add extra names through the attributes 🡪



Here for instance, we take id from the route so [FromRoute] attribute is used and from the body we take bookdto object. Also in the attribute we need to specify it 🡪

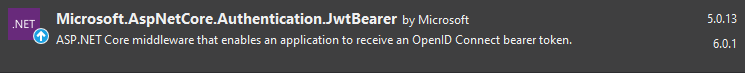
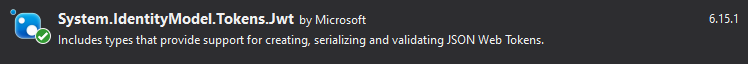


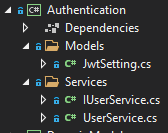
Note that update is HttpPut here.

**JWT token**

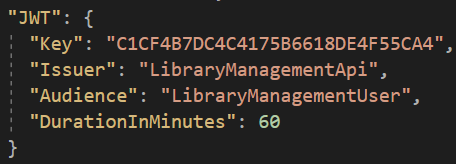
JSON Web Token (JWT) is one of the most commonly used techniques to secure APIs, allowing users to access resources they are authorized to.

First we create a new class library project for Authentication and install the following packages 🡪JWT bearer and JWT



Then we add the Settings for JWT to our appsettings.json.



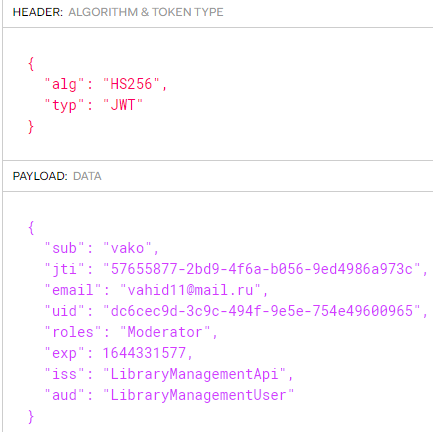
Issuer is who gives this JWT and Audience is to whom it is given. We are the api and we give it to Users.

Key – The Super Secret Key that will be used for Encryption. You can move this somewhere else for extra security.

Issuer – identifies the principal that issued the JWT. A principal is essentially another name for a company owner or member; at some corporations, the principal is also the founder, CEO, or even the chief investor.

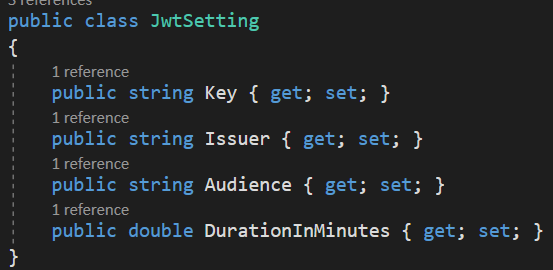
Audience – identifies the recipients that the JWT is intended for. DurationInMinutes – Defines the Minutes the generated JWT will remain valid.

When we send our token that token will have all this information (issuer, audience, token key and etc)



Now when we send a request we will send with this jwt token, first we are gonna be authorized then the info will be give if it is 200 status code. 401 is unauthorized

Then in our Models of Authentication project we create a JwtSetting.cs file which will be used to read data from our previously created JWT Section of appsettings.json using the IOptions feature of ASP.NET Core.



Then we close everywhere meaning that we put [Authorize] attribute everywhere and the only open place will be the UsersContorller so that the user can login.

In our startup 🡪



Authentication must be before Authorization

//Configuration from AppSettings

services.Configure<JwtSetting>(Configuration.GetSection("JWT"));

**This will create a new JWTSetting model and will map everything from our JWT JSON of appsettings.json**

services.AddScoped<IUserService, UserService>();

//Adding Athentication - JWT

services.AddAuthentication(options =>

{

options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;

options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;

})

.AddJwtBearer(o =>

{

o.RequireHttpsMetadata = false;

o.SaveToken = false; Should I save the token in cookies or somewhere else?

o.TokenValidationParameters = new TokenValidationParameters

{

ValidateIssuerSigningKey = true,

ValidateIssuer = true,

ValidateAudience = true,

ValidateLifetime = true,

ClockSkew = TimeSpan.Zero,

ValidIssuer = Configuration["JWT:Issuer"],

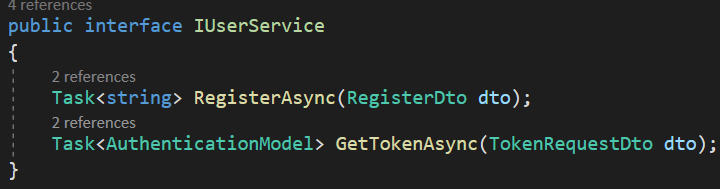
ValidAudience = Configuration["JWT:Audience"],

IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(Configuration["JWT:Key"]))

};

});

We will need a Service class that contains the Core User Functions like Register, Generate JWTs etc. Create a new Interface, Services/IUserService.cs



In our UserService class we will implement this interface. In UserService class we will have the following methods RegisterAsync, GetTokenAsync and CreateJwtToken.

All the information can be found in the shown link🡪 <https://codewithmukesh.com/blog/aspnet-core-api-with-jwt-authentication/>

In a nutshell, register is just gonna register a user and GetTokensAsync is just gonna create a token that will last 60 mins for the user and we will be sending Authentication model back and forth.

TokenRequestDto is just a dto for a user that wants to login so there will be email and password properties.

