Architecting

Modern Applications

Using Monolithic Architecture

In ASP.NET Core Web API

Author

Sardar Mudassar Ali Khan

Architecting Modern Applications using Monolithic Architecture in Asp.Net Core Web API

Practical Approach for Architecting Software Products

By

Sardar Mudassar Ali Khan

Senior Software Engineer

Scientific Researcher

Blogger at C# Corner

AUTHOR NOTE

I welcome you all during **Architecting the modern web application in Asp.Net core web API** the topic discussed in this book based on practical knowledge of the industry. In this book we will study the complete projects and discuss the project from start to end we will apply all the software Development principal, Design Architecture and Design Patterns, and Generics that are very beneficial for the development of software products. We will architect a complete application from start to end using Different Software Architectures.

This book is the property of the Author and copying any kind of Content is a criminal offense so if you see any mistake in this book then contact us at the official email address paksoftvalley@gmail.com or the official email address of Author mudassarali.official@gmail.com.
Every step taken to word the completion of this book is based on an expert review of industry experts.

SARDAR MUDASSAR ALI KHAN
Blogger at C# Corner
Senior Software Engineer
Scientific Researcher

DEDICATION

Every single step that was taken towards the completion of the book was because of the strong background provided by my parents they supported me morally. Moreover, the community of **C# Corner** helped me a lot in getting knowledge and solution where I was stuck. They motivate me in every situation and always ask to us never give up everything possible in the world. They held our hands firmly never letting us get down and made it possible for us to get through. This book is also dedicated to my seniors who motivated us to guide us in this regard.

The pure dedication of my book is to my parents our soul parents- the teacher and friends. Teachers always motivated and boosted us with full support whenever required.

DECLARATION

I Sardar Mudassar Ali hereby declare that this book contains original work and that not any part of it has been copied from any other sources. It is further declaring that I have completed my book under the project of ***Free Education for Humanity*** and with the guidance of my teachers and my senior colleagues and this is entirely possible based on my efforts and, my ideas.

SARDAR MUDASSAR ALI

ACKNOWLEDGEMENT

It would be my pleasure and I am feeling ecstatically rapturous to pay the heartiest thanks. First, To Allah (SWT) who is most beneficent and most merciful that he enabled me and blessed me to take the pebble out of his multitude of bounties. To my dear parents, their prayers become true towards the completion of my book. I am greatly beholden to my esteemed honourable motivator and kind-hearted, teachers, friends, and colleagues who supported my idea all the way and never let me down and corporate with me in this way completely sincerely and heartedly.

Also, special thanks to those who helped me a lot in my way, but I have forgotten them to mention here.

PREFACE

Architecting the application in asp.net core web API in this book we adopted a practical approach for the development of software products. This book is divided into four parts every part has its dimension. In this book, we will explain the fully featured projects and how we can architect the application using the following architectures.

- Monolithic Architecture
- Clean Architecture
- Onion Architecture
- Three Tier Architecture
- Application Deployment on Azure

PART 1: Monolithic Architecture

1) Introduction
2) Complete Understanding of Monolithic Architecture
3) Practical Approach for Architecting the Application with Asp.net Core
4) Conclusion
5) Complete Project GitHub URL

PART 2: Three Tier Architecture.

1) Introduction
2) Complete Understanding of Three Tier Architecture
3) Practical Approach for Architecting the Application with Asp.net Core
4) Conclusion
5) Complete Project GitHub URL

PART 3: Onion Architecture.

1) Introduction
2) Complete Understanding of Three Tier Architecture
3) Practical Approach for Architecting the Application with Asp.net Core
4) Conclusion
5) Complete Project GitHub URL

PART 4: Clean Architecture.

1) Introduction
2) Complete Understanding of Three Tier Architecture
3) Practical Approach for Architecting the Application with Asp.net Core
4) Conclusion
5) Complete Project GitHub URL

PART 5: Application Deployment on Server.

1) Introduction
2) Deployment Procedure
3) Server Configuration
4) Deployment Steps
5) Conclusion

PUBLISHER

CSharp Corner

https://www.c-sharpcorner.com/

About C# Corner

C# Corner is a global social community for IT professionals and data developers to exchange their knowledge and experience using various methods such as contributing articles, forums, blogs, and videos. C# Corner reaches 3+ million monthly users worldwide. The recently announced C# Corner News Section keeps you up to date with the latest technology news. A big thank you to all of C# Corner members and authors who have been giving back to the community for years.

Architecting Modern Web with Applications in Asp.Net Core

Contents

Chapter # 1- Monolithic Architecture in Asp.Net Core

- What is a software architecture?
- Software Architecture in Software Engineering
- What is Monolithic Architecture
- Understanding the monolithic architecture with software product
- Benefits of the Monolithic Application
- Implementation of Monolithic architecture
- Conclusion

What is a software architecture?

In the software industry when we want to develop a software product then we need good architecture using that architecture we can develop a high-quality product that is highly testable highly, highly scalable, and maintainable product.

Definition

"Software Architecture is simply the organization of system this organization includes all the components within the architecture how they will interact with each other with the environment in which they operate, and principles used to architect the application. In many cases, it focuses on the development of the software product into the future"

Software architecture is designed with the specific mission in mind that the product should be highly testable highly scalable and maintainable, and the behavior and structure of the software should be according to the standard and gives us the best possible results.

Software Architecture in Software Engineering

Software architecture in software engineering helps to expose the structure of the system while hiding some information. Architecture focuses on the components of the system and their relationship and how these components will interact with each other and give us the best possible result for the software product.

What is Monolithic Architecture

Monolithic architecture is the traditional and unified model for the design of software products monolithic architecture means to combine the application into one unit and the application is tightly coupled and monolithic architecture application components dependent on each other and for exaction of the application dependent component must be present for the smooth run of the application.

A monolithic application is a single-tiered application which means that multiple components of the application are combined into one single large application and the architecture of the application is highly coupled.

In a Monolithic application if we want to change in one component then the other dependent components require rewriting of the code after the change our application is recompiled Furthermore, we run the test process again and again and we repeat this process after every change. The whole process increases the cost and limits the agility and speed of the development process.

Understanding the monolithic architecture with software product

Let’s consider we have the application which gives services to the end user and these services like Cyber Security, Data Science, Software development, and DevOps Engineering, and services can be sold to the end user according to certain criteria and the end user pays for the service using the online payment method and within the application we main the credentials of the users and transaction history, project history and in the project we have the online video call feature the will be used to handle the call between the engineers and the all the stakeholders of the project.

Components of the application

1. Authentication
2. Services
3. Video Calling
4. Online Payments
5. Reports
6. Jira Project API Integration

Let’s Assume that we develop such projects using the monolithic architecture that gives us the highly coupled product and consider the above project if we want to change the authentication module then other components depend on this module after the change the dependent component needs the rewriting of the function that is used for maintaining the state of the user and the one change we need to test the entire application again and this whole process increase the cost and development time and may decrease the efficiency of the application.

IMPORTANT

If the changes are needed in one component, then the other dependent affected components also need to be changed.

Benefits of the Monolithic Application

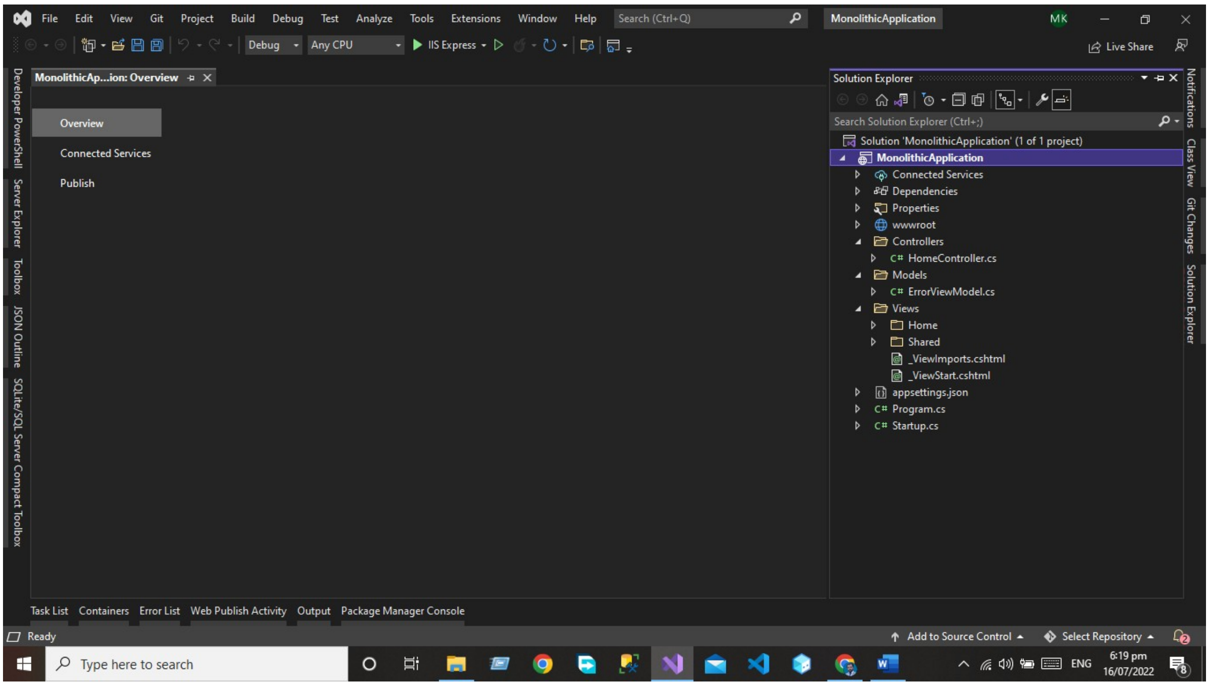
- The monolithic application may have better throughput than the modular application.
- A small application with fewer modules is easier to test and debug.
- Monolithic architecture is better for lightweight applications.

Drawbacks of Monolithic Application

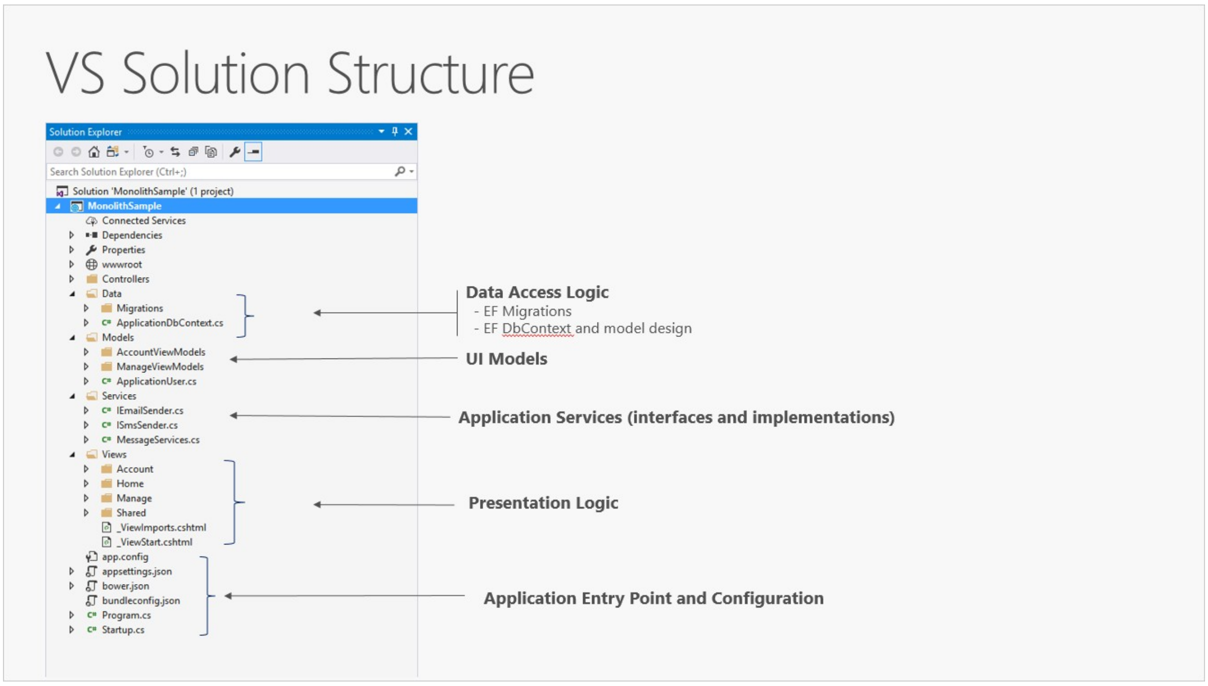
- Monolithic architecture is not suitable for complex applications.
- When the complexity of the application increases then the development team also increases in size.
- The Codebase application is difficult to understand and difficult to modify
- For Any monolithic application, the developer needs to test and deploy the entire application after a small change.
- With the monolithic application, the development cost and time also increase with changes in application structure.
- Monolithic applications are limited scalable and reliable.
- A small, big module can cause the entire application to bring down.

Implementation of Monolithic Architecture

In Monolithic architecture, all the logic of the application is combined into a single project compiled in a single assembly and deployed as a single unit. In a monolithic application, we have a single project that contains our all business, data access, and presentation logic in one application as shown in the project structure below.



For separations of concerns, we follow the folder structure in the default template we see the MVC Design pattern for models' views and controllers and some extra folders for data and services.


VS Solution Structure

Data Access Logic
- EF Migrations
- EF DbContext and model design

UI Models

Application Services (interfaces and implementations)

Presentation Logic

Application Entry Point and Configuration

In this single application, we have

Data Folder

Add the Data folder in the infrastructure layer that is used to add the database context class. The database context class is used to maintain the session with the underlying database using which you can perform the CRUD operation.

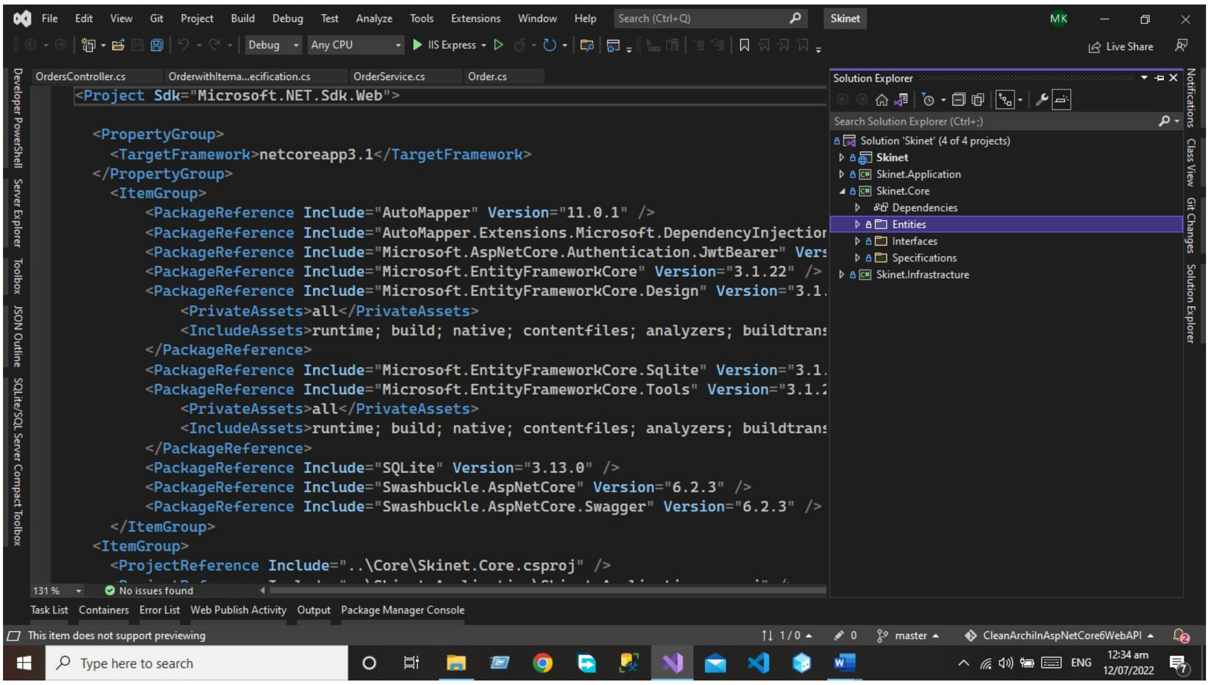
In our project, we will add the store context class that will handle the session with our database.

Code of the Database Context Class

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage.ValueConversion;
using Skinet.Core.Entities;
using Skinet.Core.Entities.OrderAggregate;
using System;
using System.Linq;
using System.Reflection;

namespace Skinet.Infrastracture.Data
{
    public class StoreContext : DbContext
    {
        public StoreContext(DbContextOptions<StoreContext> options) : base(options)
        {

        }
        public DbSet<Products> Products { get; set; }
        public DbSet<ProductType> ProductTypes { get; set; }
        public DbSet<ProductBrand> ProductBrands { get; set; }
        public DbSet<Order> Orders { get; set; }
        public DbSet<DeliveryMethod> DeliveryMethods { get; set; }
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);
            modelBuilder.ApplyConfigurationsFromAssembly(Assembly.GetExecutingAssembly());
            if(Database.ProviderName =="Microsoft.EntityFramework.Sqlite")
            {
                foreach (var entity in modelBuilder.Model.GetEntityTypes())
                {
                    var properties = entity.ClrType.GetProperties()
                        .Where(p => p.PropertyType == typeof(decimal));
                    var dateandtimepropertise = entity.ClrType.GetProperties()
                        .Where(t => t.PropertyType == typeof(DateTimeOffset));
                    foreach (var property in properties)
                    {
                        modelBuilder.Entity(entity.Name).Property(property.Name)
                            .HasConversion<double>();
                    }
                    foreach (var property in dateandtimepropertise)
                    {
                        modelBuilder.Entity(entity.Name).Property(property.Name)
                            .HasConversion(new DateTimeOffsetToBinaryConverter());
                    }
                }
            }
        }
    }
}
```

Models

First, you need to add the Models folder that will be used to create the database entities. In the Models folder, we will create the following database entities



```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>
    <ItemGroup>
        <PackageReference Include="AutoMapper" Version="11.0.1" />
        <PackageReference Include="AutoMapper.Extensions.Microsoft.DependencyInjectior
        <PackageReference Include="Microsoft.AspNetCore.Authentication.JwtBearer" Vers
        <PackageReference Include="Microsoft.EntityFrameworkCore" Version="3.1.22" />
        <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="3.1.
            <PrivateAssets>all</PrivateAssets>
            <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtrans
        </PackageReference>
        <PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite" Version="3.1.
        <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="3.1.2
            <PrivateAssets>all</PrivateAssets>
            <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtrans
        </PackageReference>
        <PackageReference Include="SQLite" Version="3.13.0" />
        <PackageReference Include="Swashbuckle.AspNetCore" Version="6.2.3" />
        <PackageReference Include="Swashbuckle.AspNetCore.Swagger" Version="6.2.3" />
    </ItemGroup>
  <ItemGroup>
    <ProjectReference Include="..\Core\Skinet.Core.csproj" />
```

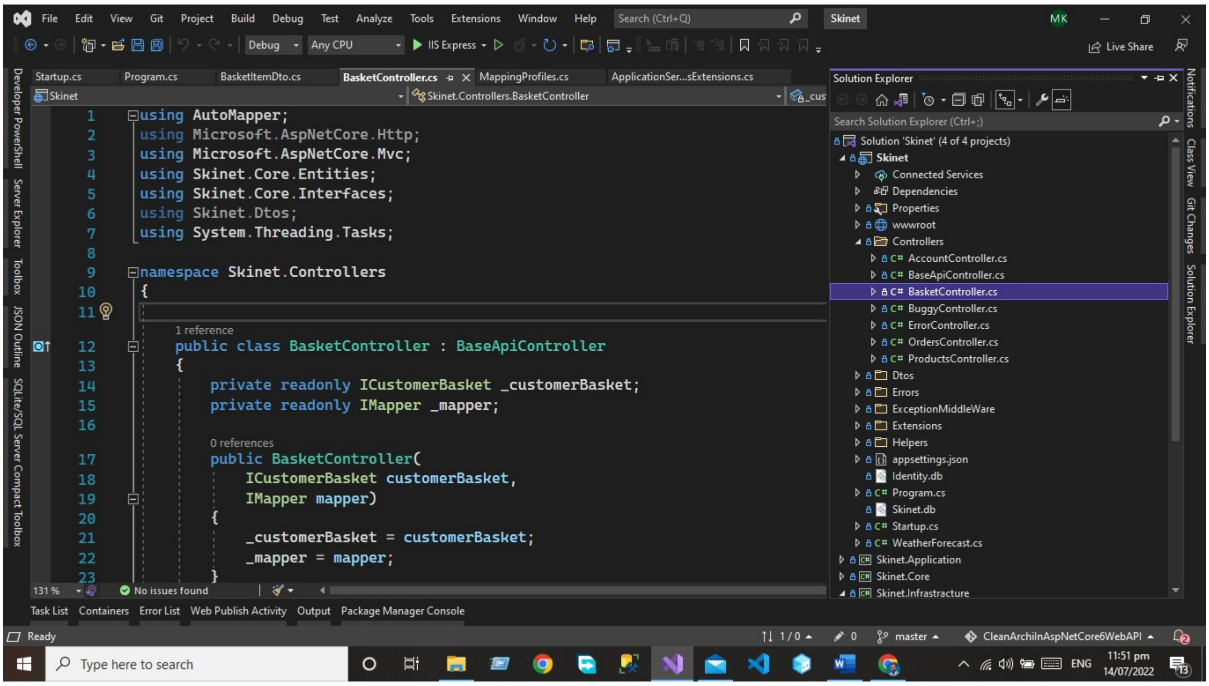
Code Example of Models

```
namespace MonolithicApplication.Models
{
    public class Employee
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Address { get; set; }
        public string Country { get; set; }
    }
}
```

Controllers

Controllers are used to handle the HTTP request. Now we need to add the student controller that will interact will our service layer and display the data to the users.

Code Example of Controller



```
using AutoMapper;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Skinet.Core.Entities;
using Skinet.Core.Interfaces;
using Skinet.Dtos;
using System.Threading.Tasks;

namespace Skinet.Controllers
{

    public class BasketController : BaseApiController
    {
        private readonly ICustomerBasket _customerBasket;
        private readonly IMapper _mapper;

        public BasketController(
            ICustomerBasket customerBasket,
            IMapper mapper)
        {
            _customerBasket = customerBasket;
            _mapper = mapper;
        }
```

