**Api Development**

There are different kinds of apis. We are going to talk about web apis first, more specifically rest apis.

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**

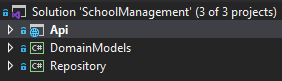
If we don’t have the name for the attribute then others actions must have the name otherwise there will be a conflict.

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



**Layers**

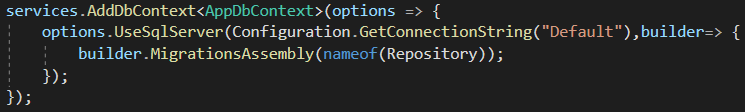
In api apps, we have a repository , DomainModels and Api project itself 🡪



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

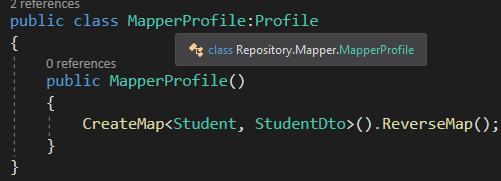
In the Repository, we have got our DAL which means we also need the following packages: EntityFrameworkCore and EntityFramework. SqlServer. EntityFrameworkCore is to inherit from DbContext and EntityFramework.SqlServer is for migrations. We will place the migrations folder in the repository project!

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**

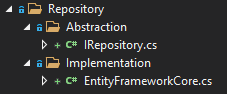
The repository and unit of work patterns are intended to create an abstraction layer between the data access layer and the business logic layer of an application. The repository pattern creates the abstraction layer between database access and business logic. Instead of writing the entire data access logic on the controller, it's better to write it in a different class called a repository. It hides the details of how exatly the data is saved or retrieved.

It tackles the problem of duplication. Otherwise, we would violate the DRY rule. Also some of the problems that it solves: tight coupling, rigid code, testability, separation of concerns

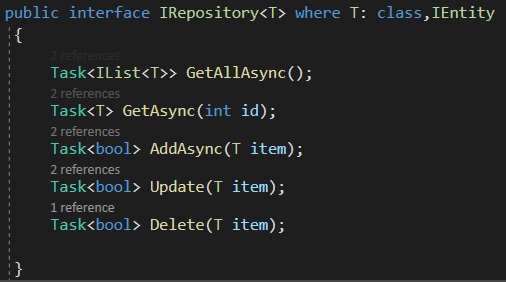
**Non-Generic Repository pattern** - All database actions related to a given entity are defined using the non-generic repository pattern within a separate class. For Example, suppose we have two entities (Student and Employee ). In that case, we will create two repositories for each entity with the basic CRUD operations and other operations related to each entity.

**Generic Repository pattern-** The generic repository pattern is used to provide common database operations for all database entities in a single class, such as CRUD operations and so on.

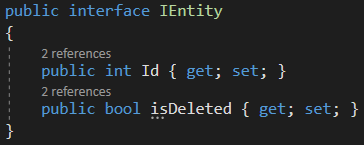
To implement repository pattern, first we create our abstaction and implementation folders in the repository folder. 🡪

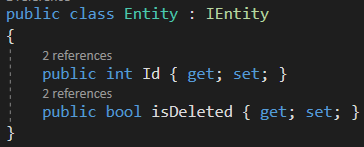


Our IRepository interface will have all the abstarct methods of our database 🡪



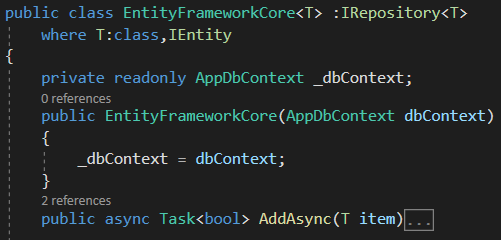
When we retrieve data from Repository it cannot be IQuerable because then we would be violating the repository pattern rules we don’t want to retrieve IQuerable what we want is IEnumerable. TEntity is our base class that all the models inherit from 🡪



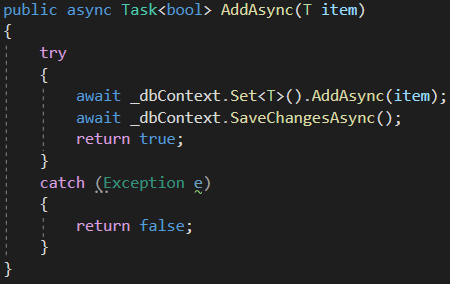


All of our models inherit from this Entity class.

