**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**

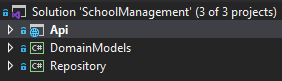
If we don’t have the name for the attribute then others actions must have the name otherwise there will be a conflict.

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



**Layers**

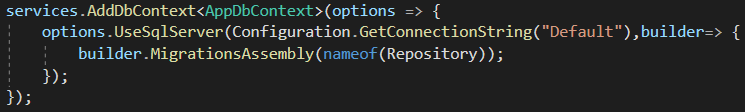
In api apps, we have a repository , DomainModels and Api project itself 🡪



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

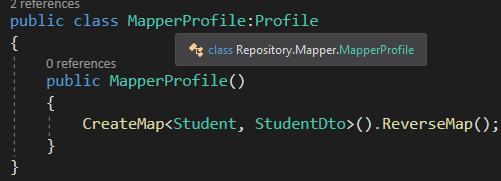
In the Repository, we have got our DAL which means we also need the following packages: EntityFrameworkCore and EntityFramework. SqlServer. EntityFrameworkCore is to inherit from DbContext and EntityFramework.SqlServer is for migrations. We will place the migrations folder in the repository project!

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

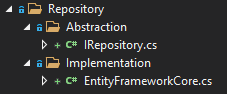
Also in the Startup we need to add the following 🡪



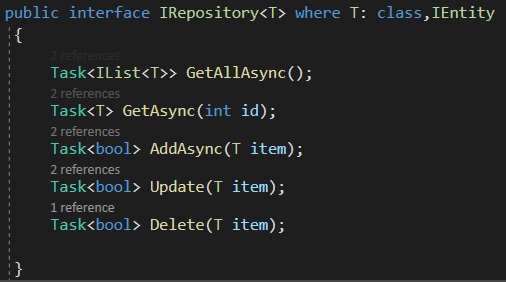
**Repository pattern**

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.

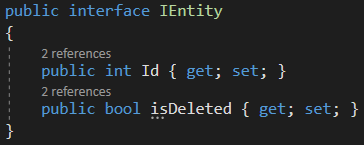
To implement repository pattern, first we create our abstaction and implementation folders in the repository folder. 🡪

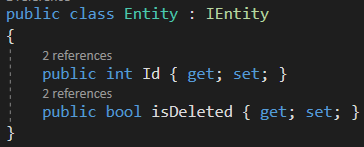


Our IRepository interface will have all the abstarct methods of our database 🡪



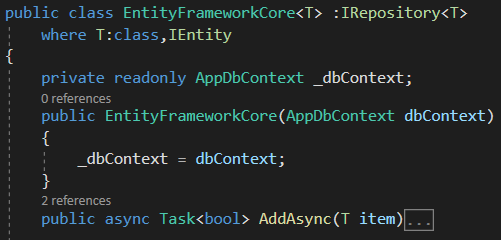
TEntity is our base class that all the models inherit from 🡪



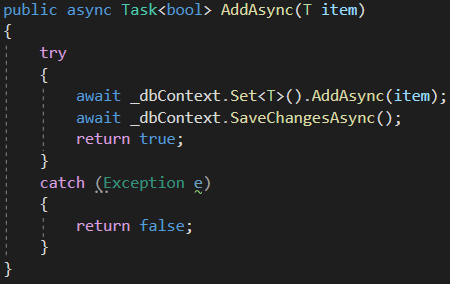


All of our models inherit from this Entity class.

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.