```
using AutoMapper;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Skinet.Core.Entities;
using Skinet.Core.Interfaces;
using Skinet.Dtos;
using System.Threading.Tasks;

namespace Skinet.Controllers
{

  public class BasketController : BaseApiController
  {
    private readonly ICustomerBasket _customerBasket;
    private readonly IMapper _mapper;

    public BasketController(
      ICustomerBasket customerBasket,
      IMapper mapper)
    {
      _customerBasket = customerBasket;
      _mapper = mapper;
    }
    [HttpGet(nameof(GetBasketElement))]
    public async Task<ActionResult<CustomerBasket>> GetBasketElement([FromQuery]string Id)
    {
      var basketelements = await _customerBasket.GetBasketAsync(Id);
      return Ok(basketelements??new CustomerBasket(Id));
    }

    [HttpPost(nameof(UpdateProduct))]
    public async Task<ActionResult<CustomerBasket>> UpdateProduct(CustomerBasket product)
```

```
        {
            //var customerbasket = _mapper.Map<CustomerbasketDto, CustomerBasket>(product);
            var data =  await _customerBasket.UpdateBasketAsync(product);
            return Ok(data);
        }
        [HttpDelete(nameof(DeleteProduct))]
        public async Task DeleteProduct(string Id)
        {
            await _customerBasket.DeleteBasketAsync(Id);
        }
    }
}
```

Services Folder

This folder will be used to add the custom services to our system and lets us create some custom services for our project so that our concept will be clear about Custom services. All the custom services will inherit the I Custom services interface using that interface we will add the CRUD Operation in our system.

Code Example of I Custom Interface

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading.Tasks;

namespace Skynet.Application.ICustomServices
{
    public interface ICustomService<T>
    {
        IEnumerable<T> GetAll();
        void FindById(int Id);
        void Insert(T entity);
        Task<T> Update(T entity);
        void Delete(T entity);
    }
}
```

Code Example of Custom Service

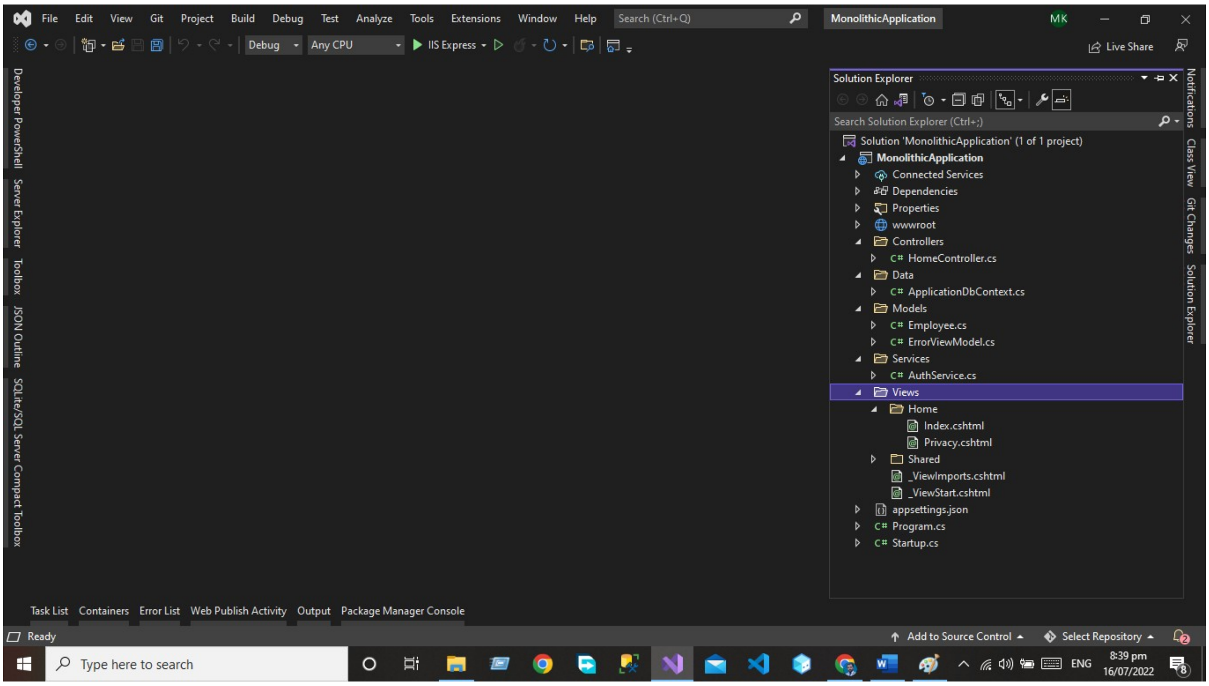
```
using DomainLayer.Models;
using RepositoryLayer.IRepository;
using ServiceLayer.ICustomServices;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ServiceLayer.CustomServices
{
    public class DepartmentsService : ICustomService<Departments>
    {
        private readonly IRepository<Departments> _studentRepository;
        public  DepartmentsService(IRepository<Departments>  studentRepository)
        {
            _studentRepository = studentRepository;
```

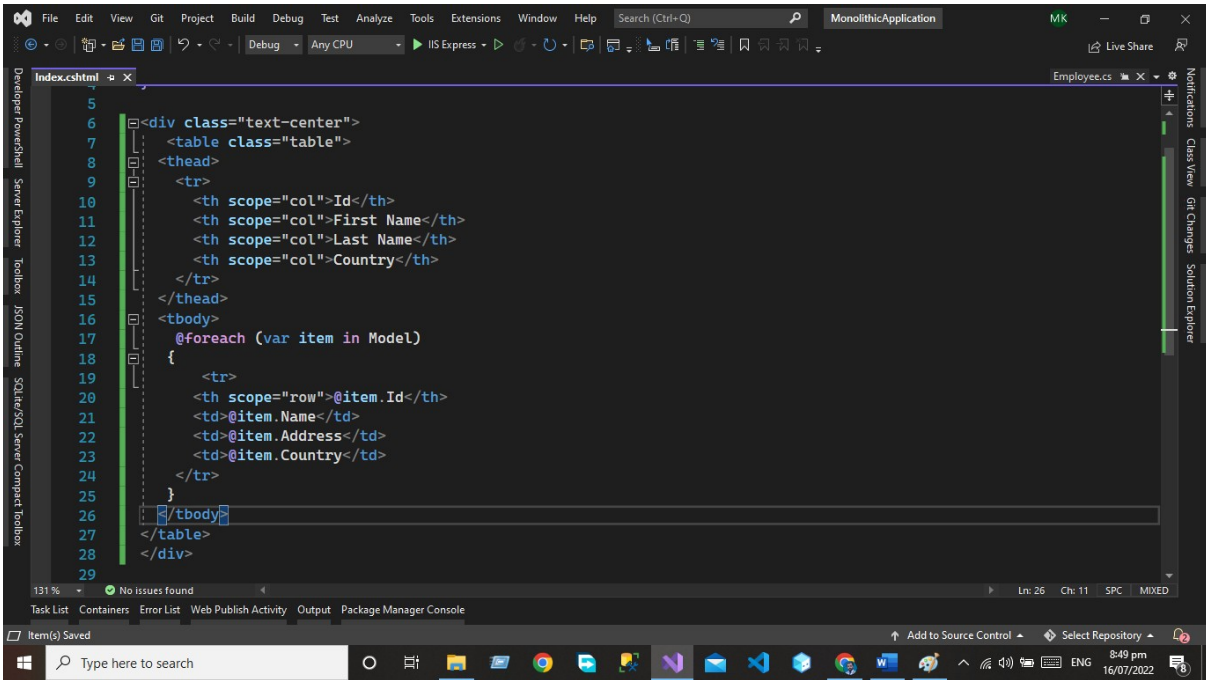
```
}
public void Delete(Departments entity)
{
    try
    {
        if (entity != null)
        {
            _studentRepository.Delete(entity);
            _studentRepository.SaveChanges();
        }
    }
    catch (Exception)
    {

        throw;
    }
}

public Departments Get(int Id)
{
    try
    {
        var obj = _studentRepository.Get(Id);
        if (obj != null)
        {
            return obj;
        }
        else
        {
            return null;
        }

    }
    catch (Exception)
    {

        throw;
    }
}

public IEnumerable<Departments> GetAll()
{
    try
    {
        var obj = _studentRepository.GetAll();
        if (obj != null)
        {
            return obj;
        }
        else
        {
            return null;
        }
    }
    catch (Exception)
    {

        throw;
    }
}

public void Insert(Departments entity)
{
```

```
            try
            {
                if (entity != null)
                {
                    _studentRepository.Insert(entity);
                    _studentRepository.SaveChanges();
                }
            }
            catch (Exception)
            {

                throw;
            }
        }

        public void Remove(Departments entity)
        {
            try
            {
                if (entity != null)
                {
                    _studentRepository.Remove(entity);
                    _studentRepository.SaveChanges();
                }
            }
            catch (Exception)
            {

                throw;
            }
        }
        public void Update(Departments entity)
        {
            try
            {
                if (entity != null)
                {
                    _studentRepository.Update(entity);
                    _studentRepository.SaveChanges();
                }
            }
            catch (Exception)
            {

                throw;
            }
        }
    }
}
```

View Folder

In the view folder, we write the code for View that present the data to the user.





```

<div class="text-center">
   <table class="table">
  <thead>
    <tr>
      <th scope="col">Id</th>
      <th scope="col">First Name</th>
      <th scope="col">Last Name</th>
      <th scope="col">Country</th>
    </tr>
  </thead>
  <tbody>
    @foreach (var item in Model)
   {
      <tr>
      <th scope="row">@item.Id</th>
      <td>@item.Name</td>
      <td>@item.Address</td>
      <td>@item.Country</td>
    </tr>
   }
  </tbody>
</table>
</div>

```

Code Example of View

```
@model IEnumerable<MonolithicApplication.Models.Employee>
@{
   ViewData["Title"] = "Home Page";
}

<div class="text-center">
   <table class="table">
  <thead>
    <tr>
      <th scope="col">Id</th>
      <th scope="col">First Name</th>
      <th scope="col">Last Name</th>
      <th scope="col">Country</th>
     </tr>
  </thead>
  <tbody>
     @foreach (var item in Model)
   {
     <tr>
      <th scope="row">@item.Id</th>
      <td>@item.Name</td>
      <td>@item.Address</td>
      <td>@item.Country</td>
   </tr>
   }
  </tbody>
</table>
</div>
```

Conclusion

If we want to develop a small application, then best suitable architecture is monolithic. But if we want to develop a complex application then we need microservices architecture or any other architecture that decouples the application.

Note: We will develop the complete Monolithic application in Three Tier Architecture.

Complete GitHub project URL

Click here for complete project access

Chapter 2 - Three Tier Architecture

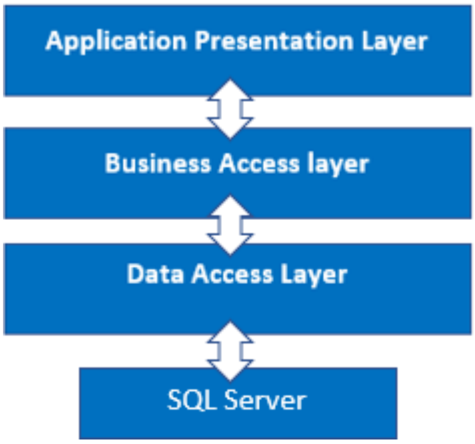
- ✓ Introduction
- ✓ Project Structure
- ✓ Presentation Layer (PL)
- ✓ Business Access Layer (BAL)
- ✓ Data Access Layer (DAL)
- ✓ Add the References to the Project
- ✓ Data Access Layer
- ✓ Contacts
- ✓ Data
- ✓ Migrations
- ✓ Models
- ✓ Repositors
- ✓ Business Access Layer
- ✓ Services
- ✓ Presentation layer
- ✓ Advantages of 3-Tier Architecture
- ✓ Dis-Advantages of 3-Tier Architecture
- ✓ Conclusion
- ✓ Complete GitHub Project URL

Introduction

In this chapter, we'll look at three-tier architecture and how to incorporate the Data Access Layer and Business Access Layer into a project, as well as how these layers interact.

Project Structure

We use a three-tier architecture in this project, with a data access layer, a business access layer, and an application presentation layer.



Presentation Layer (PL)

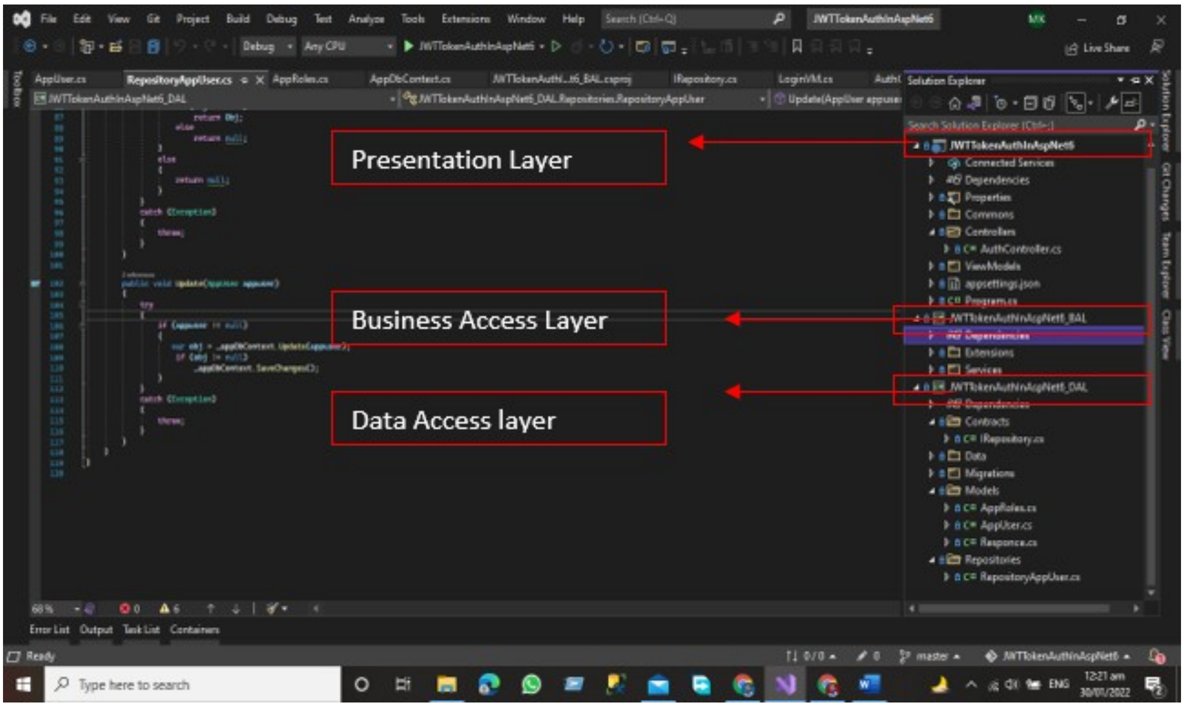
The Presentation layer is the top-most layer of the 3-tier architecture, and its major role is to display the results to the user, or to put it another way, to present the data that we acquire from the business access layer and offer the results to the front-end user.

Business Access Layer (BAL)

The logic layer interacts with the data access layer and the presentation layer to process the activities that lead to logical decisions and assessments. This layer's primary job is to process data between other layers.

Data Access Layer (DAL)

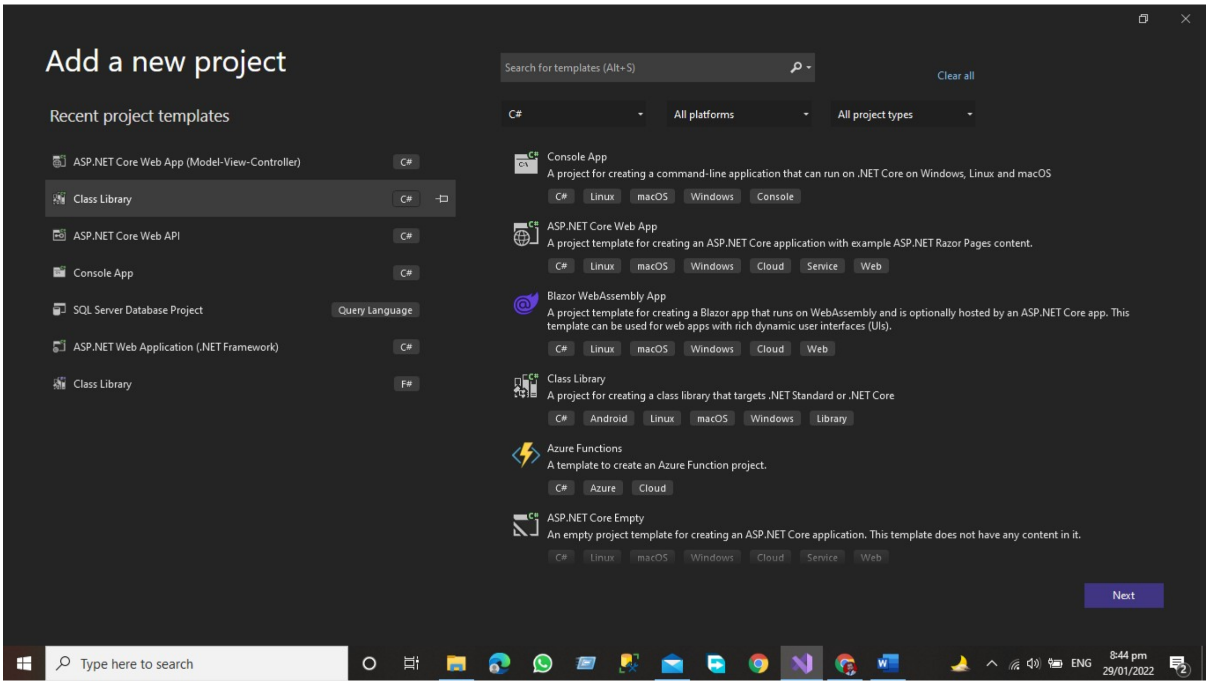
The main function of this layer is to access and store the data from the database and the process of the data to business access layer data goes to the presentation layer against user request.



```
                    return Obj;
                else
                    return null;
            }
            else
            {
                return null;
            }
        }
        catch (Exception)
        {
            throw;
        }
    }

    public void Update(AppUser appuser)
    {
        try
        {
            if (appuser != null)
            {
                var obj = _appDbContext.Update(appuser);
                if (obj != null)
                    _appDbContext.SaveChanges();
            }
        }
        catch (Exception)
        {
            throw;
        }
    }
}
}
```

Steps to Follow for configuring these layer

- Add the Class Library project of Asp.net for Data Access Layer
- Right Click on the project and then go to the add the new project window and then add the Asp.net Core class library project.



Add a new project

Recent project templates

- ASP.NET Core Web App (Model-View-Controller) — C#
- Class Library — C#
- ASP.NET Core Web API — C#
- Console App — C#
- SQL Server Database Project — Query Language
- ASP.NET Web Application (.NET Framework) — C#
- Class Library — F#

Search for templates (Alt+S)

Clear all

C# | All platforms | All project types

Console App
A project for creating a command-line application that can run on .NET Core on Windows, Linux and macOS
C# Linux macOS Windows Console

ASP.NET Core Web App
A project template for creating an ASP.NET Core application with example ASP.NET Razor Pages content.
C# Linux macOS Windows Cloud Service Web

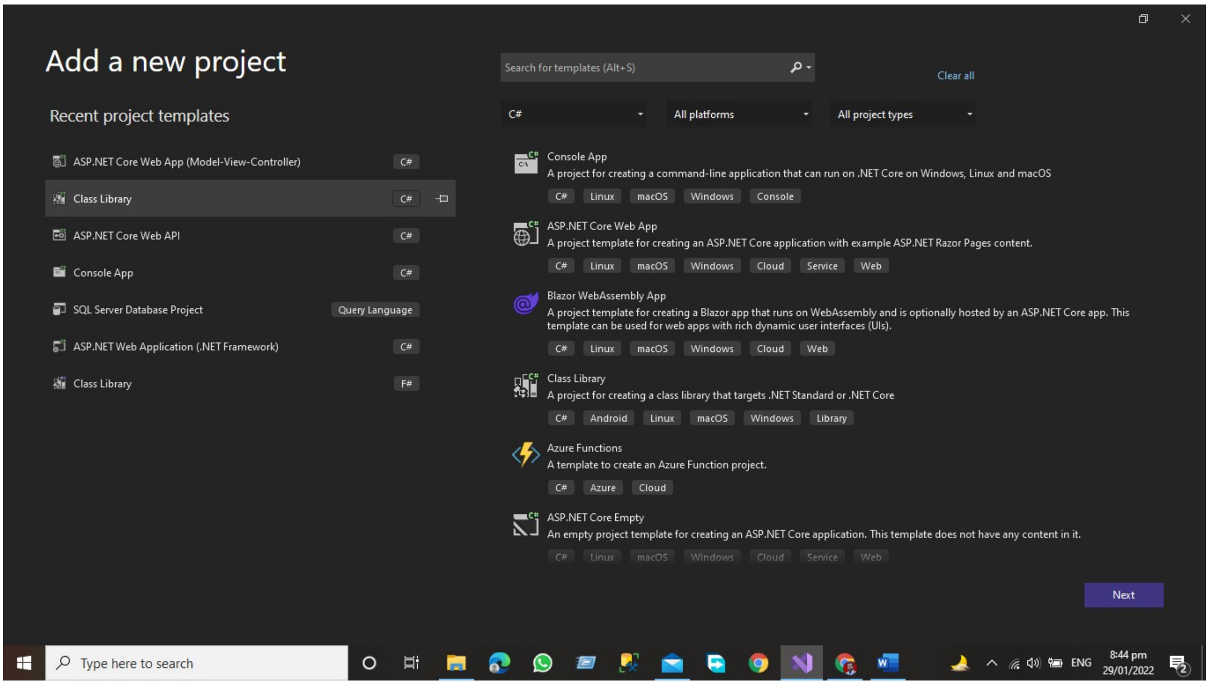
Blazor WebAssembly App
A project template for creating a Blazor app that runs on WebAssembly and is optionally hosted by an ASP.NET Core app. This template can be used for web apps with rich dynamic user interfaces (UIs).
C# Linux macOS Windows Cloud Web

Class Library
A project for creating a class library that targets .NET Standard or .NET Core
C# Android Linux macOS Windows Library

Azure Functions
A template to create an Azure Function project.
C# Azure Cloud

ASP.NET Core Empty
An empty project template for creating an ASP.NET Core application. This template does not have any content in it.
C# Linux macOS Windows Cloud Service Web

- After Adding the Data Access layer project now, we will add the Business access layer folder
- Add the Class library project of Asp.Net Core for Business Access
- Right Click on the project and then go to the add the new project window and then add the Asp.net Core class library project.