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 and if it is generic then we need to use the typeof expression🡪

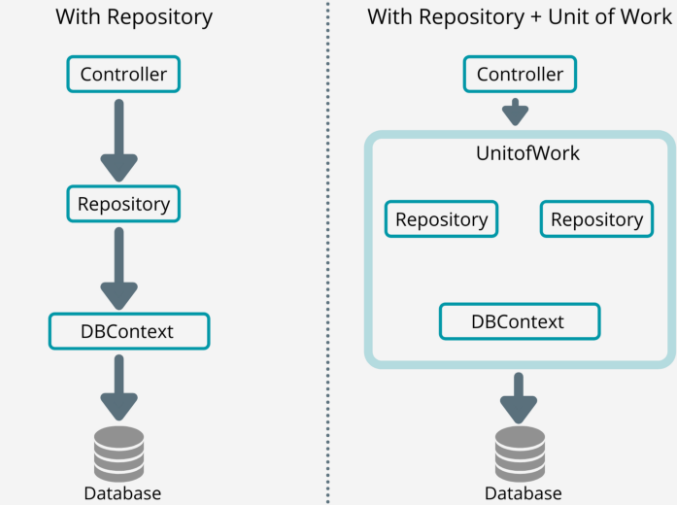


Then in our controller, for example Student controller we inject IRepository through the constructor to our controller 🡪

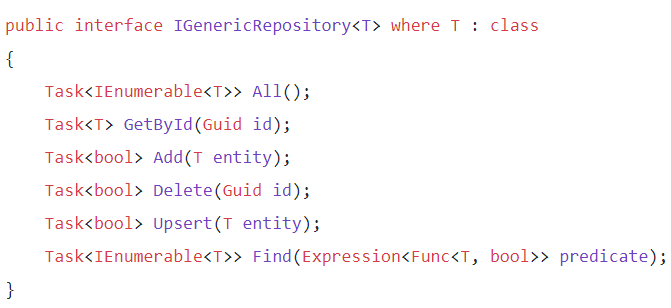


**Unit of Work pattern(UoW)**

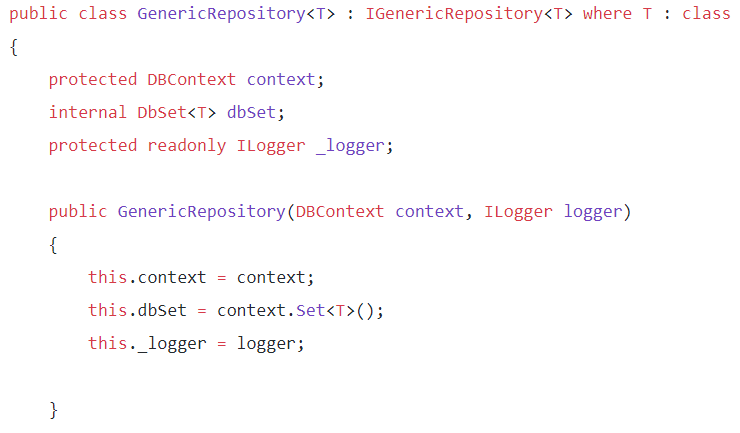
UoW pattern tracks a business transaction and translates it into the database transaction. So for context.SaveChanges method we will use the unit of work. UnitOfWork will have all the repositories.