Add a new project

Recent project templates

- ASP.NET Core Web App (Model-View-Controller) C#
- Class Library C#
- ASP.NET Core Web API C#
- Console App C#
- SQL Server Database Project Query Language
- ASP.NET Web Application (.NET Framework) C#
- Class Library F#

Search for templates (Alt+S)

Clear all

C# · All platforms · All project types

Console App
A project for creating a command-line application that can run on .NET Core on Windows, Linux and macOS
C# Linux macOS Windows Console

ASP.NET Core Web App
A project template for creating an ASP.NET Core application with example ASP.NET Razor Pages content.
C# Linux macOS Windows Cloud Service Web

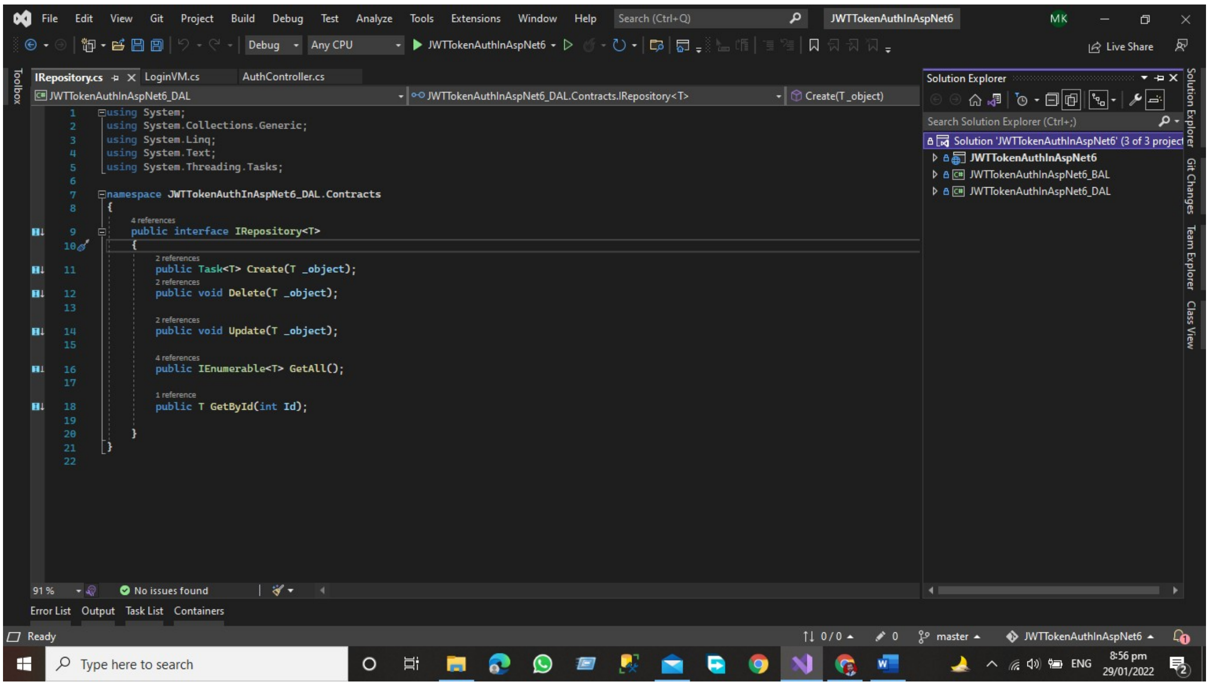
Blazor WebAssembly App
A project template for creating a Blazor app that runs on WebAssembly and is optionally hosted by an ASP.NET Core app. This template can be used for web apps with rich dynamic user interfaces (UIs).
C# Linux macOS Windows Cloud Web

Class Library
A project for creating a class library that targets .NET Standard or .NET Core
C# Android Linux macOS Windows Library

Azure Functions
A template to create an Azure Function project.
C# Azure Cloud

ASP.NET Core Empty
An empty project template for creating an ASP.NET Core application. This template does not have any content in it.
C# Linux macOS Windows Cloud Service Web

After adding the business access layer and data access layer, we must first add the references of the Data Access layer and Business Access layer, so that the project structure can be seen.



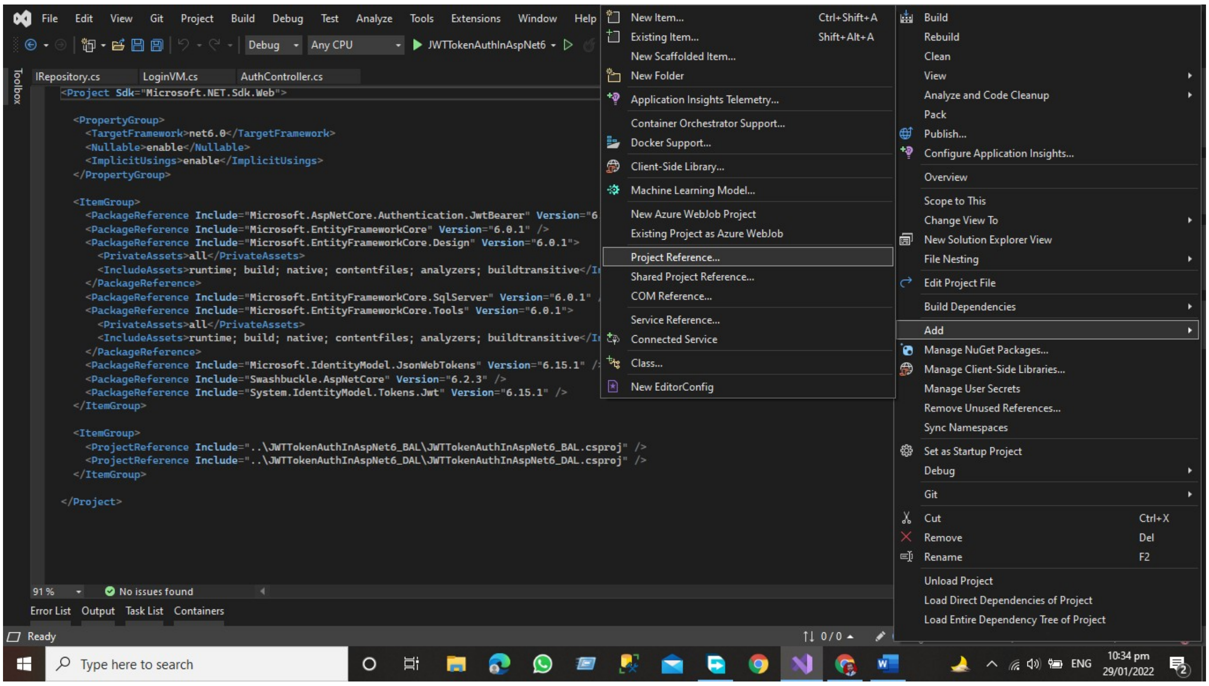
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace JWTTokenAuthInAspNet6_DAL.Contracts
{
    public interface IRepository<T>
    {
        public Task<T> Create(T _object);
        public void Delete(T _object);

        public void Update(T _object);

        public IEnumerable<T> GetAll();

        public T GetById(int Id);

    }
}
```

As you can see, our project now has a Presentation layer, Data Access Layer, and Business Access layer.

Add the References to the Project

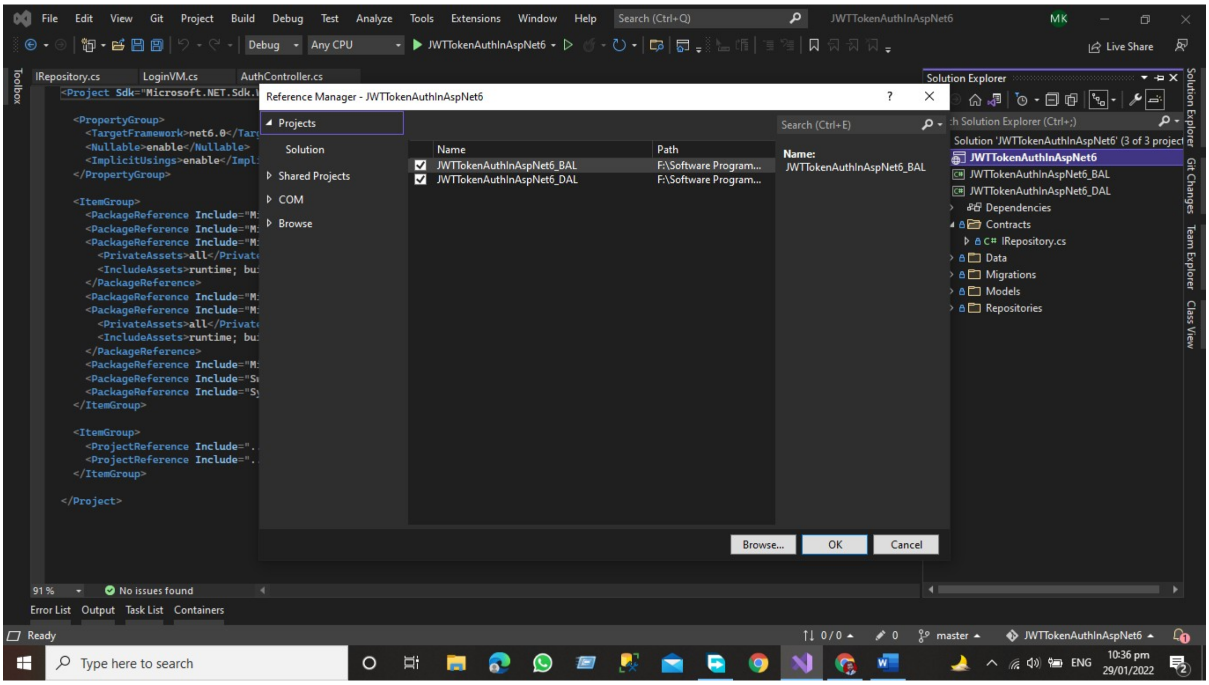
Now, in the Presentation Layer, add the references to the Data Access Layer and the Business Access Layer.

Right Click on the Presentation layer and then click on add => project Reference



```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Authentication.JwtBearer" Version="6
    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="6.0.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="6.0.1">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</I
    </PackageReference>
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="6.0.1" /
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="6.0.1">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</I
    </PackageReference>
    <PackageReference Include="Microsoft.IdentityModel.JsonWebTokens" Version="6.15.1" /
    <PackageReference Include="Swashbuckle.AspNetCore" Version="6.2.3" />
    <PackageReference Include="System.IdentityModel.Tokens.Jwt" Version="6.15.1" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include="..\JWTTokenAuthInAspNet6_BAL\JWTTokenAuthInAspNet6_BAL.csproj" />
    <ProjectReference Include="..\JWTTokenAuthInAspNet6_DAL\JWTTokenAuthInAspNet6_DAL.csproj" />
  </ItemGroup>

</Project>
```

Add the References by Checking both the checkboxes

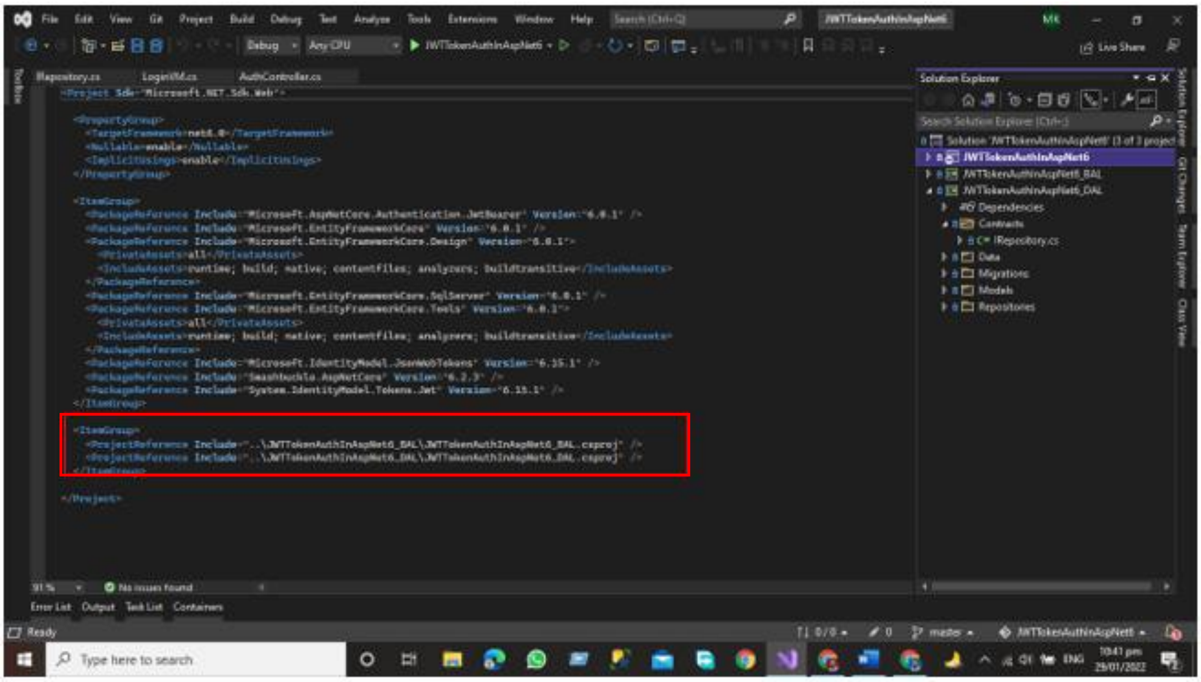


Reference Manager - JWTokenAuthInAspNet6

	Name	Path
☑	JWTokenAuthInAspNet6_BAL	F:\Software Program...
☑	JWTokenAuthInAspNet6_DAL	F:\Software Program...

Name: JWTokenAuthInAspNet6_BAL

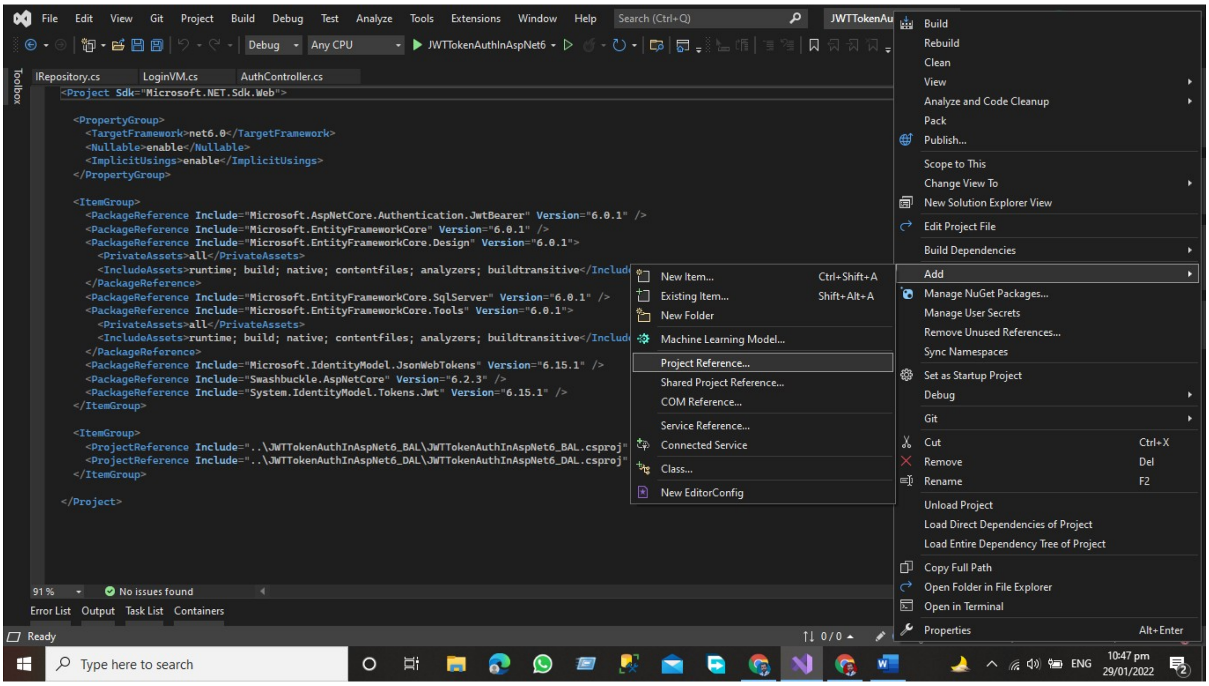
References have been added for both DAL and BAL



```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Authentication.JwtBearer" Version="6.0.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="6.0.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="6.0.1">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
    </PackageReference>
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="6.0.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="6.0.1">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
    </PackageReference>
    <PackageReference Include="Microsoft.IdentityModel.JsonWebTokens" Version="6.15.1" />
    <PackageReference Include="Swashbuckle.AspNetCore" Version="6.2.3" />
    <PackageReference Include="System.IdentityModel.Tokens.Jwt" Version="6.15.1" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include="..\JWTokenAuthInAspNet6_BAL\JWTokenAuthInAspNet6_BAL.csproj" />
    <ProjectReference Include="..\JWTokenAuthInAspNet6_DAL\JWTokenAuthInAspNet6_DAL.csproj" />
  </ItemGroup>

</Project>
```

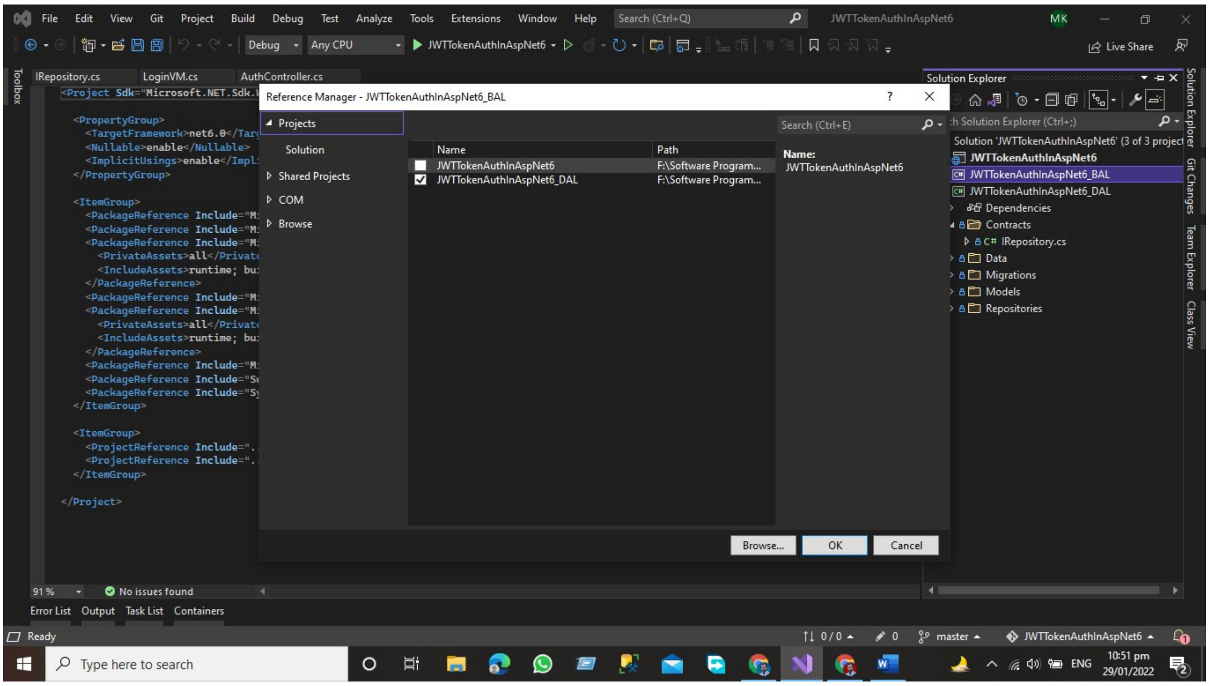
Now we'll add the Data Access layer project references to the Business layer.

➢ Right Click on the Business access layer and then Click Add Button => Project References



```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Authentication.JwtBearer" Version="6.0.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="6.0.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="6.0.1">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</Includ
    </PackageReference>
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="6.0.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="6.0.1">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</Includ
    </PackageReference>
    <PackageReference Include="Microsoft.IdentityModel.JsonWebTokens" Version="6.15.1" />
    <PackageReference Include="Swashbuckle.AspNetCore" Version="6.2.3" />
    <PackageReference Include="System.IdentityModel.Tokens.Jwt" Version="6.15.1" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include="..\JWTTokenAuthInAspNet6_BAL\JWTTokenAuthInAspNet6_BAL.csproj"
    <ProjectReference Include="..\JWTTokenAuthInAspNet6_DAL\JWTTokenAuthInAspNet6_DAL.csproj"
  </ItemGroup>