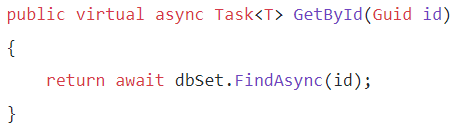
Our GenericIRepository interface 🡪



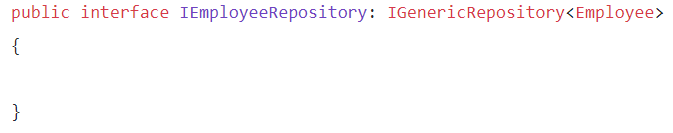
GenericRepository class will implement this interface and the methods will have to be virtual so that when it is inherited from another Repository class the methods can be overriden if anything🡺



For example GetById method will be virtual 🡪



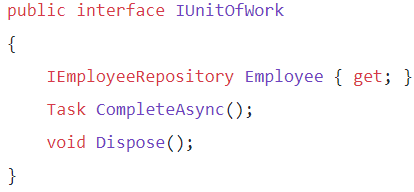
After this we create our IRepositories for specific models and they will all implement IGenericRepository interface so that they all have common methods.



Here we could have some methods that are specific to Employee model for example GetTopEmployees and etc.

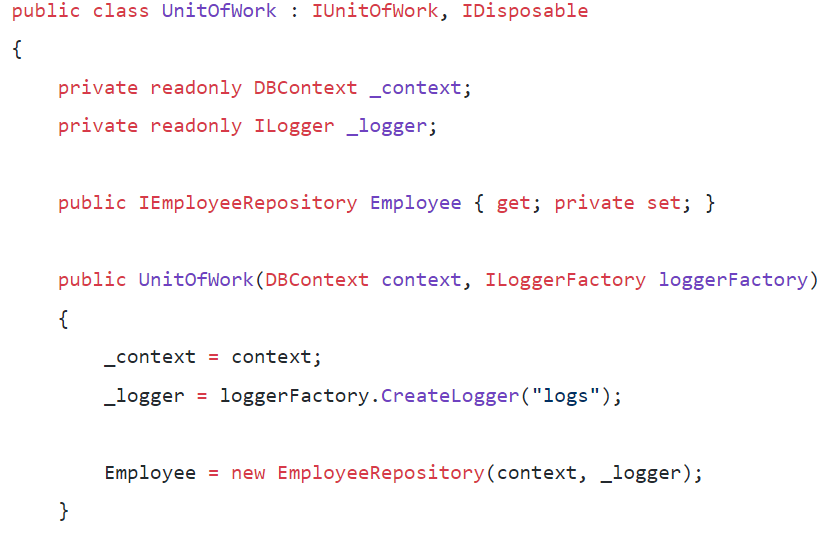


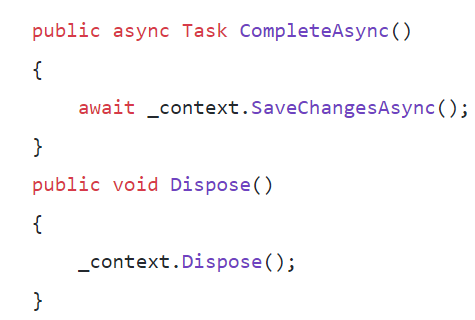
Here we inherit from GenericRepository class so that we don’t implement those common interfaces again. Here as you can see if we want to have a different logic for Employee model when adding then we just override that method.



CompleteAsync is for saving to the database and ending the entity tracking.

Unif of work class (implementor) will also implement IDisposable interface so that it can be disposed of.







We only have to register IUnitOfWork. We don't need to register other repositories as we are doing in the basic repository pattern. Because we inject all of the repositories in the IUnitOfWork, then we could access them through it.

Then in our controllers we will use IUnitOfWork to access and save data to the database.

**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 🡪

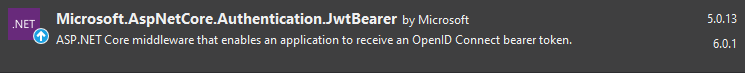
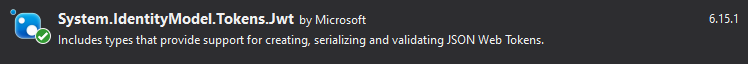


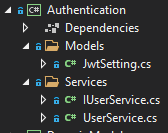
Note that update is HttpPut here.

**JWT token**

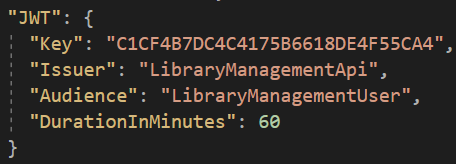
JSON Web Token (JWT) is one of the most commonly used techniques to secure APIs, allowing users to access resources they are authorized to.

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.



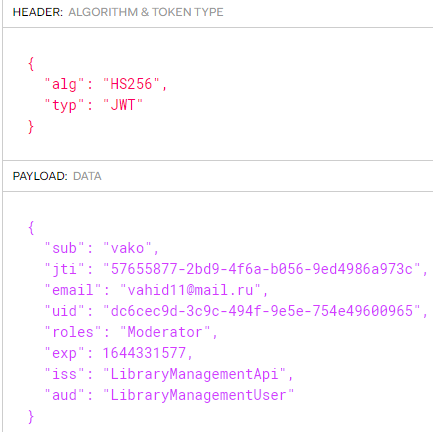
Issuer is who gives this JWT and Audience is to whom it is given. We are the api and we give it to Users.

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.

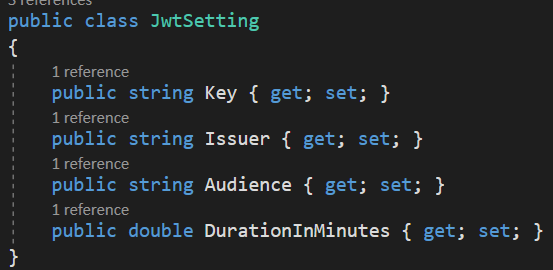
Audience – identifies the recipients that the JWT is intended for. DurationInMinutes – Defines the Minutes the generated JWT will remain valid.

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"],

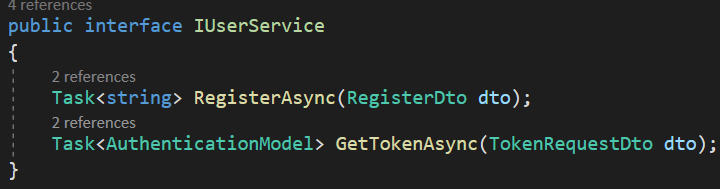
ValidAudience = Configuration["JWT:Audience"],

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



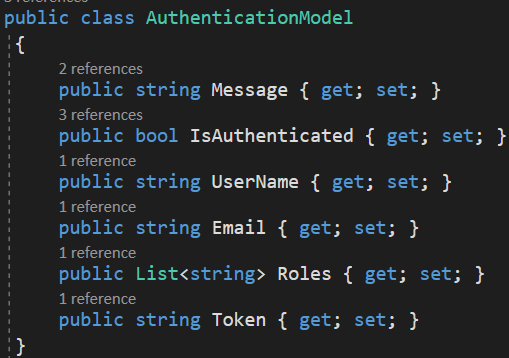
In our UserService class we will implement this interface. In UserService class we will have the following methods RegisterAsync, GetTokenAsync and CreateJwtToken.

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.

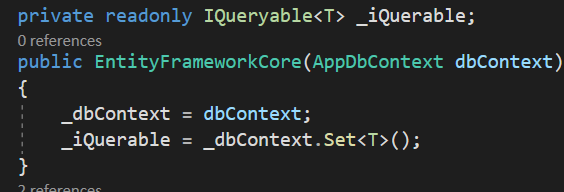
Remember that GetTokensAsync although it doesn’t send a request to the database, it creates something (jwt token) therefore It must be HTTP Post request. Post requests are more secure than Get requests.