</Project>
```

“Remember that we have included the DAL and BAL references in the project, so don't add the Presentation layer references again in the business layer. We only need the DAL Layer references in the business layer”

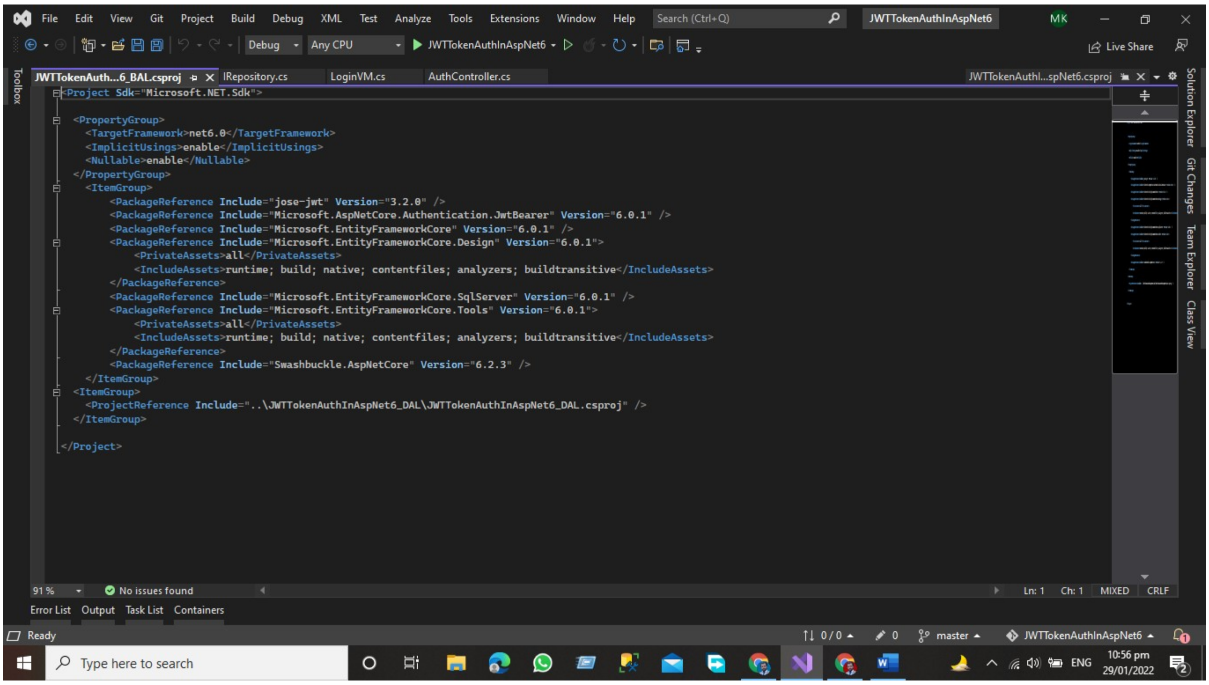


Reference Manager - JWTTokenAuthInAspNet6_BAL

	Name	Path
□	JWTTokenAuthInAspNet6	F:\Software Program...
☑	JWTTokenAuthInAspNet6_DAL	F:\Software Program...

Name:
JWTTokenAuthInAspNet6

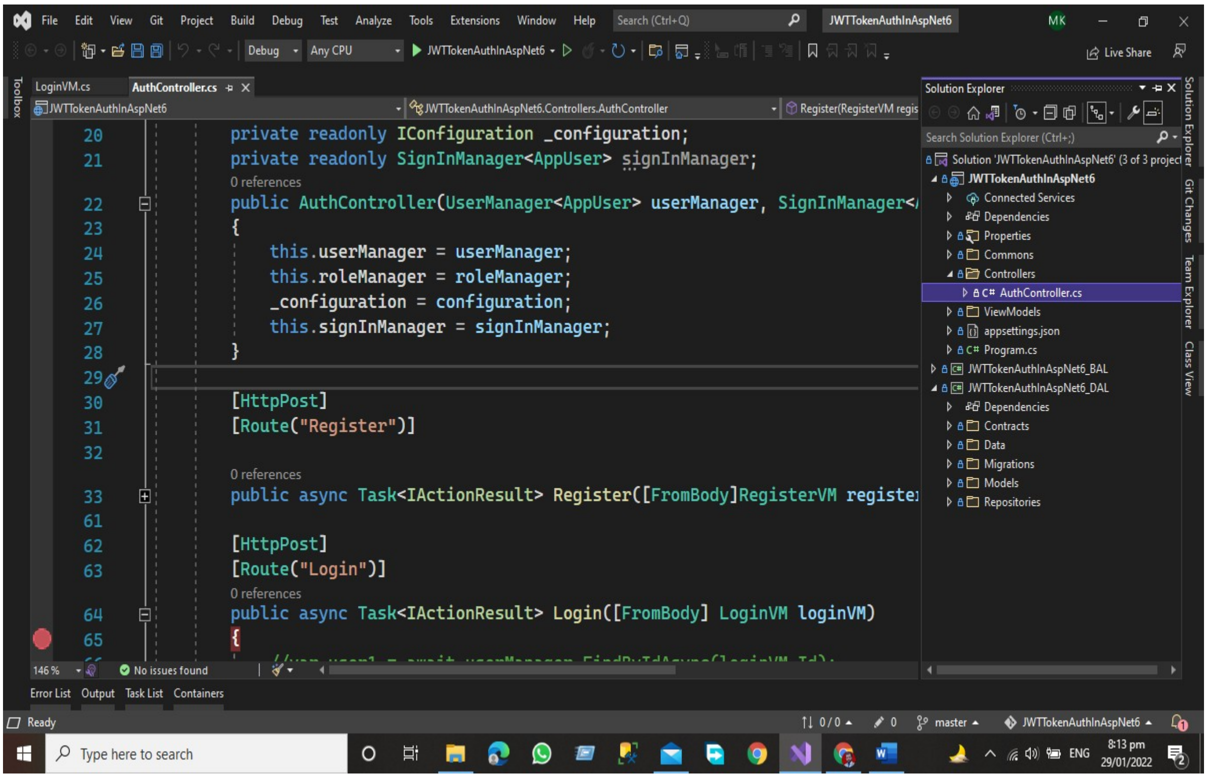
Project References of DAL Has been Added to BAL Layer



```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
    <ItemGroup>
        <PackageReference Include="jose-jwt" Version="3.2.0" />
        <PackageReference Include="Microsoft.AspNetCore.Authentication.JwtBearer" Version="6.0.1" />
        <PackageReference Include="Microsoft.EntityFrameworkCore" Version="6.0.1" />
        <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="6.0.1">
            <PrivateAssets>all</PrivateAssets>
            <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
        </PackageReference>
        <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="6.0.1" />
        <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="6.0.1">
            <PrivateAssets>all</PrivateAssets>
            <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
        </PackageReference>
        <PackageReference Include="Swashbuckle.AspNetCore" Version="6.2.3" />
    </ItemGroup>
  <ItemGroup>
    <ProjectReference Include="..\JWTTokenAuthInAspNet6_DAL\JWTTokenAuthInAspNet6_DAL.csproj" />
  </ItemGroup>

</Project>
```

Data Access Layer

The Data Access Layer contains methods that assist the Business Access Layer in writing business logic, whether the functions are linked to accessing or manipulating data. The Data Access Layer's major goal is to interface with the database and the Business Access Layer in our project.



```
private readonly IConfiguration _configuration;
private readonly SignInManager<AppUser> signInManager;
0 references
public AuthController(UserManager<AppUser> userManager, SignInManager<
{
    this.userManager = userManager;
    this.roleManager = roleManager;
    _configuration = configuration;
    this.signInManager = signInManager;
}

[HttpPost]
[Route("Register")]

0 references
public async Task<IActionResult> Register([FromBody]RegisterVM registe

[HttpPost]
[Route("Login")]
0 references
public async Task<IActionResult> Login([FromBody] LoginVM loginVM)
{
```

the structure will be like this.

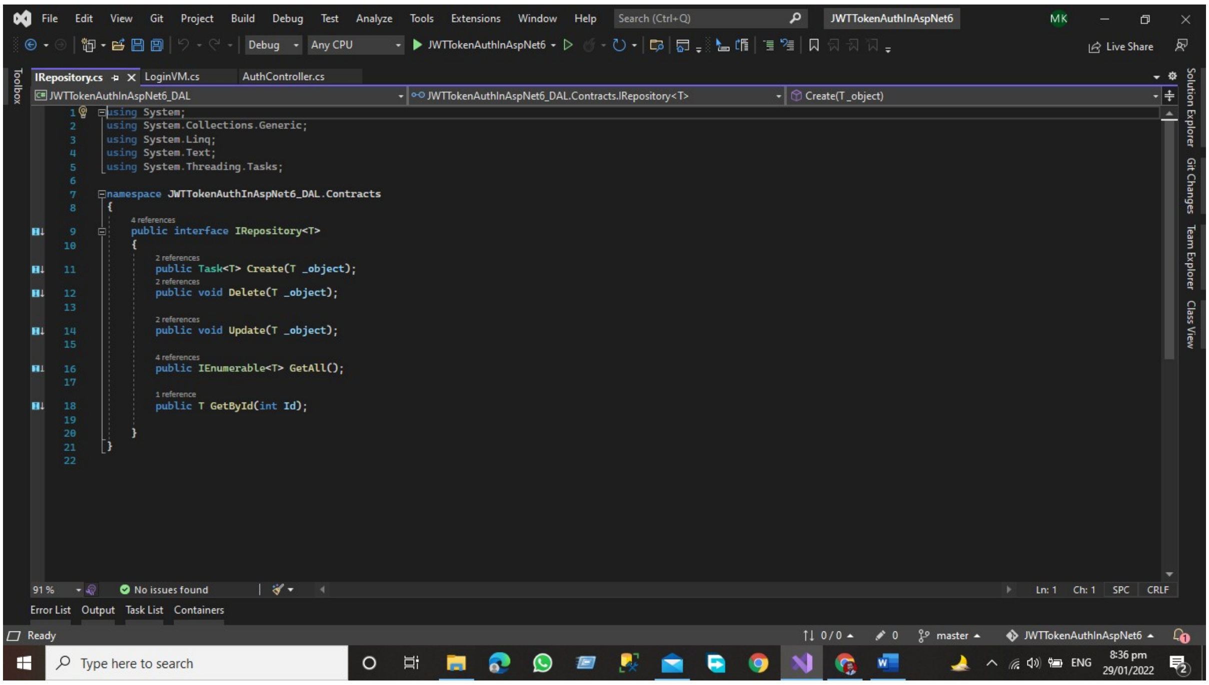
In this Layer we have the Following Folders Add these folders to your Data access layer

- Contacts
- Data
- Migrations
- Models
- Repositories

Contacts

In the Contract Folder, we define the interface that has the following function that performs the desired functionalities with the database like

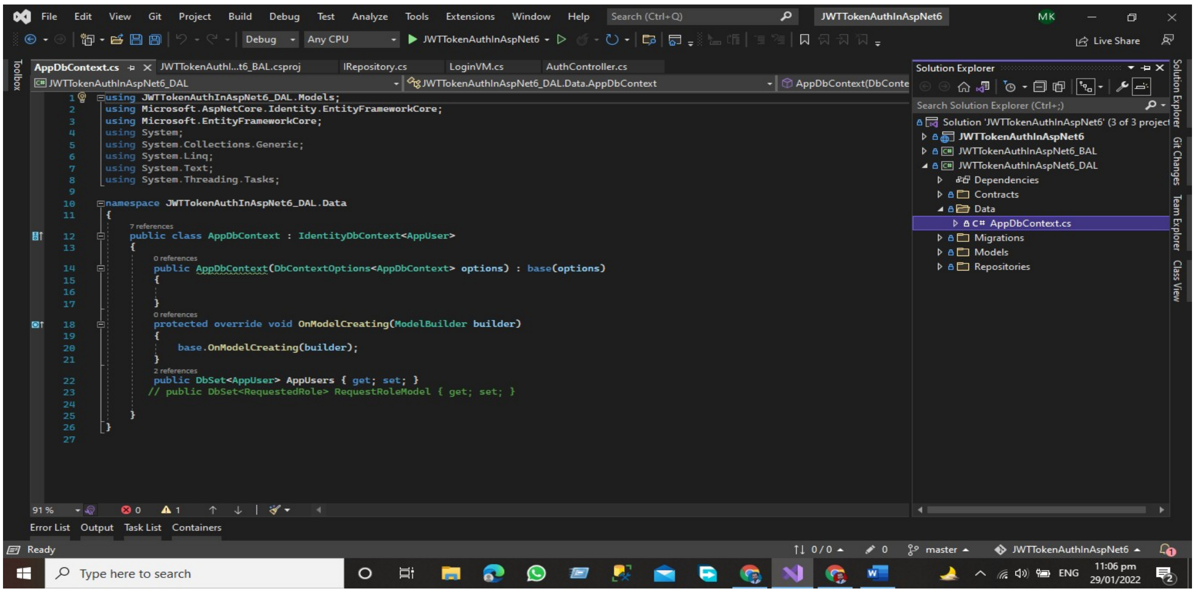
```
using System;
using System.Collections.Generic;
using System. Linq;
using System. Text;
using System.Threading.Tasks;

namespace JWTTokenAuthInAspNet6_DAL.Contracts
{
    public interface IRepository<T>
    {
        public Task<T> Create(T _object);
        public void Delete(T _object);

        public void Update(T _object);

        public IEnumerable<T> GetAll();

        public T GetById(int Id);

    }
}
```



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace JWTTokenAuthInAspNet6_DAL.Contracts
{
    4 references
    public interface IRepository<T>
    {
        2 references
        public Task<T> Create(T _object);
        2 references
        public void Delete(T _object);

        2 references
        public void Update(T _object);

        4 references
        public IEnumerable<T> GetAll();

        1 reference
        public T GetById(int Id);

    }
}
```

Data

In the Data folder, we have Db Context Class this class is very important for accessing the data from the database.



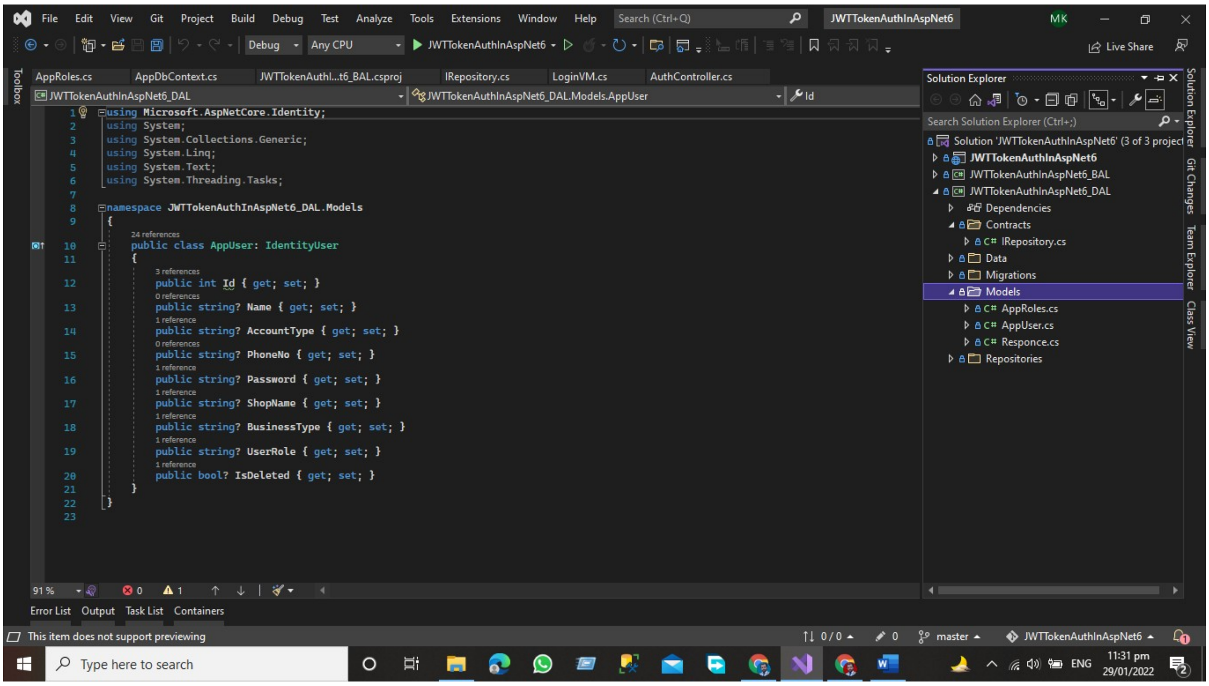
```
using JWTTokenAuthInAspNet6_DAL.Models;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace JWTTokenAuthInAspNet6_DAL.Data
{
    public class AppDbContext : IdentityDbContext<AppUser>
    {
        public AppDbContext(DbContextOptions<AppDbContext> options) : base(options)
        {

        }
        protected override void OnModelCreating(ModelBuilder builder)
        {
            base.OnModelCreating(builder);
        }
        public DbSet<AppUser> AppUsers { get; set; }
      // public DbSet<RequestedRole> RequestRoleModel { get; set; }

    }
}
```

Migrations

The Migration Folder contains information on all the migrations we performed during the construction of this project. The Migration Folder contains information on all the migrations we performed during the construction of this project.

Models

Our application models, which contain domain-specific data, and business logic models, which represent the structure of the data as public attributes, business logic, and methods, are stored in the Model folder.

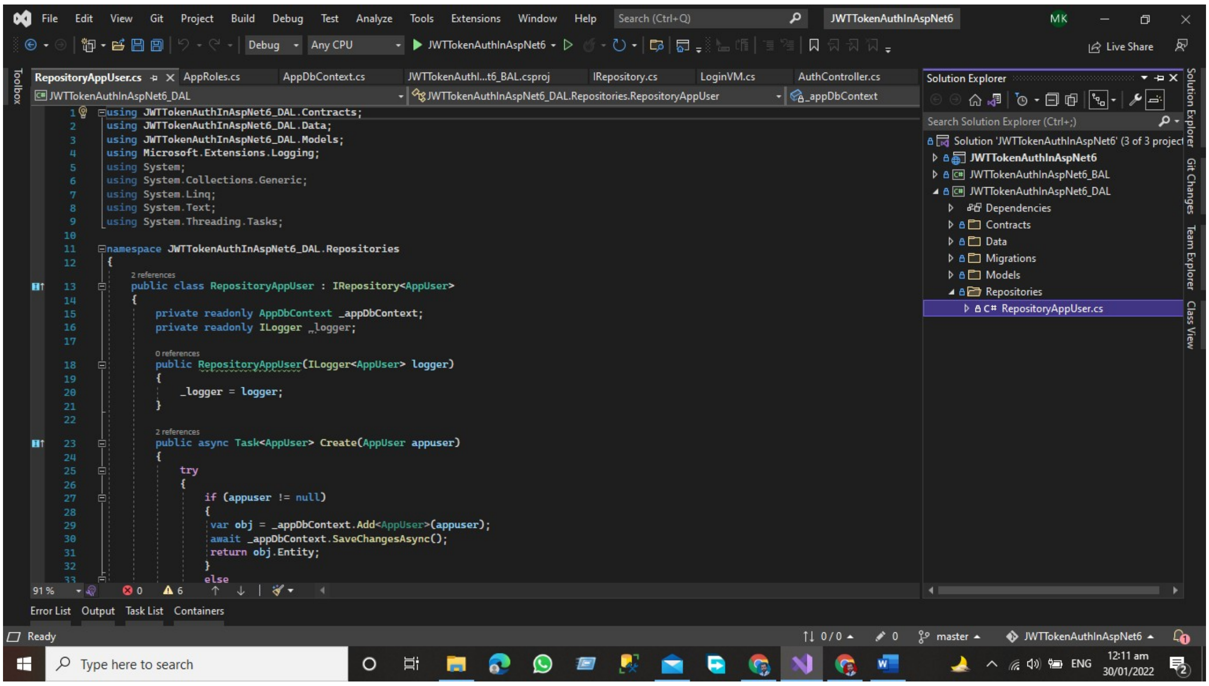


```
using Microsoft.AspNetCore.Identity;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace JWTTokenAuthInAspNet6_DAL.Models
{
    public class AppUser: IdentityUser
    {
        public int Id { get; set; }
        public string? Name { get; set; }
        public string? AccountType { get; set; }
        public string? PhoneNo { get; set; }
        public string? Password { get; set; }
        public string? ShopName { get; set; }
        public string? BusinessType { get; set; }
        public string? UserRole { get; set; }
        public bool? IsDeleted { get; set; }
    }
}
```

```
using Microsoft.AspNetCore.Identity;
using System;
using System.Collections.Generic;
using System. Linq;
using System. Text;
using System.Threading.Tasks;

namespace JWTTokenAuthInAspNet6_DAL.Models
{
   public class AppUser: IdentityUser
   {
      public int Id { get; set; }
      public string? Name { get; set; }
      public string? AccountType { get; set; }
      public string? PhoneNo { get; set; }
      public string? Password { get; set; }
      public string? ShopName { get; set; }
      public string? BusinessType { get; set; }
      public string? UserRole { get; set; }
      public bool? IsDeleted { get; set; }
   }
}
```

Repositors

In the Repository folder, we add the repository classes against each model. We write the CRUD function that communicates with the database using the entity framework. We add the repository class that inherits our Interface that is present in our contract folder.



```
using JWTTokenAuthInAspNet6_DAL.Contracts;
using JWTTokenAuthInAspNet6_DAL.Data;
using JWTTokenAuthInAspNet6_DAL.Models;
using Microsoft.Extensions.Logging;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace JWTTokenAuthInAspNet6_DAL.Repositories
{
    2 references
    public class RepositoryAppUser : IRepository<AppUser>
    {
        private readonly AppDbContext _appDbContext;
        private readonly ILogger _logger;

        0 references
        public RepositoryAppUser(ILogger<AppUser> logger)
        {
            _logger = logger;
        }

        2 references
        public async Task<AppUser> Create(AppUser appuser)
        {
            try
            {
                if (appuser != null)
                {
                    var obj = _appDbContext.Add<AppUser>(appuser);
                    await _appDbContext.SaveChangesAsync();
                    return obj.Entity;
                }
                else
```

```
using JWTTokenAuthInAspNet6_DAL.Contracts;
using JWTTokenAuthInAspNet6_DAL.Data;
using JWTTokenAuthInAspNet6_DAL.Models;
using Microsoft.Extensions.Logging;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace JWTTokenAuthInAspNet6_DAL.Repositories
{
    public class RepositoryAppUser : IRepository<AppUser>
    {
        private readonly AppDbContext _appDbContext;
        private readonly ILogger _logger;

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

        public async Task<AppUser> Create(AppUser appuser)
        {
            try
            {
```

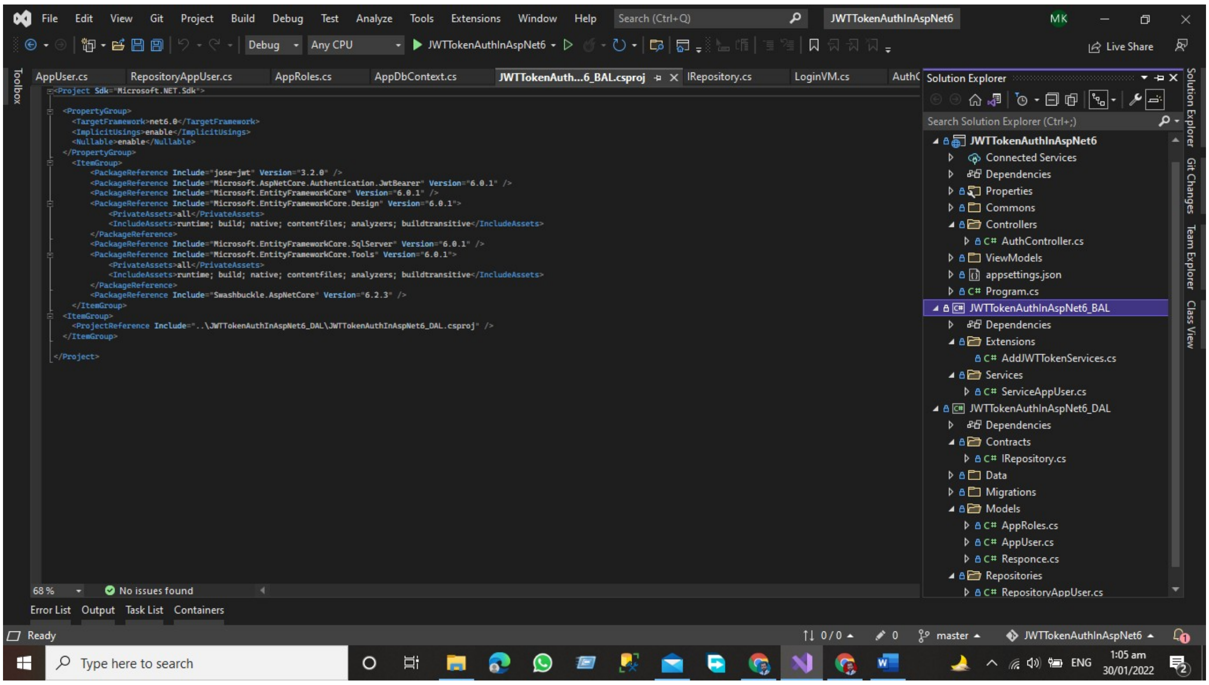
```
        if (appuser != null)
        {
         var obj = _appDbContext.Add<AppUser>(appuser);
         await _appDbContext.SaveChangesAsync();
         return obj.Entity;
        }
        else
        {
            return null;
        }
    }
    catch (Exception)
    {
        throw;
    }
}

public void Delete(AppUser appuser)
{
    try
    {
        if(appuser!=null)
        {
            var obj = _appDbContext.Remove(appuser);
            if (obj!=null)
            {
                _appDbContext.SaveChangesAsync();
            }
        }
    }
    catch (Exception)
    {
        throw;
    }
}

public IEnumerable<AppUser> GetAll()
{
    try
    {
        var obj = _appDbContext.AppUsers.ToList();
        if (obj != null)
```

```
            return obj;
        else
            return null;
    }
    catch (Exception)
    {
        throw;
    }
}

public AppUser GetById(int Id)
{
    try
    {
        if(Id!=null)
        {
          var Obj = _appDbContext.AppUsers.FirstOrDefault(x => x.Id == Id);
            if (Obj != null)
                return Obj;
            else
                return null;
        }
        else
        {
            return null;
        }
    }
    catch (Exception)
    {
        throw;
    }
}

public void Update(AppUser appuser)
{
    try
    {
        if (appuser != null)
        {
           var obj = _appDbContext.Update(appuser);
            if (obj != null)
                _appDbContext.SaveChanges();
                }
            }
            catch (Exception)
            {
                throw;
            }
        }
    }
}
```

Business Access Layer

The business layer communicates with the data access layer and the presentation layer logic layer process the actions that make the logical decision and evaluations the main function of this layer is to process the data between surrounding layers.