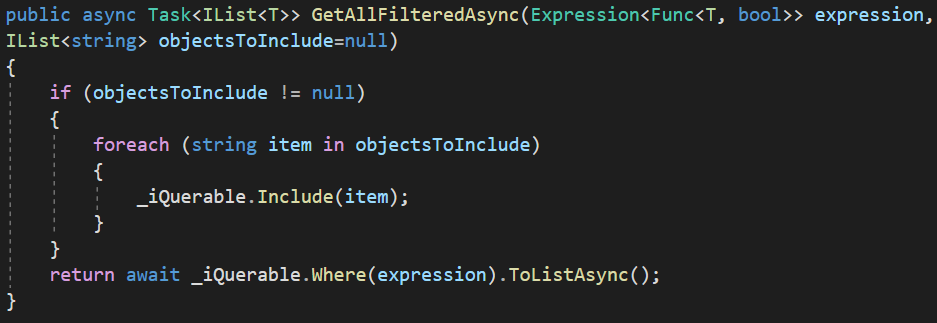
TokenRequestDto is just a dto for a user that wants to login so there will be email and password properties.



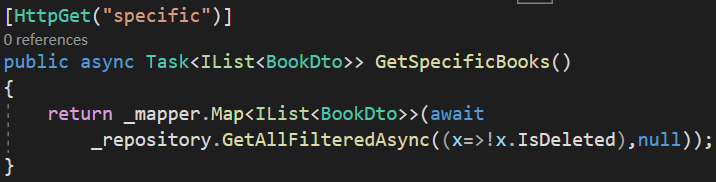
**Filtering database queries with IRepository**

If we want to write a different query then we are going to use an expression. We can also ease our lives by setting IQuerable to \_dbContext.Set<T>() so that we don’t repeat these Sets<T>()🡪





We will pass an expression and take whatever we want. Also, if we desire to include anything we can pass the collection of the models that we want to include 🡪



**Rest and Restful Api**

Rest stands for representational state transfer. It is an architecture style for designing networked apps. It relies on a stateless protocol. So an api is the messenger and Rest lets us use HTTP reqeusts to format those messages. Restful api refers to conforming to the rest constraints. They are essentially the same thing. So Rest is a set of guidelines and Resful api is the implementation of rest.

When you create something we must use HTTP Post request otherwise we violate the rest rules.

Rest api must accept and respond with JSON. Over 100 status codes have already been built by HTTP. It is a boon for developers to use status codes in their REST API design.

Some rest api rules 🡪

1. REST API MUST ACCEPT AND RESPOND WITH JSON
2. GO WITH ERROR STATUS CODES
3. DON’T USE VERBS IN URLS
4. USE PLURAL NOUNS TO NAME A COLLECTION
5. RETURN ERROR DETAILS IN THE RESPONSE BODY
6. SECURE YOUR API

HTTP GET should be used for all *retrieval*. It should never be used to create, update, or do things.

HTTP POST should be used for *creating*. It shouldn’t be used to update or get a resource.

HTTP PUT should be used for *updating* — meaning *replacing* a collection with different data.

HTTP DELETE should be used for *deleting*.

**EndPoints**

<https://example.com/this-is-an-endpoint>

<https://example.com/another/endpoint>

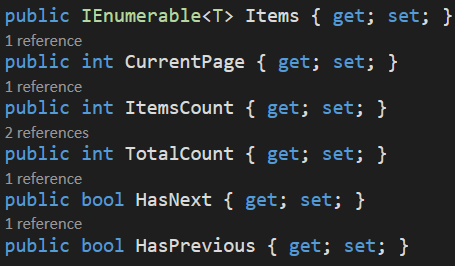
<https://example.com/some/other/endpoint>

**Pagination**

In order to realize pagination first we create a Pagination folder in which we will create our PaginationDto model 🡪



In the PaginationDto we will need to keep track of a couple properties 🡪

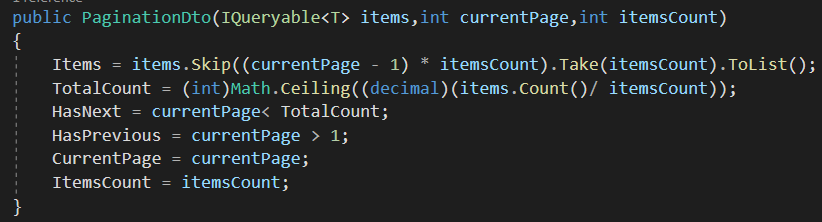


Items- the items for placing in the page

CurrentPage- The number of the current page

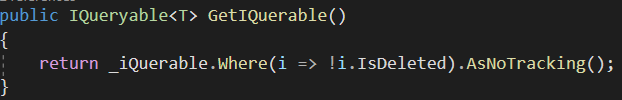
ItemsCount- The number of items for each pagination

TotalCount- the number of pages left after paginating

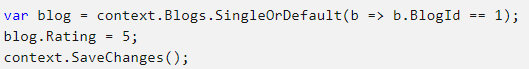
Now we need to define these properties. We use the constructor 🡪  


So we will receive IQuerable of Ts , currentpage number and how many items we want to skip for each pagination.

And when we take this IQeurable we need to tell dbContext not to track this.



By default, queries that return entity types are tracking. Which means you can make changes to those entity instances and have those changes persisted by SaveChanges(). In the following example, the change to the blogs rating will be detected and persisted to the database during SaveChanges().

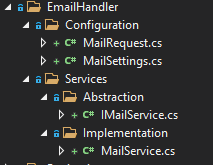


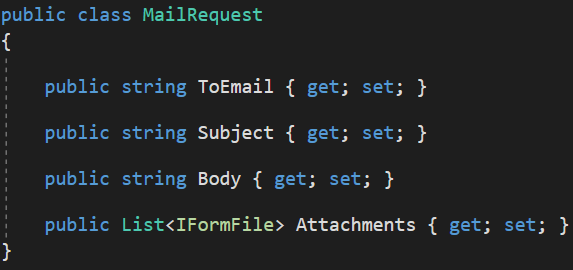
Once we make the query it will be tracked till the saveChanges method.

No tracking queries are useful when the results are used in a read-only scenario. They're quicker to execute because there's no need to set up the change tracking information. If you don't need to update the entities retrieved from the database, then a no-tracking query should be used. You can swap an individual query to be no-tracking. No tracking query will also give you results based on what is in the database disregarding any local changes or added entities.

**Email Service**

We will need parameters like Email, Subject, Body, List of Attachments in the Request Model so that these data can be passed on to the service layer. We create a EmailHandler folder in which we will have Configuration folder and we will create MailRequest in Configuration folder 🡪





We also need to add the information about our email 🡪



Port and Host are taken from the website itself link 🡪<https://codewithmukesh.com/blog/send-emails-with-aspnet-core/>

Wonder what is Host, Port and how i got these sets of data? Well, to send mails, we need a server. Now it’s quite not practical to setup your own server/applications to do that. Hence we use server of other providers. The most popular one is Gmail ofcourse. In our case, the host refers to an SMTP Server.

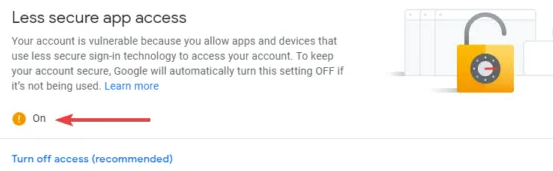
SMTP or Simple Mail Transfer Protocol servers are just applications whose primary aim to send and receive mails. When you send an email, the SMTP server determines the target domain and relays the mail to the corresponding server.

Developers usually prefer using the GMAIL SMTP server for production and testing purposes, as it is quite easy and reliable too. Gmail allows you to send 2,000 emails per day using the Gmail SMTP Server for FREE.

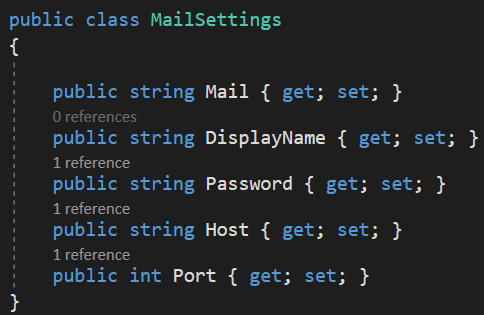
Here is what you will need to use Gmail SMTP.