Our Structure of the project in the Business Access Layer will be like this.



```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
    <ItemGroup>
        <PackageReference Include="jose-jwt" Version="3.2.0" />
        <PackageReference Include="Microsoft.AspNetCore.Authentication.JwtBearer" Version="6.0.1" />
        <PackageReference Include="Microsoft.EntityFrameworkCore" Version="6.0.1" />
        <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="6.0.1">
            <PrivateAssets>all</PrivateAssets>
            <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
        </PackageReference>
        <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="6.0.1" />
        <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="6.0.1">
            <PrivateAssets>all</PrivateAssets>
            <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
        </PackageReference>
        <PackageReference Include="Swashbuckle.AspNetCore" Version="6.2.3" />
    </ItemGroup>
  <ItemGroup>
    <ProjectReference Include="..\JWTTokenAuthInAspNet6_DAL\JWTTokenAuthInAspNet6_DAL.csproj" />
  </ItemGroup>

</Project>
```

```
JWTTokenAuthInAspNet6
    Connected Services
    Dependencies
    Properties
    Commons
    Controllers
        AuthController.cs
    ViewModels
    appsettings.json
    Program.cs
JWTTokenAuthInAspNet6_BAL
    Dependencies
    Extensions
        AddJWTTokenServices.cs
    Services
        ServiceAppUser.cs
JWTTokenAuthInAspNet6_DAL
    Dependencies
    Contracts
        IRepository.cs
    Data
    Migrations
    Models
        AppRoles.cs
        AppUser.cs
        Responce.cs
    Repositories
        RepositoryAppUser.cs
```

In this layer, we will have two folders

- Extensions Method Folder
- And Services Folder

In the business access layer, we have our services that communicate with the surrounding layers like the data layer and Presentation Layer.

Services

Services define the application's business logic. We develop services that interact with the data layer and move data from the data layer to the presentation layer and from the presentation layer to the data access layer.

Example

```
using JWTTokenAuthInAspNet6_DAL.Contracts;
using JWTTokenAuthInAspNet6_DAL.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace JWTTokenAuthInAspNet6_BAL.Services
{
    public class ServiceAppUser
    {
        public readonly IRepository<AppUser> _repository;
        public ServiceAppUser(IRepository<AppUser> repository)
        {
            _repository = repository;
        }
        //Create Method
        public async Task<AppUser> AddUser(AppUser appUser)
        {
            try
            {
                if (appUser == null)
                {
                    throw new ArgumentNullException(nameof(appUser));
                }
                else
                {
                    return await _repository.Create(appUser);
                }
            }
            catch (Exception)
            {
```

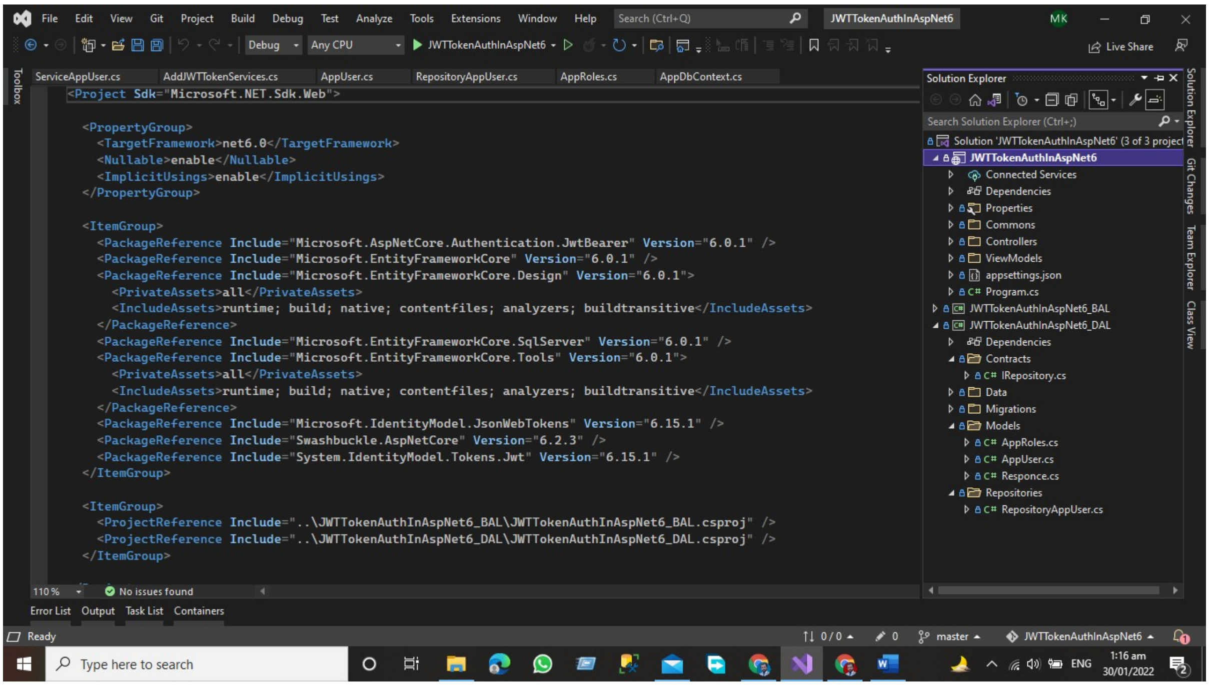
```
        throw;
    }
}

public void DeleteUser(int Id)

{
    try
    {
        if (Id != 0)
        {
            var obj = _repository.GetAll().Where(x => x.Id == Id).FirstOrDefault();
            _repository.Delete(obj);
        }
    }
    catch (Exception)
    {
        throw;
    }
}

public void UpdateUser(int Id)

{
    try
    {
        if (Id != 0)
        {
            var obj = _repository.GetAll().Where(x => x.Id == Id).FirstOrDefault();
            if (obj != null)
                _repository.Update(obj);
        }
    }
    catch (Exception)
    {

        throw;
    }

}

public IEnumerable<AppUser> GetAllUser()
```

```
        {
          try
          {
            return _repository.GetAll().ToList();
          }
          catch (Exception)
          {

              throw;
          }
        }


    }
}
```

Presentation layer

The top-most layer of the 3-Tier architecture is the Presentation layer the main function of this layer is to give the results to the user or in simple words to present the data that we get from the business access layer and give the results to the front-end user.

In the presentation layer, we have the default template of the asp.net core application here OUR structure of the application will be like this.



```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Authentication.JwtBearer" Version="6.0.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="6.0.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="6.0.1">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
    </PackageReference>
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="6.0.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="6.0.1">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
    </PackageReference>
    <PackageReference Include="Microsoft.IdentityModel.JsonWebTokens" Version="6.15.1" />
    <PackageReference Include="Swashbuckle.AspNetCore" Version="6.2.3" />
    <PackageReference Include="System.IdentityModel.Tokens.Jwt" Version="6.15.1" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include="..\JWTTokenAuthInAspNet6_BAL\JWTTokenAuthInAspNet6_BAL.csproj" />
    <ProjectReference Include="..\JWTTokenAuthInAspNet6_DAL\JWTTokenAuthInAspNet6_DAL.csproj" />
  </ItemGroup>
```

The presentation layer is responsible to give the results to the user against every HTTP request. Here we have a controller that is responsible for handling the HTTP request and communicates with the business layer against each HTTP request get the get data from the associated layer and then present it to the User.

Advantages of 3-Tier Architecture

- It makes the logical separation between the presentation layer, business layer, and data layer.
- Migration to a new environment is fast.
- In This Model, each tier is independent so we can set the different developers on each tier for the fast development of applications
- Easy to maintain and understand the large-scale applications
- The business layer is in between the presentation layer and data layer the application is more secured because the client will not have direct access to the database.
- The process data that is sent from the business layer is validated at this level.
- The Posted data from the presentation layer will be validated at the business layer before Inserting or Updating the Database
- Database security can be provided at BAL Layer
- Define the business logic one in the BAL Layer and then share it among the other components at the presentation layer
- Easy to Appy for Object-oriented Concepts
- Easy to update the data provided quires at DAL Layer

Dis-Advantages of 3-Tier Architecture

- It takes a lot of time to build the small part of the application
- Need Good understanding of Object-Oriented programming

Conclusion

In this chapter, we have learned about 3-tier Architecture and how the three-layer exchange data between them how we can add the Data Access Layer and Business access layer in the project and studied how these layers communicate with each other.

Presentation Layer (PL)

The top-most layer of the 3-Tier architecture is the Presentation layer the main function of this layer is to give the results to the user or in simple words to present the data that we get from the business access layer and give the results to the front-end user.

Business Access Layer (BAL)

The logic layer communicates with the data access layer and the presentation layer logic layer process the actions that make the logical decision and evaluations the main function of this layer is to process the data between surrounding layers.

Data Access Layer (DAL)

The main function of this application is to access and store the data from the database and the process of the data to business access layer data goes to the presentation layer against user request.

Complete Git Hub Project URL

Click here for complete project access

Chapter 3- Onion Architecture

- ✓ Introduction
- ✓ What is Onion architecture
- ✓ Layers in Onion Architecture
- ✓ Domain Layer
- ✓ Repository Layer
- ✓ Service Layer
- ✓ Presentation Layer
- ✓ Advantages of Onion Architecture
- ✓ Implementation of Onion Architecture
- ✓ Project Structure.
- ✓ Domain Layer
- ✓ Models Folder
- ✓ Data Folder
- ✓ Repository Layer
- ✓ Service Layer
- ✓ I Custom Service
- ✓ Custom Service
- ✓ Department Service
- ✓ Student Service
- ✓ Result Service
- ✓ Subject GPA Service
- ✓ Presentation Layer
- ✓ Dependency Injection
- ✓ Modify Program.cs File
- ✓ Controllers
- ✓ Output
- ✓ Conclusion

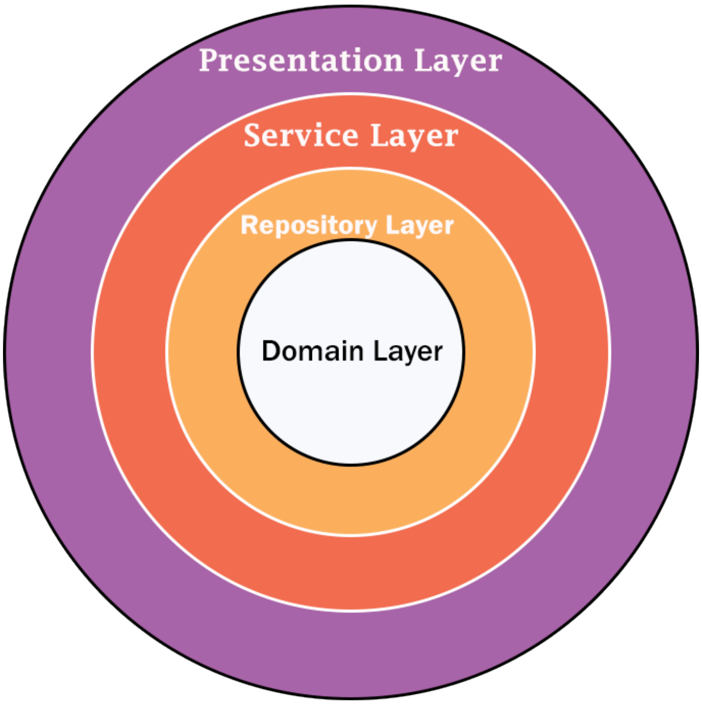
Introduction

In this chapter, we will cover the onion architecture using the Asp.Net 6 Web API. Onion architecture term introduces by Jeffrey Palermo in 2008 this architecture provides us a better way to build applications using this architecture our applications are better testable maintainable and dependable on infrastructures like databases and services. Onion architecture solves common problems like coupling and separation of concerns.

What is Onion architecture

In onion architecture, we have the domain layer, repository layer, service layer, and presentation layer. Onion architecture solves the problem that we face during the enterprise applications like coupling and separations of concerns. Onion architecture also solves the problem that we confronted in three-tier architecture and N-Layer architecture. In Onion architecture, our layer communicates with each other using interfaces.

Onion Architecture in Asp.Net Core Web API



All Rights Reserved By Sardar Mudassar Ali Khan

Layers in Onion Architecture

Onion architecture uses the concept of the layer, but it is different from N-layer architecture and 3-Tier architecture.

Domain Layer

This layer lies in the center of the architecture where we have application entities which are the application model classes or database model classes using the code first approach in the application development using Asp.net core these entities are used to create the tables in the database.

Repository Layer

The repository layer act as a middle layer between the service layer and model objects we will maintain all the database migrations and database context Object in this layer. We will add the interfaces that consist the of data access pattern for reading and writing operations with the database.

Service Layer

This layer is used to communicate with the presentation and repository layer. The service layer holds all the business logic of the entity. In this layer services interfaces are kept separate from their implementation for loose coupling and separation of concerns.

Presentation Layer

In the case of the API Presentation layer that presents us the object data from the database using the HTTP request in the form of JSON Object. But in the case of front-end applications, we present the data using the UI by consuming the APIS.

Advantages of Onion Architecture

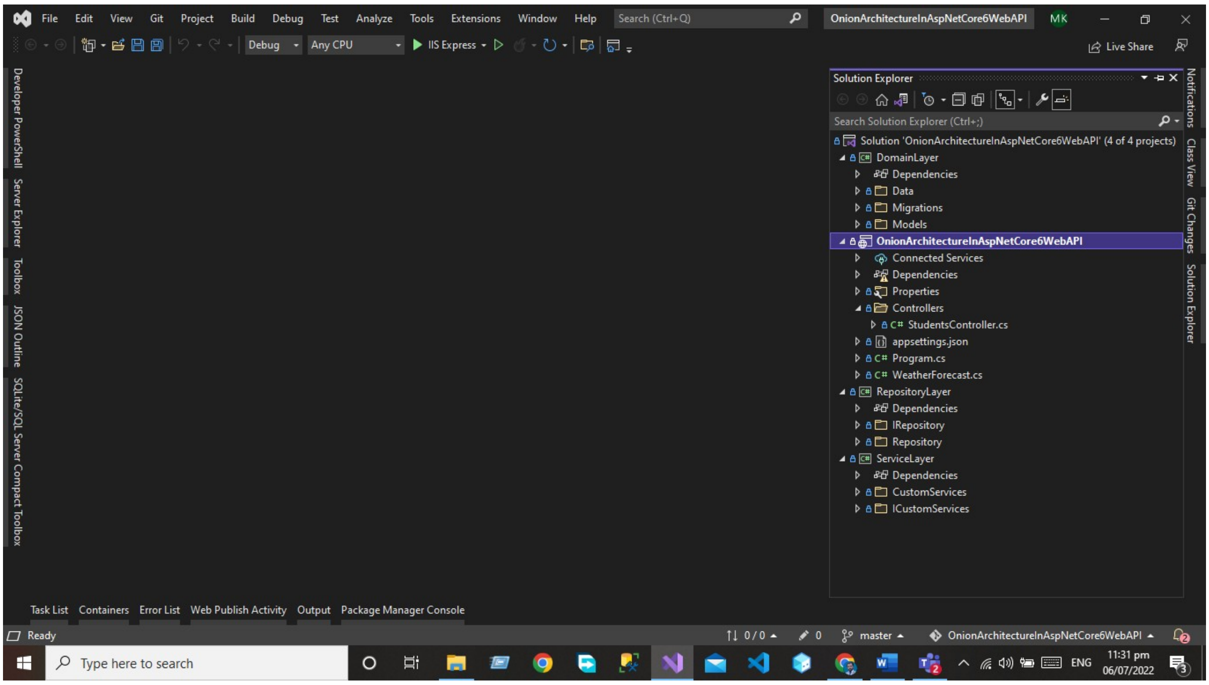
- Onion architecture provides us with the batter maintainability of code because code depends on layers.
- It provides us batter testability for unit tests we can write the separate test cases in layers without affecting the other module in the application.
- Using the onion architecture our application is loosely coupled because our layers communicate with each other using the interface.
- Domain entities are the core and center of the architecture and have access to databases and UI Layer.
- A Complete implementation would be provided to the application at run time.
- The external layer never depends on the external layer.

Implementation of Onion Architecture

Now we are going to develop the project using the Onion Architecture

First, you need to create the Asp.net Core web API project using visual studio. After creating the project, we will add our layer to the project after adding all the layers our project structure will look like this.

Project Structure.



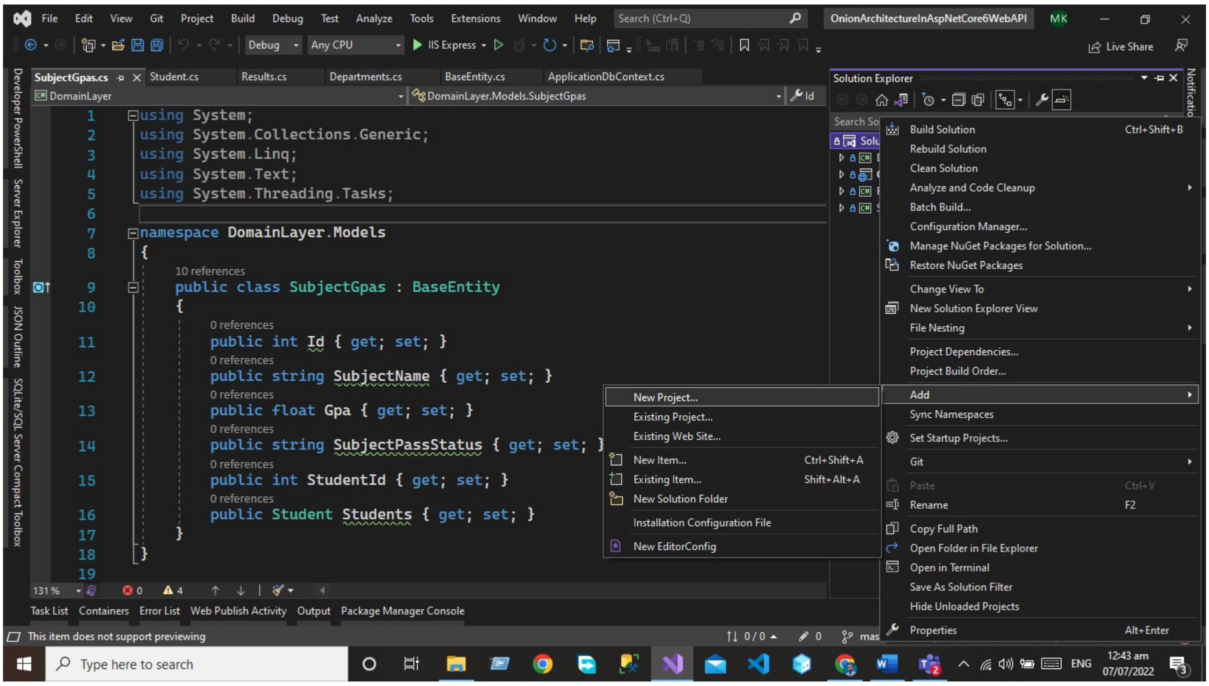
```
Solution 'OnionArchitectureInAspNetCore6WebAPI' (4 of 4 projects)
  DomainLayer
    Dependencies
    Data
    Migrations
    Models
  OnionArchitectureInAspNetCore6WebAPI
    Connected Services
    Dependencies
    Properties
    Controllers
      StudentsController.cs
    appsettings.json
    Program.cs
    WeatherForecast.cs
  RepositoryLayer
    Dependencies
    IRepository
    Repository
  ServiceLayer
    Dependencies
    CustomServices
    ICustomServices
```

Domain Layer

This layer lies in the center of the architecture where we have application entities which are the application model classes or database model classes using the code first approach in the application development using Asp.net core these entities are used to create the tables in the database.

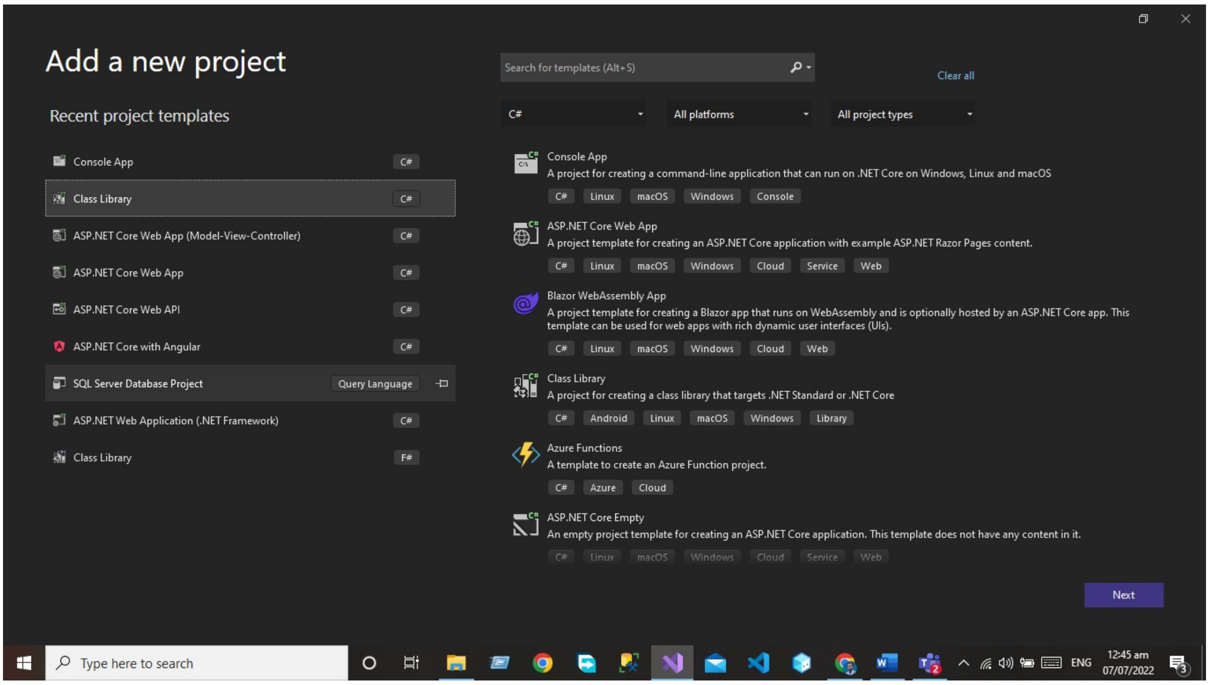
For the Domain layer, we need to add the library project to our application.

Write click on the Solution and then click on add option.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DomainLayer.Models
{
    10 references
    public class SubjectGpas : BaseEntity
    {
        0 references
        public int Id { get; set; }
        0 references
        public string SubjectName { get; set; }
        0 references
        public float Gpa { get; set; }
        0 references
        public string SubjectPassStatus { get; set; }
        0 references
        public int StudentId { get; set; }
        0 references
        public Student Students { get; set; }
    }
}
```

Add the library project to your solution



Add a new project

Recent project templates

- Console App C#
- Class Library C#
- ASP.NET Core Web App (Model-View-Controller) C#
- ASP.NET Core Web App C#
- ASP.NET Core Web API C#
- ASP.NET Core with Angular C#
- SQL Server Database Project Query Language
- ASP.NET Web Application (.NET Framework) C#
- Class Library F#

Search for templates (Alt+S) | Clear all

C# | All platforms | All project types

Console App
A project for creating a command-line application that can run on .NET Core on Windows, Linux and macOS
C# Linux macOS Windows Console

ASP.NET Core Web App
A project template for creating an ASP.NET Core application with example ASP.NET Razor Pages content.
C# Linux macOS Windows Cloud Service Web

Blazor WebAssembly App
A project template for creating a Blazor app that runs on WebAssembly and is optionally hosted by an ASP.NET Core app. This template can be used for web apps with rich dynamic user interfaces (UIs).
C# Linux macOS Windows Cloud Web

Class Library
A project for creating a class library that targets .NET Standard or .NET Core
C# Android Linux macOS Windows Library

Azure Functions
A template to create an Azure Function project.
C# Azure Cloud

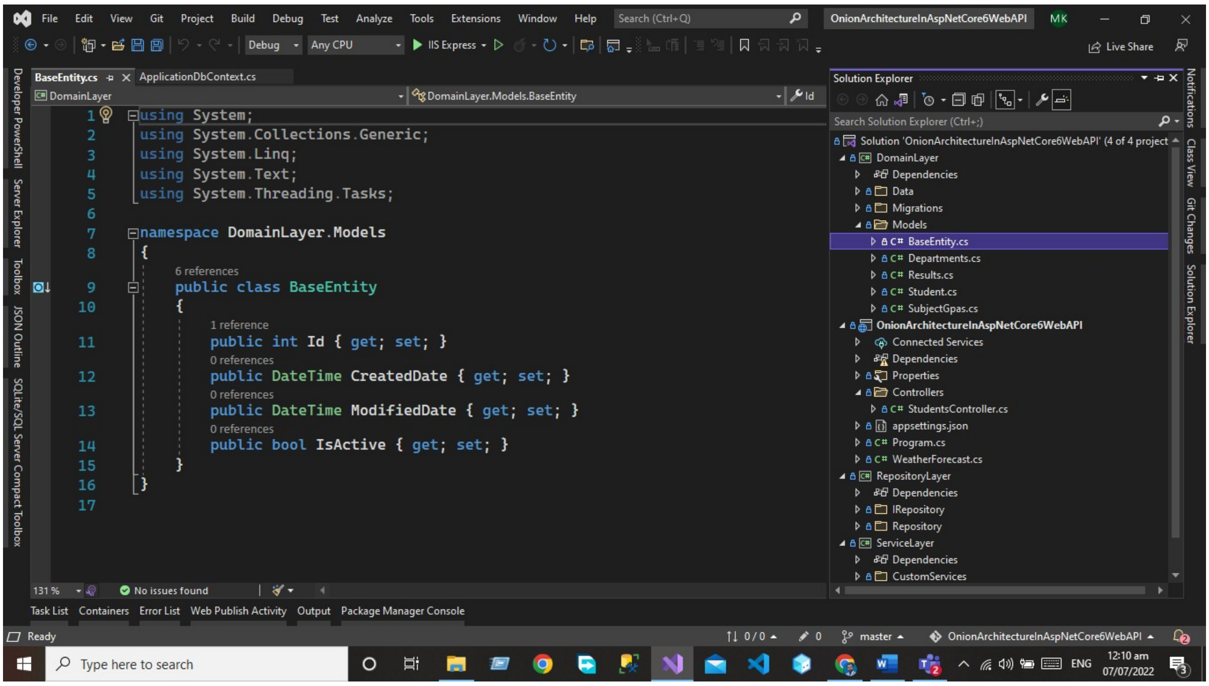
ASP.NET Core Empty
An empty project template for creating an ASP.NET Core application. This template does not have any content in it.
C# Linux macOS Windows Cloud Service Web

Next

Name this project as Domain Layer

Models Folder

First, you need to add the Models folder that will be used to create the database entities. In the Models folder, we will create the following database entities.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DomainLayer.Models
{
    public class BaseEntity
    {
        public int Id { get; set; }
        public DateTime CreatedDate { get; set; }
        public DateTime ModifiedDate { get; set; }
        public bool IsActive { get; set; }
    }
}
```

Base Entity

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DomainLayer.Models
{
    public class BaseEntity
    {
        public int Id { get; set; }
        public DateTime CreatedDate { get; set; }
        public DateTime ModifiedDate { get; set; }
        public bool IsActive { get; set; }
    }
}
```

Department

Department entity will extend the base entity

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DomainLayer.Models
{
    public class Departments :BaseEntity
    {
        public int Id { get; set; }
        public string DepartmentName { get; set; }
        public int StudentId { get; set; }
        public Student Students { get; set; }
    }
}
```

Result

Result Entity will extend the base entity

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DomainLayer.Models
{
    public class Resultss : BaseEntity
    {
        [Key]
        public int Id { get; set; }
        public string? ResultStatus { get; set; }
        public int StudentId { get; set; }
        public Student Students { get; set; }
    }
}
```

Students

Student Entity will extend the base entity

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DomainLayer.Models
{
    public class Student : BaseEntity
    {
        [Key]
        public int Id { get; set; }
        public string? Name { get; set; }
        public string? Address { get; set; }
        public string? Emial { get; set; }
        public string? City { get; set; }
        public string? State { get; set; }
        public string? Country { get; set; }
        public int? Age { get; set; }
        public DateTime? BirthDate { get; set;}

    }
}
```

SubjectGpa

Subject GPA Entity will extend the Base Entity

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DomainLayer.Models
{
    public class SubjectGpas : BaseEntity
    {
        public int Id { get; set; }
        public string SubjectName { get; set; }
        public float Gpa { get; set; }
        public string SubjectPassStatus { get; set; }
        public int StudentId { get; set; }
        public Student Students { get; set; }
    }
}
```

Data Folder

Add the Data in the domain that is used to add the database context class. The database context class is used to maintain the session with the underlying database using which you can perform the CRUD operation.

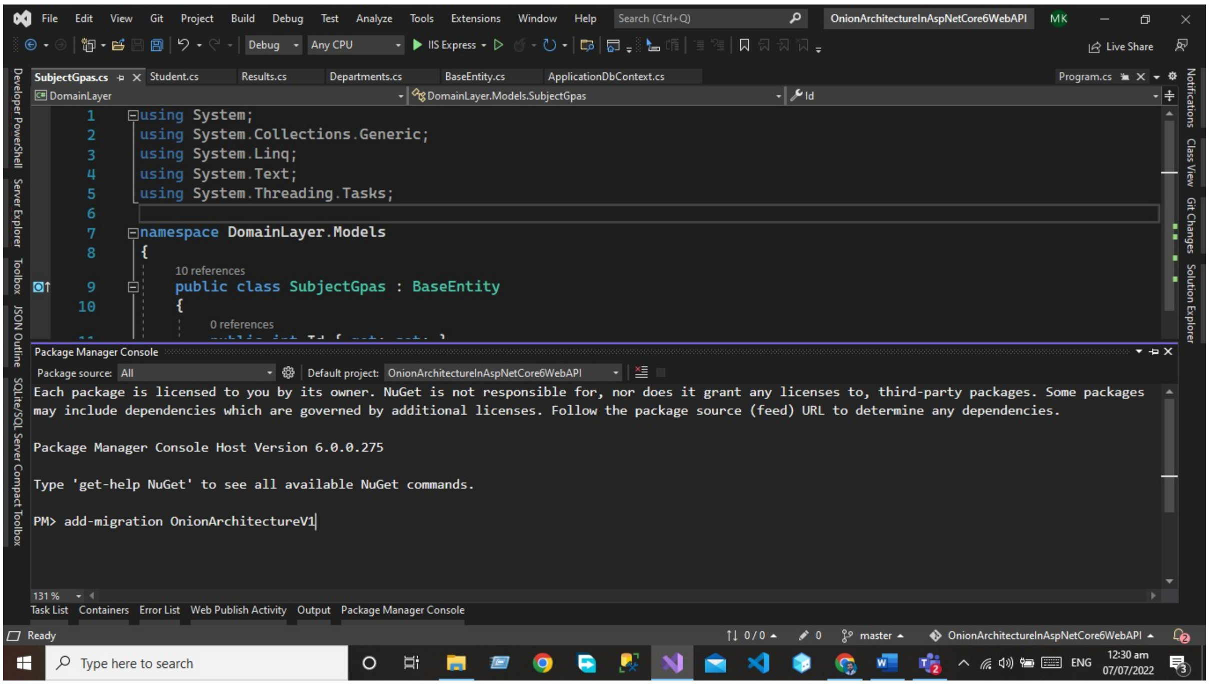
Write click on the application and create the class ApplicationDbContext.cs

```
using DomainLayer.Models;
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DomainLayer.Data
{
    public class ApplicationDbContext : DbContext
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options)
        {


        }
        protected override void OnModelCreating(ModelBuilder builder)
        {
            base.OnModelCreating(builder);
        }
        public DbSet<Student> Students { get; set; }
        public DbSet<Departments> Departments { get; set; }
        public DbSet<SubjectGpas> SubjectGpas { get; set; }
        public DbSet<Resultss> Results { get; set; }
    }
```

}

After Adding the DbSet properties we need to add the migration using the package manager console and run the command Add-Migration.

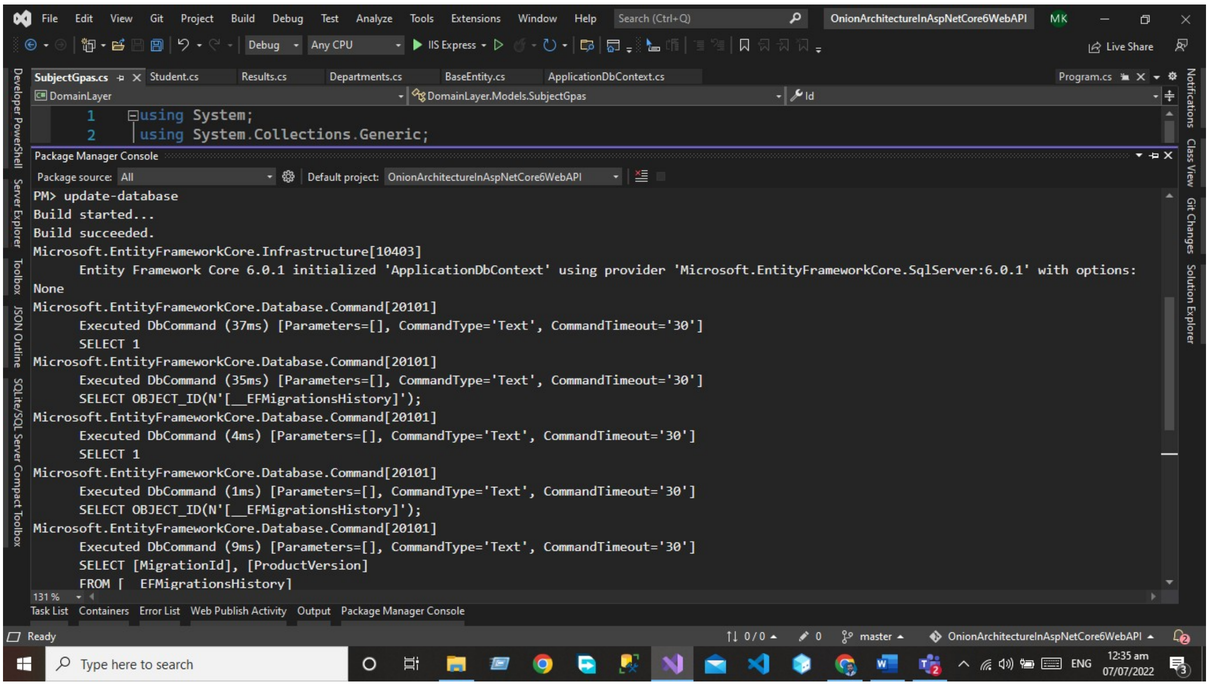
add-migration OnionArchitectureV1



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DomainLayer.Models
{
    10 references
    public class SubjectGpas : BaseEntity
    {
        0 references
```

```
Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages. Some packages
may include dependencies which are governed by additional licenses. Follow the package source (feed) URL to determine any dependencies.

Package Manager Console Host Version 6.0.0.275

Type 'get-help NuGet' to see all available NuGet commands.

PM> add-migration OnionArchitectureV1
```

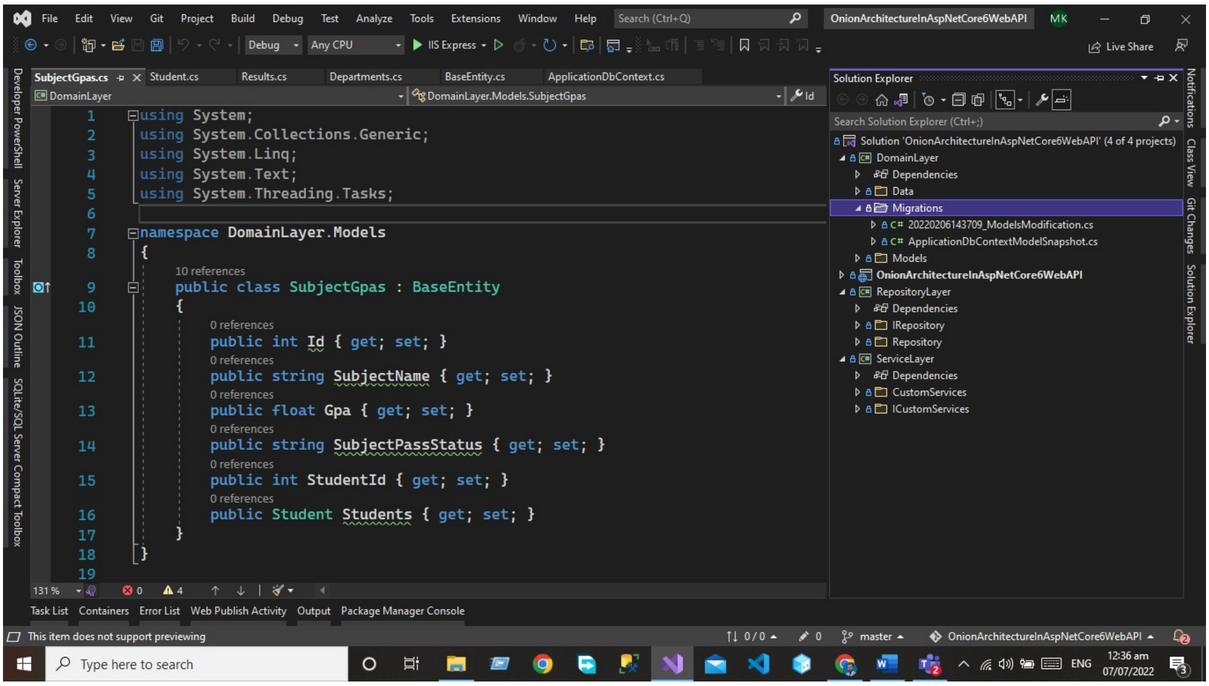
After executing the commands now, you need to update the database by executing the **update-database** Command



```
PM> update-database
Build started...
Build succeeded.
Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core 6.0.1 initialized 'ApplicationDbContext' using provider 'Microsoft.EntityFrameworkCore.SqlServer:6.0.1' with options:
None
Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (37ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT 1
Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (35ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT OBJECT_ID(N'[__EFMigrationsHistory]');
Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (4ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT 1
Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (1ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT OBJECT_ID(N'[__EFMigrationsHistory]');
Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (9ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT [MigrationId], [ProductVersion]
      FROM [__EFMigrationsHistory]
```

Migration Folder

After executing both commands now you can see the migration folder in the domain layer.



File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help
Search (Ctrl+Q)
OnionArchitectureInAspNetCore6WebAPI
MK
Debug
Any CPU
IIS Express
Live Share
SubjectGpas.cs
Student.cs
Results.cs
Departments.cs
BaseEntity.cs
ApplicationDbContext.cs
DomainLayer
DomainLayer.Models.SubjectGpas
Id
Solution Explorer
Search Solution Explorer (Ctrl+;)
131 %
0
4
Task List Containers Error List Web Publish Activity Output Package Manager Console
This item does not support previewing
0 / 0
0
master
OnionArchitectureInAspNetCore6WebAPI
Type here to search
ENG
12:36 am
07/07/2022

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DomainLayer.Models
{
    10 references
    public class SubjectGpas : BaseEntity
    {
        0 references
        public int Id { get; set; }
        0 references
        public string SubjectName { get; set; }
        0 references
        public float Gpa { get; set; }
        0 references
        public string SubjectPassStatus { get; set; }
        0 references
        public int StudentId { get; set; }
        0 references
        public Student Students { get; set; }
    }
}
```

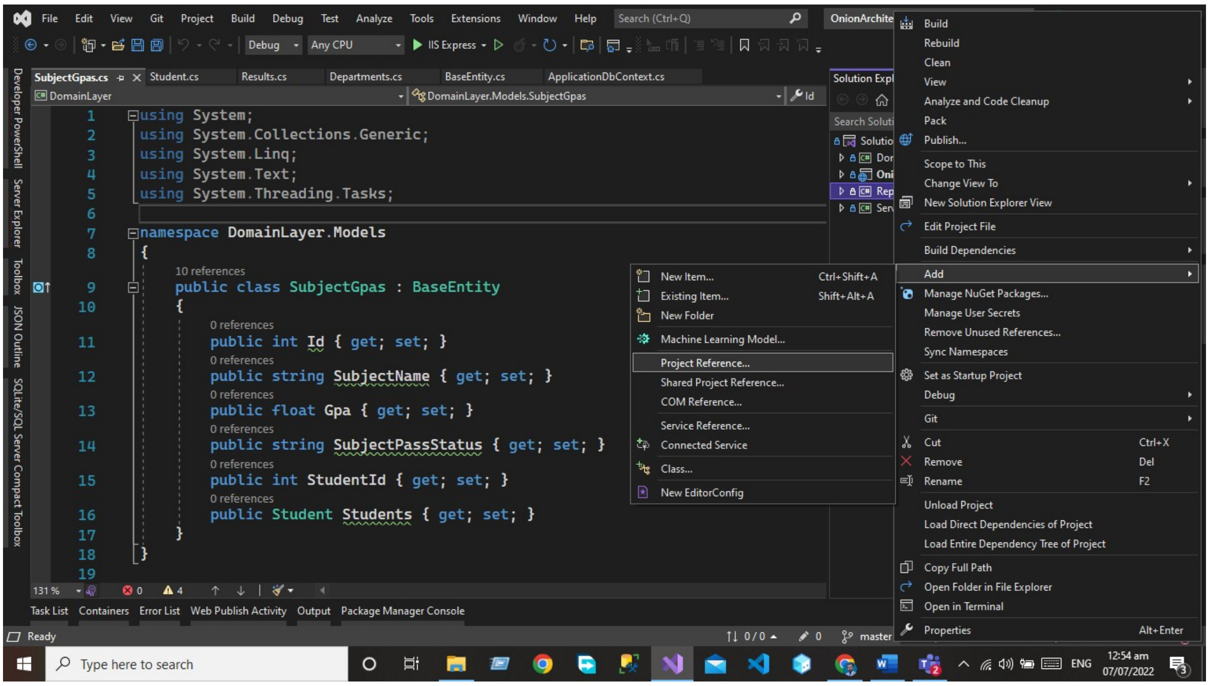
```
Solution 'OnionArchitectureInAspNetCore6WebAPI' (4 of 4 projects)
  DomainLayer
    Dependencies
    Data
    Migrations
      20220206143709_ModelsModification.cs
      ApplicationDbContextModelSnapshot.cs
    Models
  OnionArchitectureInAspNetCore6WebAPI
  RepositoryLayer
    Dependencies
    IRepository
    Repository
  ServiceLayer
    Dependencies
    CustomServices
    ICustomServices
```

Repository Layer

The repository layer act as a middle layer between the service layer and model objects we will maintain all the database migrations and database context Object in this layer. We will add the interfaces that consist the of data access pattern for reading and writing operations with the database.

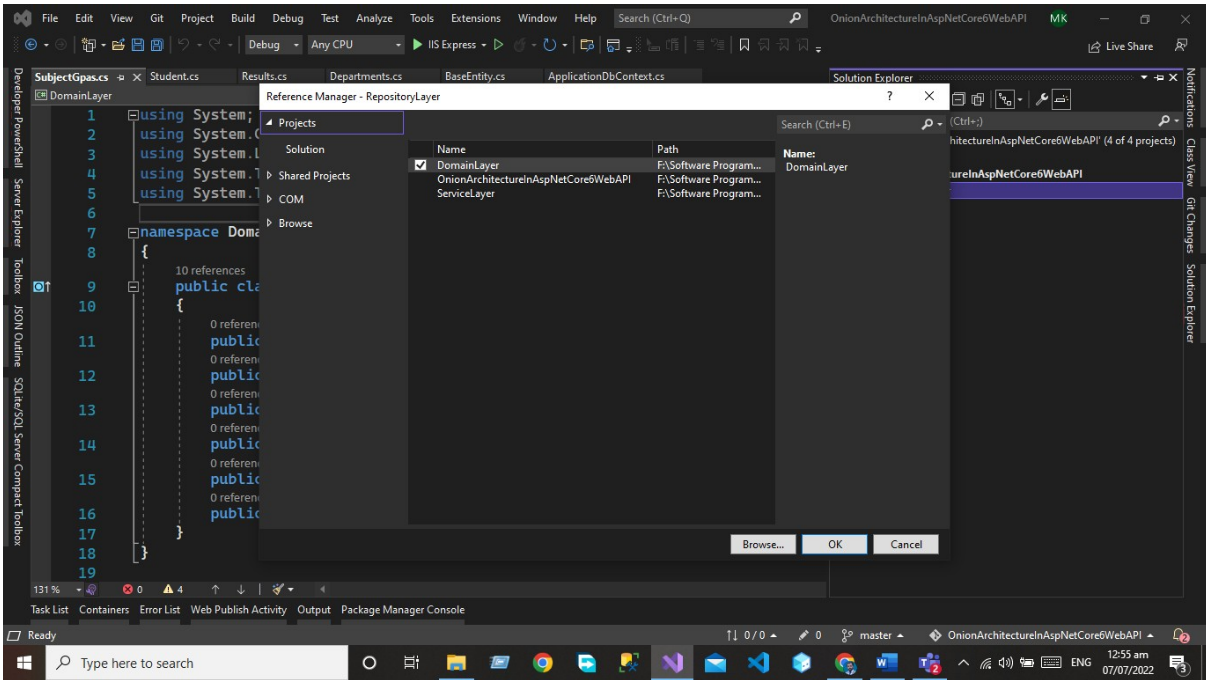
Now we will work on Repository Layer we will follow the same project as we did for the Domain layer add the library project in your applicant and give a name to that project Repository layer.

But here we need to add the project reference of the Domain layer in the repository layer. Write click on the project and then click the Add button after that we will add the project references in the Repository layer.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DomainLayer.Models
{
    10 references
    public class SubjectGpas : BaseEntity
    {
        0 references
        public int Id { get; set; }
        0 references
        public string SubjectName { get; set; }
        0 references
        public float Gpa { get; set; }
        0 references
        public string SubjectPassStatus { get; set; }
        0 references
        public int StudentId { get; set; }
        0 references
        public Student Students { get; set; }
    }
}
```

Click on project reference now and select the Domain layer.