1. Your Gmail ID (Do not use a personal Email ID, quite risky if compromised.) I always have a mail-id specifically for this purpose.
2. The Password.
3. Host or SMTP Server address – If you are going with Gmail, use **smtp.gmail.com**
4. Port – Use 465 (SSL) or 587 (TLS)

After you get the details, we will need to allow your Gmail account to send emails using external (less- secure applications). To enable this, go to [https://myaccount.google.com/security](https://myaccount.google.com/security%20) and turn ON Less Secure App Access.



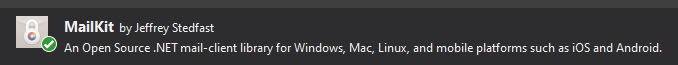
After adding all the required data to the appsettings.json, we will have to make a model that will hold the details from this JSON File. Let’s call it MailSettings. Add a new class, Settings/MailSettings.cs

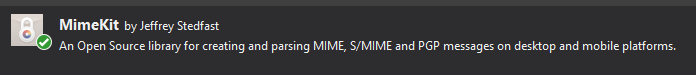


We have the data in our JSON file. How do we tranfer these data to an instance of MailSettings at runtime? You are right. IOptions and Dependency Injection to the rescue. Navigate to Starup/ConfigureServices Method and add the following line.



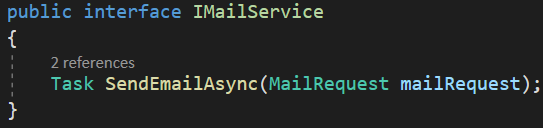
We will be using Mailkit and Mimekit packages for this implementation. These are probably the only two nugget packages that you will need to implement mailing in your ASP.NET Core Applications.



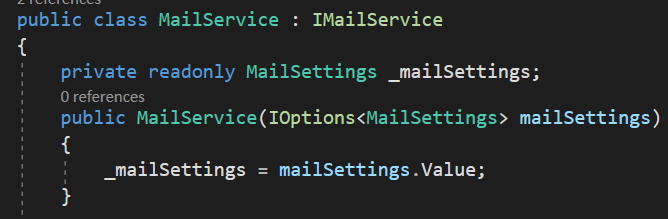


MailKit does server work and MimeKit is for messaging.

Now with that out of the way, let’s build Service classes that is actually responsible to send Mails.



And the implementation of the interface 🡪



You can see that we are Injecting the IOptions<MailSettings> to the constructor and assigning its value to the instance of MailSettings. Like this, we will be able to access the data from the JSON at runtime.

public async Task SendEmailAsync(MailRequest mailRequest)

{

var email = new MimeMessage();

email.Sender = MailboxAddress.Parse(\_mailSettings.Mail);

email.To.Add(MailboxAddress.Parse(mailRequest.ToEmail));

email.Subject = mailRequest.Subject;

var builder = new BodyBuilder();

if (mailRequest.Attachments != null)

{

byte[] fileBytes;

foreach (var file in mailRequest.Attachments)

{

if (file.Length > 0)

{

using (var ms = new MemoryStream())

{

file.CopyTo(ms);

fileBytes = ms.ToArray();

}

builder.Attachments.Add(file.FileName, fileBytes, ContentType.Parse(file.ContentType));

}

}

}

builder.HtmlBody = mailRequest.Body;

email.Body = builder.ToMessageBody();

using var smtp = new SmtpClient();

smtp.CheckCertificateRevocation = false;

smtp.Connect(\_mailSettings.Host, \_mailSettings.Port, SecureSocketOptions.StartTls);

smtp.Authenticate(\_mailSettings.Mail, \_mailSettings.Password);

await smtp.SendAsync(email);

smtp.Disconnect(true);

}

At last we will call this method 🡪