Reference Manager - RepositoryLayer

	Name	Path
☑	DomainLayer	F:\Software Program...
	OnionArchitectureInAspNetCore6WebAPI	F:\Software Program...
	ServiceLayer	F:\Software Program...

Name:
DomainLayer

Now we need to add the two folders.

- IRepository
- Repository

IRepository

IRepsitory folder contains the generic interface using this interface we will create the CRUD operation for our all entities.

Generic Interface will extend the Base-Entity for using some properties. The Code of the generic interface is given below.

```
using DomainLayer.Models;

namespace RepositoryLayer.IRepository
{
    public interface IRepository<T> where T: BaseEntity
    {
        IEnumerable<T> GetAll();
        T Get(int Id);
        void Insert(T entity);
        void Update(T entity);
        void Delete(T entity);
        void Remove(T entity);
        void SaveChanges();
    }
}
```

Repository

Now we create the Generic repository that extends the Generic IRepository Interface. The Code of the generic repository is given below.

```
using DomainLayer.Data;
using DomainLayer.Models;
using Microsoft.EntityFrameworkCore;
using RepositoryLayer.IRepository;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace RepositoryLayer.Repository
{
    public class Repository<T> : IRepository<T> where T : BaseEntity
    {
        #region property
```

```
private readonly ApplicationDbContext _applicationDbContext;
private DbSet<T> entities;
#endregion

#region Constructor
public Repository(ApplicationDbContext applicationDbContext)
{
    _applicationDbContext = applicationDbContext;
    entities = _applicationDbContext.Set<T>();
}
#endregion

public void Delete(T entity)
{
    if (entity == null)
    {
        throw new ArgumentNullException("entity");
    }
    entities.Remove(entity);
    _applicationDbContext.SaveChanges();
}

public T Get(int Id)
{
    return entities.SingleOrDefault(c => c.Id == Id);
}

public IEnumerable<T> GetAll()
{
    return entities.AsEnumerable();
}

public void Insert(T entity)
{
    if (entity == null)
    {
        throw new ArgumentNullException("entity");
    }
     entities.Add(entity);
    _applicationDbContext.SaveChanges();
}
```

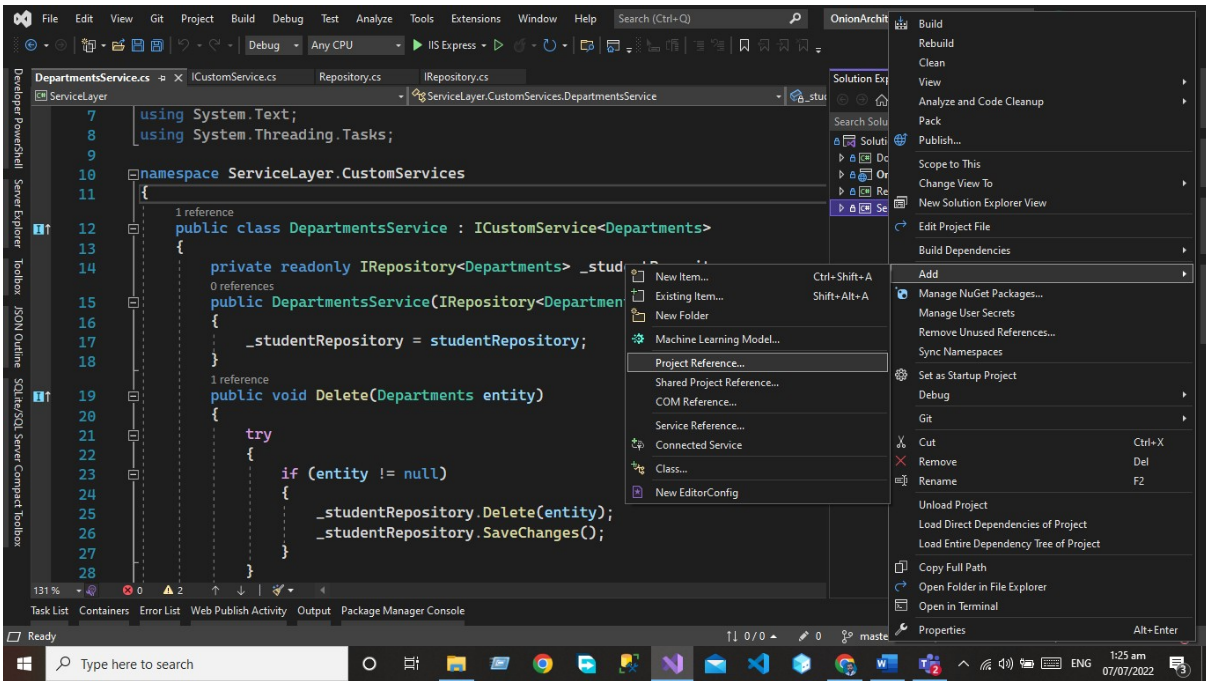
```
        public void Remove(T entity)
        {
            if (entity == null)
            {
                throw new ArgumentNullException("entity");
            }
            entities.Remove(entity);
        }

        public void SaveChanges()
        {
            _applicationDbContext.SaveChanges();
        }

        public void Update(T entity)
        {
            if (entity == null)
            {
                throw new ArgumentNullException("entity");
            }
            entities.Update(entity);
            _applicationDbContext.SaveChanges();
        }

    }
}
```

We will use this repository in our service layer.

Service Layer

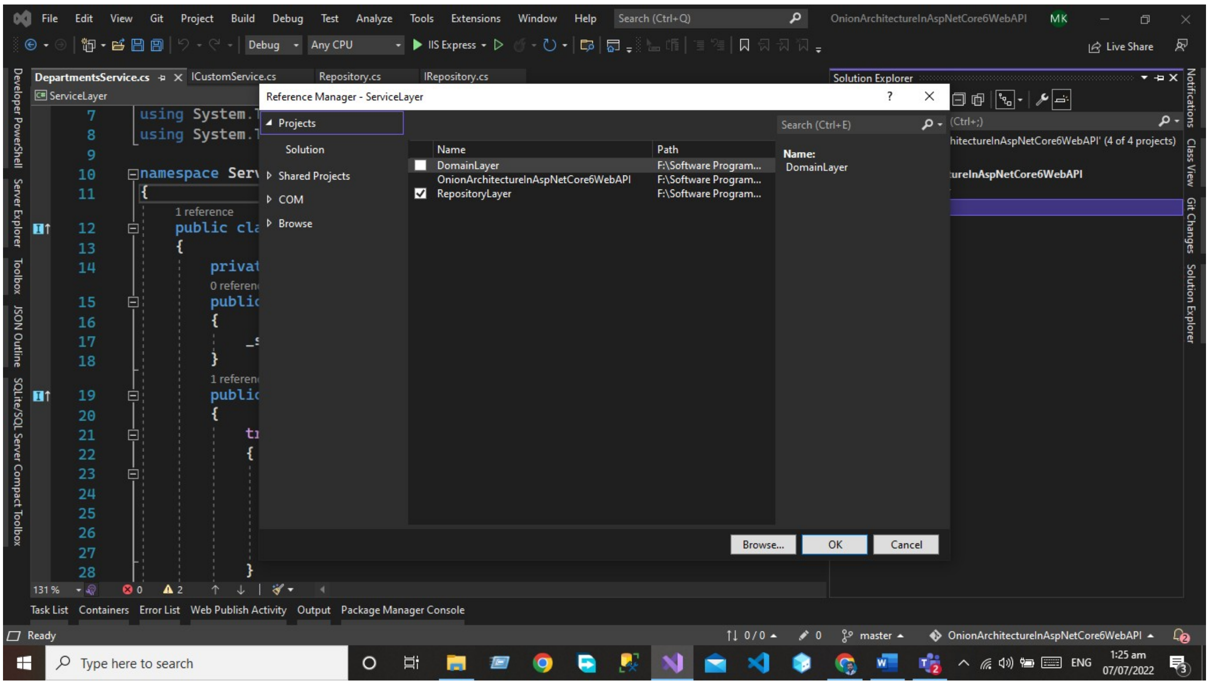
This layer is used to communicate with the presentation and repository layer. The service layer holds all the business logic of the entity. In this layer services interfaces are kept separate from their implementation for loose coupling and separation of concerns.

Now we need to add a new project to our solution that will be the service layer we will follow the same process for adding the library project in our application but here we need some extra work after adding the project we need to add the reference of the **Repository Layer.**



```
using System.Text;
using System.Threading.Tasks;

namespace ServiceLayer.CustomServices
{
    1 reference
    public class DepartmentsService : ICustomService<Departments>
    {
        private readonly IRepository<Departments> _stud
        0 references
        public DepartmentsService(IRepository<Departmen
        {
            _studentRepository = studentRepository;
        }
        1 reference
        public void Delete(Departments entity)
        {
            try
            {
                if (entity != null)
                {
                    _studentRepository.Delete(entity);
                    _studentRepository.SaveChanges();
                }
            }
```

Now our service layer contains the reference of the repository layer.



Reference Manager - ServiceLayer

	Name	Path
□	DomainLayer	F:\Software Program...
	OnionArchitectureInAspNetCore6WebAPI	F:\Software Program...
☑	RepositoryLayer	F:\Software Program...

Name:
DomainLayer

In the Service layer, we will create the two folders.

- ICustomServices
- CustomServices

ICustom Service

Now in the ICustomServices folder, we will create the ICustomServices Interface this interface holds the signature of the method we will implement these methods in the customs service code of the ICustomServices Interface given below.

```
using DomainLayer.Models;
using System;
using System.Collections.Generic;
using System. Linq;
using System. Text;
using System.Threading.Tasks;

namespace ServiceLayer.ICustomServices
{
    public interface ICustomService<T> where T : class
    {
        IEnumerable<T> GetAll();
        T Get(int Id);
        void Insert(T entity);
        void Update(T entity);
        void Delete(T entity);
        void Remove(T entity);

    }
}
```

Custom Service

In the custom service folder, we will create the custom service class that inherits the ICustomService interface code of the custom service class given below. We will write the crud for our all entities.

For every service, we will write the CRUD operation using our generic repository.

Department Service

```
using DomainLayer.Models;
using RepositoryLayer.IRepository;
using ServiceLayer.ICustomServices;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ServiceLayer.CustomServices
{
    public class DepartmentsService : ICustomService<Departments>
    {
        private readonly IRepository<Departments> _studentRepository;
        public DepartmentsService(IRepository<Departments> studentRepository)
        {
            _studentRepository = studentRepository;
        }
        public void Delete(Departments entity)
        {
            try
            {
                if (entity != null)
                {
                    _studentRepository.Delete(entity);
                    _studentRepository.SaveChanges();
                }
            }
            catch (Exception)
            {

                throw;
            }
        }

        public Departments Get(int Id)
        {
            try
            {
                var obj = _studentRepository.Get(Id);
                if (obj != null)
                {
                    return obj;
                }
                else
                {
```

```
            return null;
        }

    }
    catch (Exception)
    {

        throw;
    }
}

public IEnumerable<Departments> GetAll()
{
    try
    {
        var obj = _studentRepository.GetAll();
        if (obj != null)
        {
            return obj;
        }
        else
        {
            return null;
        }
    }
    catch (Exception)
    {

        throw;
    }
}

public void Insert(Departments entity)
{
    try
    {
        if (entity != null)
        {
            _studentRepository.Insert(entity);
            _studentRepository.SaveChanges();
        }
    }
```

```
            catch (Exception)
            {

                throw;
            }
        }

        public void Remove(Departments entity)
        {
            try
            {
                if (entity != null)
                {
                    _studentRepository.Remove(entity);
                    _studentRepository.SaveChanges();
                }
            }
            catch (Exception)
            {

                throw;
            }
        }
        public void Update(Departments entity)
        {
            try
            {
                if (entity != null)
                {
                    _studentRepository.Update(entity);
                    _studentRepository.SaveChanges();
                }
            }
            catch (Exception)
            {

                throw;
            }
        }
    }
}
```

Student Service

```
using DomainLayer.Models;
using RepositoryLayer.IRepository;
using ServiceLayer.ICustomServices;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ServiceLayer.CustomServices
{
    public class StudentService : ICustomService<Student>
    {
        private readonly IRepository<Student> _studentRepository;
        public StudentService(IRepository<Student> studentRepository)
        {
            _studentRepository = studentRepository;
        }
        public void Delete(Student entity)
        {
            try
            {
                if(entity!=null)
                {
                    _studentRepository.Delete(entity);
                    _studentRepository.SaveChanges();
                }
            }
            catch (Exception)
            {

                throw;
            }
        }

        public  Student Get(int Id)
        {
            try
            {
                var obj = _studentRepository.Get(Id);
                if(obj!=null)
                {
                    return obj;
                }
                else
                {
                    return null;
                }

            }
            catch (Exception)
            {

                throw;
            }
        }

        public IEnumerable<Student> GetAll()
        {
            try
            {
                var obj = _studentRepository.GetAll();
                if (obj != null)
                {
                    return obj;
                }
                else
                {
                    return null;
                }
            }
            catch (Exception)
            {

                throw;
            }
        }

        public void Insert(Student entity)
        {
            try
            {
```

```
    if (entity != null)
    {
      _studentRepository.Insert(entity);
      _studentRepository.SaveChanges();
    }
  }
  catch (Exception)
  {

    throw;
  }
}

public void Remove(Student entity)
{
  try
  {
    if(entity!=null)
    {
     _studentRepository.Remove(entity);
     _studentRepository.SaveChanges();
    }
  }
  catch (Exception)
  {

    throw;
  }
}
public void Update(Student entity)
{
  try
  {
    if(entity!=null)
    {
      _studentRepository.Update(entity);
      _studentRepository.SaveChanges();
    }
  }
  catch (Exception)
  {
```

```
                throw;
            }
        }
    }
}
```

Result Service

```
using DomainLayer.Models;
using RepositoryLayer.IRepository;
using ServiceLayer.ICustomServices;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ServiceLayer.CustomServices
{
    public class ResultService : ICustomService<Resultss>
    {
        private readonly IRepository<Resultss> _studentRepository;
        public ResultService(IRepository<Resultss> studentRepository)
        {
            _studentRepository = studentRepository;
        }
        public void Delete(Resultss entity)
        {
            try
            {
                if (entity != null)
                {
                    _studentRepository.Delete(entity);
                    _studentRepository.SaveChanges();
                }
            }
            catch (Exception)
            {

                throw;
            }

        }
```

```
public Resultss Get(int Id)
{
    try
    {
        var obj = _studentRepository.Get(Id);
        if (obj != null)
        {
            return obj;
        }
        else
        {
            return null;
        }

    }
    catch (Exception)
    {

        throw;
    }
}

public IEnumerable<Resultss> GetAll()
{
    try
    {
        var obj = _studentRepository.GetAll();
        if (obj != null)
        {
            return obj;
        }
        else
        {
            return null;
        }
    }
    catch (Exception)
    {

        throw;

}
```

```
}

public void Insert(Resultss entity)
{
    try
    {
        if (entity != null)
        {
            _studentRepository.Insert(entity);
            _studentRepository.SaveChanges();
        }
    }
    catch (Exception)
    {

        throw;
    }
}

public void Remove(Resultss entity)
{
    try
    {
        if (entity != null)
        {
            _studentRepository.Remove(entity);
            _studentRepository.SaveChanges();
        }
    }
    catch (Exception)
    {

        throw;
    }
}
public void Update(Resultss entity)
{
    try
    {
        if (entity != null)
        {
            _studentRepository.Update(entity);
```

```
                    _studentRepository.SaveChanges();
                }
            }
            catch (Exception)
            {

                throw;
            }
        }
    }
}
```

Subject GPA Service

```
using DomainLayer.Models;
using RepositoryLayer.IRepository;
using ServiceLayer.ICustomServices;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ServiceLayer.CustomServices
{
    public class SubjectGpasService : ICustomService<SubjectGpas>
    {
        private readonly IRepository<SubjectGpas> _studentRepository;
        public SubjectGpasService(IRepository<SubjectGpas> studentRepository)
        {
            _studentRepository = studentRepository;
        }
        public void Delete(SubjectGpas entity)
        {
            try
            {
                if (entity != null)
                {
                    _studentRepository.Delete(entity);
                    _studentRepository.SaveChanges();
                }

            }
            catch (Exception)
            {

                throw;
            }
        }

        public SubjectGpas Get(int Id)
        {
            try
            {
                var obj = _studentRepository.Get(Id);
                if (obj != null)
                {
                    return obj;
                }
                else
                {
                    return null;
                }

            }
            catch (Exception)
            {

                throw;
            }
        }

        public IEnumerable<SubjectGpas> GetAll()
        {
            try
            {
                var obj = _studentRepository.GetAll();
                if (obj != null)
                {
                    return obj;
                }
                else
                {
                    return null;

            }
            }
            catch (Exception)
            {

                throw;
            }
        }

        public void Insert(SubjectGpas entity)
        {
            try
            {
                if (entity != null)
                {
                    _studentRepository.Insert(entity);
                    _studentRepository.SaveChanges();
                }
            }
            catch (Exception)
            {

                throw;
            }
        }

        public void Remove(SubjectGpas entity)
        {
            try
            {
                if (entity != null)
                {
                    _studentRepository.Remove(entity);
                    _studentRepository.SaveChanges();
                }
            }
            catch (Exception)
            {

                throw;
            }
        }
        public void Update(SubjectGpas entity)
```

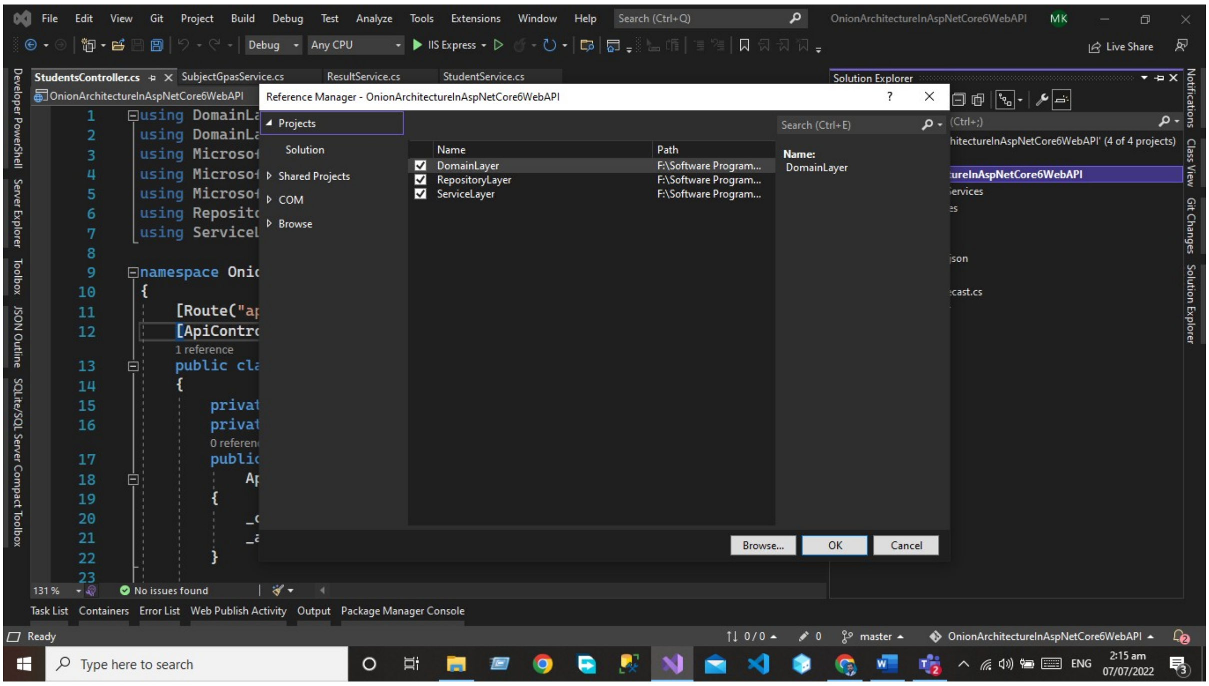
```
        {
            try
            {
                if (entity != null)
                {
                    _studentRepository.Update(entity);
                    _studentRepository.SaveChanges();
                }
            }
            catch (Exception)
            {

                throw;
            }
        }
    }
}
```

Presentation Layer

The presentation layer is our final layer that presents the data to the front-end user on every HTTP request.

In the case of the API presentation layer that presents us the object data from the database using the HTTP request in the form of JSON Object. But in the case of front-end applications, we present the data using the UI by consuming the APIS.

The presentation layer is the default Asp.net core web API project Now we need to add the project references of all the layers as we did before



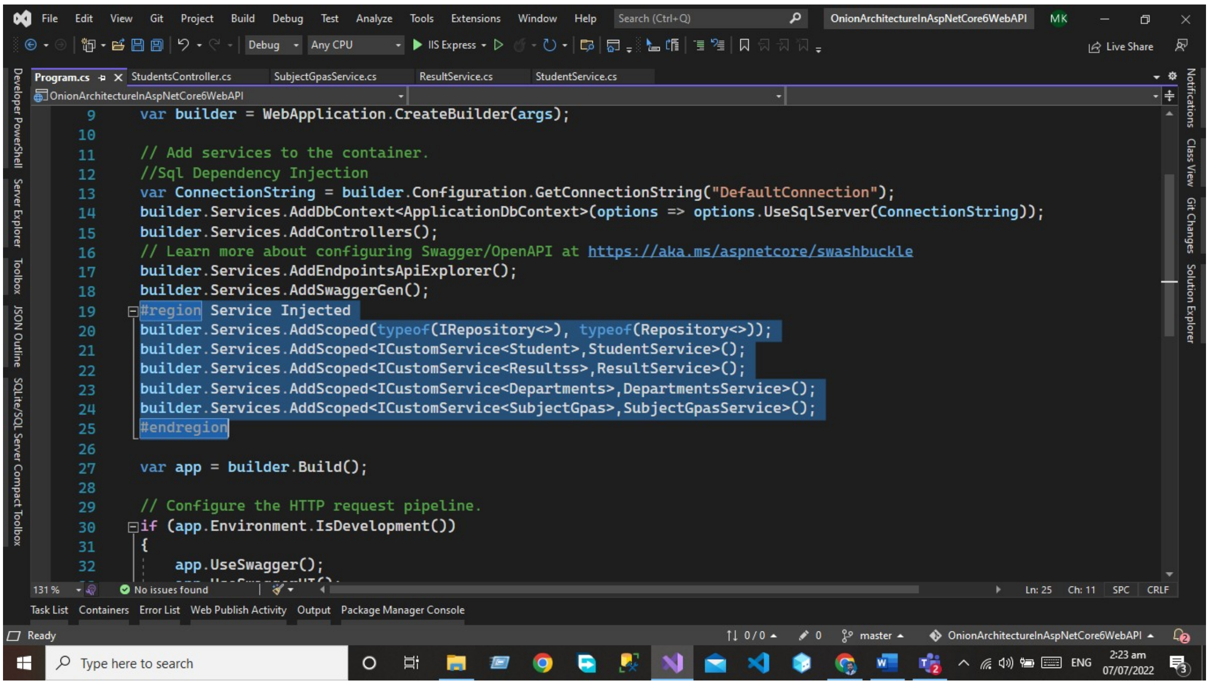
Reference Manager - OnionArchitectureInAspNetCore6WebAPI

	Name	Path
☑	DomainLayer	F:\Software Program...
☑	RepositoryLayer	F:\Software Program...
☑	ServiceLayer	F:\Software Program...

Name:
DomainLayer

Dependency Injection

Now we need to add the dependency Injection of our all services in the program.cs class



```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
//Sql Dependency Injection
var ConnectionString = builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options => options.UseSqlServer(ConnectionString));
builder.Services.AddControllers();
// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
#region Service Injected
builder.Services.AddScoped(typeof(IRepository<>), typeof(Repository<>));
builder.Services.AddScoped<ICustomService<Student>,StudentService>();
builder.Services.AddScoped<ICustomService<Resultss>,ResultService>();
builder.Services.AddScoped<ICustomService<Departments>,DepartmentsService>();
builder.Services.AddScoped<ICustomService<SubjectGpas>,SubjectGpasService>();
#endregion

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
```

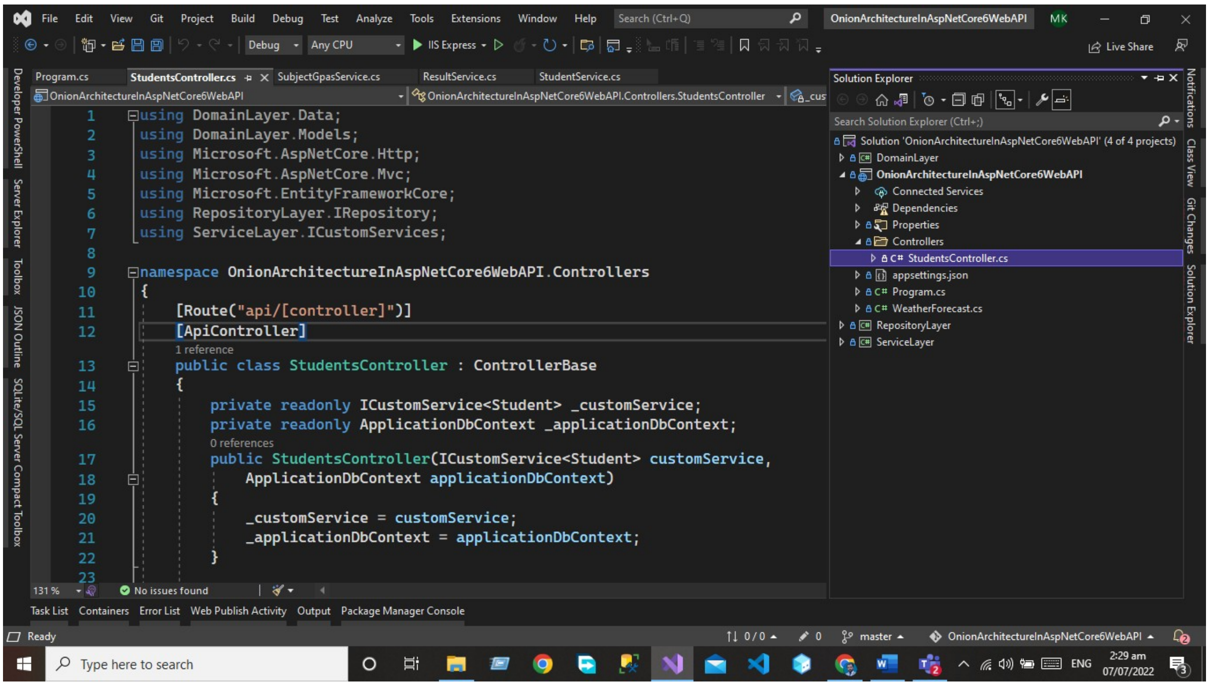
Modify Program.cs File

The Code of the Startup Class is Given below

```
using DomainLayer.Data;
using DomainLayer.Models;
using Microsoft.EntityFrameworkCore;
using RepositoryLayer.IRepository;
using RepositoryLayer.Repository;
using ServiceLayer.CustomServices;
using ServiceLayer.ICustomServices;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
//Sql Dependency Injection
var ConnectionString = builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options => options.UseSqlServer(ConnectionString));
builder.Services.AddControllers();
// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
#region Service Injected
builder.Services.AddScoped(typeof(IRepository<>), typeof(Repository<>));
builder.Services.AddScoped<ICustomService<Student>,StudentService>();
builder.Services.AddScoped<ICustomService<Resultss>,ResultService>();
builder.Services.AddScoped<ICustomService<Departments>,DepartmentsService>();
builder.Services.AddScoped<ICustomService<SubjectGpas>,SubjectGpasService>();
#endregion

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Controllers

Controllers are used to handle the HTTP request. Now we need to add the student controller that will interact will our service layer and display the data to the users.



```
using DomainLayer.Data;
using DomainLayer.Models;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using RepositoryLayer.IRepository;
using ServiceLayer.ICustomServices;

namespace OnionArchitectureInAspNetCore6WebAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    1 reference
    public class StudentsController : ControllerBase
    {
        private readonly ICustomService<Student> _customService;
        private readonly ApplicationDbContext _applicationDbContext;
        0 references
        public StudentsController(ICustomService<Student> customService,
            ApplicationDbContext applicationDbContext)
        {
            _customService = customService;
            _applicationDbContext = applicationDbContext;
        }
```

File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help
Search (Ctrl+Q)
OnionArchitectureInAspNetCore6WebAPI
Debug
Any CPU
IIS Express
Live Share
Program.cs
StudentsController.cs
SubjectGpasService.cs
ResultService.cs
StudentService.cs
OnionArchitectureInAspNetCore6WebAPI.Controllers.StudentsController
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'OnionArchitectureInAspNetCore6WebAPI' (4 of 4 projects)
DomainLayer
OnionArchitectureInAspNetCore6WebAPI
Connected Services
Dependencies
Properties
Controllers
StudentsController.cs
appsettings.json
Program.cs
WeatherForecast.cs
RepositoryLayer
ServiceLayer
131 %
No issues found
Task List Containers Error List Web Publish Activity Output Package Manager Console
Ready
master
Type here to search
ENG
2:29 am
07/07/2022

The Code of the Controller is given below.

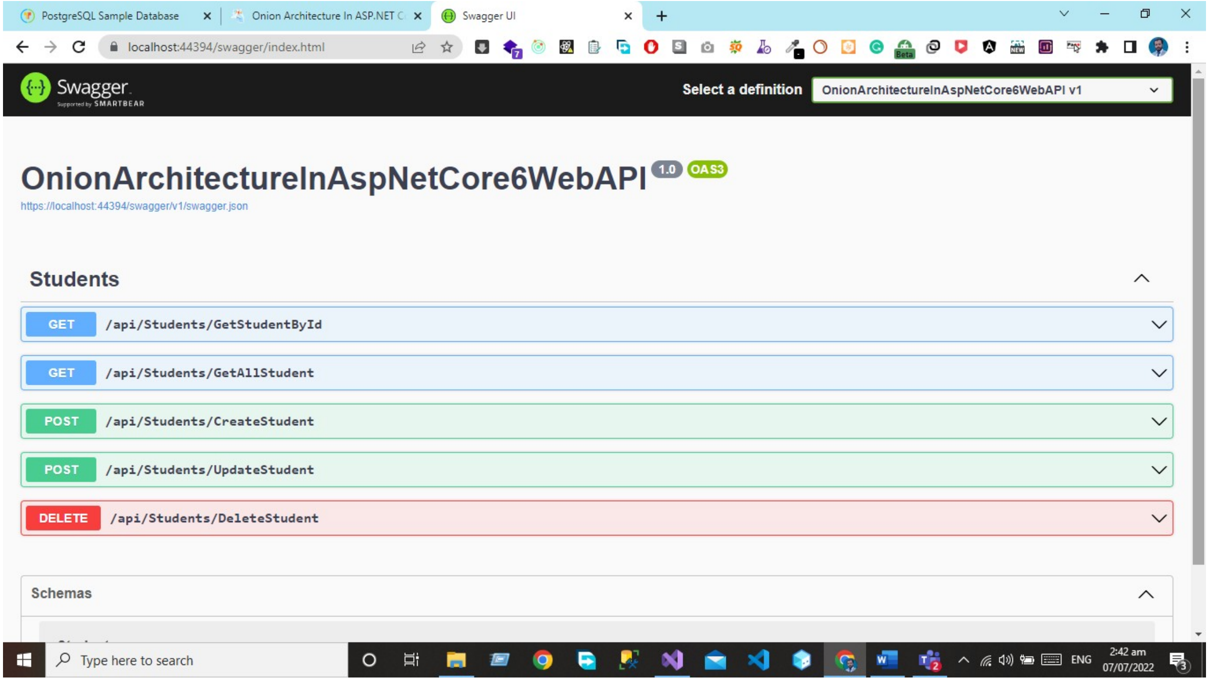
```
using DomainLayer.Data;
using DomainLayer.Models;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using RepositoryLayer.IRepository;
using ServiceLayer.ICustomServices;

namespace OnionArchitectureInAspNetCore6WebAPI.Controllers
{
   [Route("api/[controller]")]
   [ApiController]
   public class StudentsController : ControllerBase
   {
      private readonly ICustomService<Student> _customService;
      private readonly ApplicationDbContext _applicationDbContext;
      public StudentsController(ICustomService<Student> customService,
         ApplicationDbContext applicationDbContext)
      {
```

```
    _customService = customService;
    _applicationDbContext = applicationDbContext;
}

[HttpGet(nameof(GetStudentById))]
public IActionResult GetStudentById(int Id)
{
    var obj = _customService.Get(Id);
    if (obj == null)
    {
        return NotFound();
    }
    else
    {
        return Ok(obj);
    }
}
[HttpGet(nameof(GetAllStudent))]
public IActionResult GetAllStudent()
{
    var obj = _customService.GetAll();
    if(obj == null)
    {
        return NotFound();
    }
    else
    {
        return Ok(obj);
    }
}

[HttpPost(nameof(CreateStudent))]
public IActionResult CreateStudent(Student student)
{
    if (student!=null)
    {
      _customService.Insert(student);
      return Ok("Created Successfully");
    }
    else
    {
        return BadRequest("Somethingwent wrong");
```

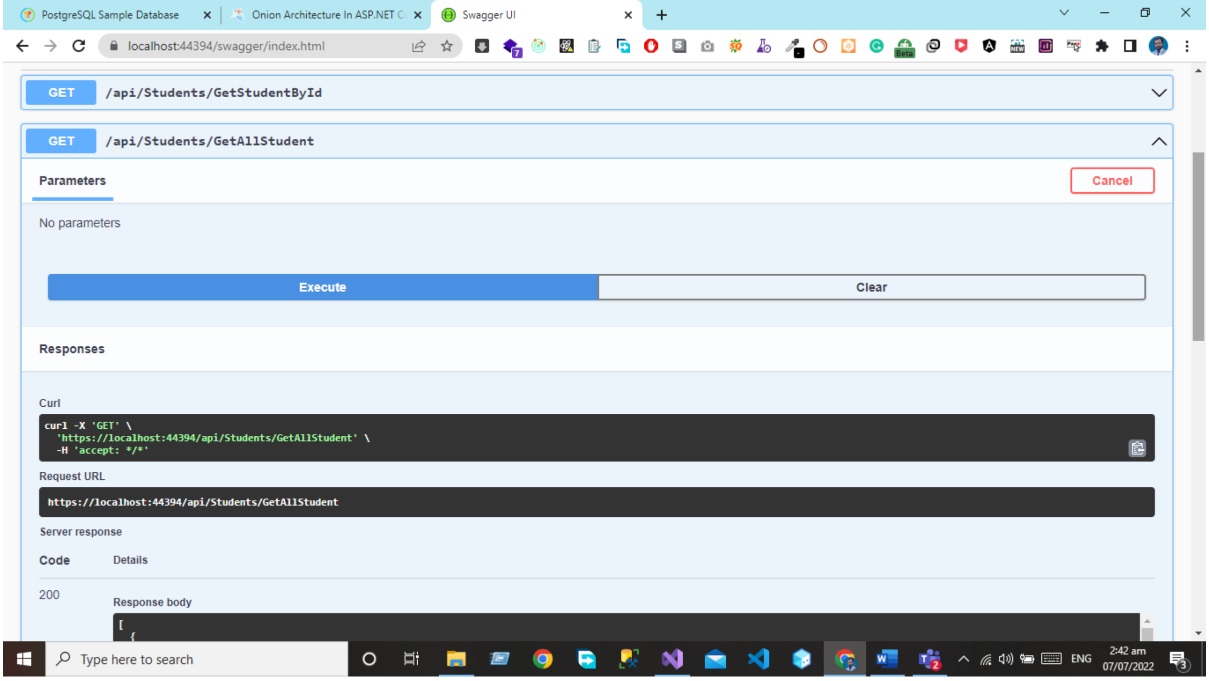
```
            }
        }

        [HttpPost(nameof(UpdateStudent))]
        public IActionResult UpdateStudent(Student student)
        {
            if(student!=null)
            {
                _customService.Update(student);
                return Ok("Updated SuccessFully");
            }
            else
            {
                return BadRequest();
            }

        }

        [HttpDelete(nameof(DeleteStudent))]
        public IActionResult DeleteStudent(Student student)
        {
            if(student!=null)
            {
                _customService.Delete(student);
                return Ok("Deleted Successfully");
            }
            else
            {
                return BadRequest("Something went wrong");
            }
        }



    }
}
```

Output

Now we will run the project and will see the output using the swagger.


PostgreSQL Sample Database
Onion Architecture In ASP.NET C
Swagger UI
localhost:44394/swagger/index.html
Swagger
Supported by SMARTBEAR
Select a definition
OnionArchitectureInAspNetCore6WebAPI v1
OnionArchitectureInAspNetCore6WebAPI
1.0
OAS3
https://localhost:44394/swagger/v1/swagger.json
Students
GET
/api/Students/GetStudentById
GET
/api/Students/GetAllStudent
POST
/api/Students/CreateStudent
POST
/api/Students/UpdateStudent
DELETE
/api/Students/DeleteStudent
Schemas
Type here to search
ENG
2:42 am
07/07/2022

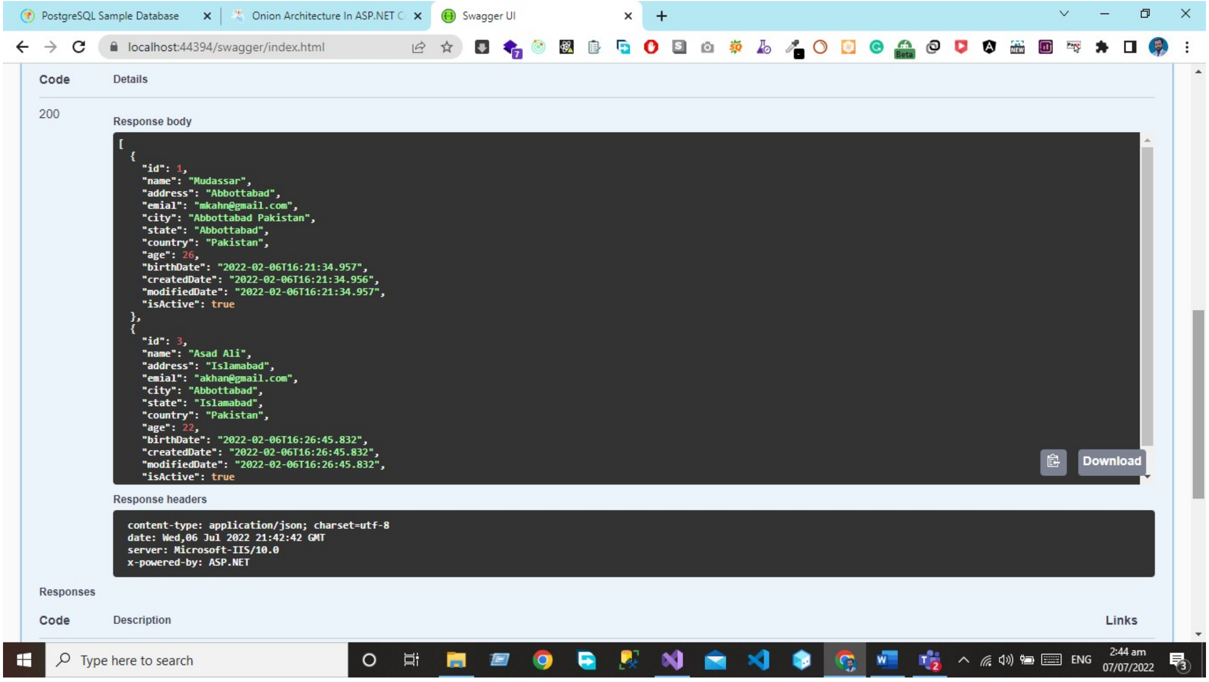
Now we can see when we hit the GetAllStudent Endpoint we can see the data of students from the database in the form of JSON projects.



```
curl -X 'GET' \
  'https://localhost:44394/api/Students/GetAllStudent' \
  -H 'accept: */*'
```

```
https://localhost:44394/api/Students/GetAllStudent
```

```
[
  {
```


PostgreSQL Sample Database
Onion Architecture In ASP.NET C
Swagger UI
localhost:44394/swagger/index.html
Download
Type here to search
ENG
2:44 am
07/07/2022

Code Details

200

Response body

```
[
  {
    "id": 1,
    "name": "Mudassar",
    "address": "Abbottabad",
    "emial": "mkahn@gmail.com",
    "city": "Abbottabad Pakistan",
    "state": "Abbottabad",
    "country": "Pakistan",
    "age": 26,
    "birthDate": "2022-02-06T16:21:34.957",
    "createdDate": "2022-02-06T16:21:34.956",
    "modifiedDate": "2022-02-06T16:21:34.957",
    "isActive": true
  },
  {
    "id": 3,
    "name": "Asad Ali",
    "address": "Islamabad",
    "emial": "akhan@gmail.com",
    "city": "Abbottabad",
    "state": "Islamabad",
    "country": "Pakistan",
    "age": 22,
    "birthDate": "2022-02-06T16:26:45.832",
    "createdDate": "2022-02-06T16:26:45.832",
    "modifiedDate": "2022-02-06T16:26:45.832",
    "isActive": true
```

Response headers

```
content-type: application/json; charset=utf-8
date: Wed,06 Jul 2022 21:42:42 GMT
server: Microsoft-IIS/10.0
x-powered-by: ASP.NET
```

Responses

Code Description Links

Conclusion

In this chapter, we have implemented the Onion architecture using the Entity Framework and Code First approach. We have now the knowledge of how the layer communicates with each other’s in onion architecture and how we can write the Generic code for the Interface repository and services. Now we can develop our project using onion architecture for API Development OR MVC Core Based projects.

Complete GitHub project URL

Click here for complete project

Chapter 4- Clean Architecture

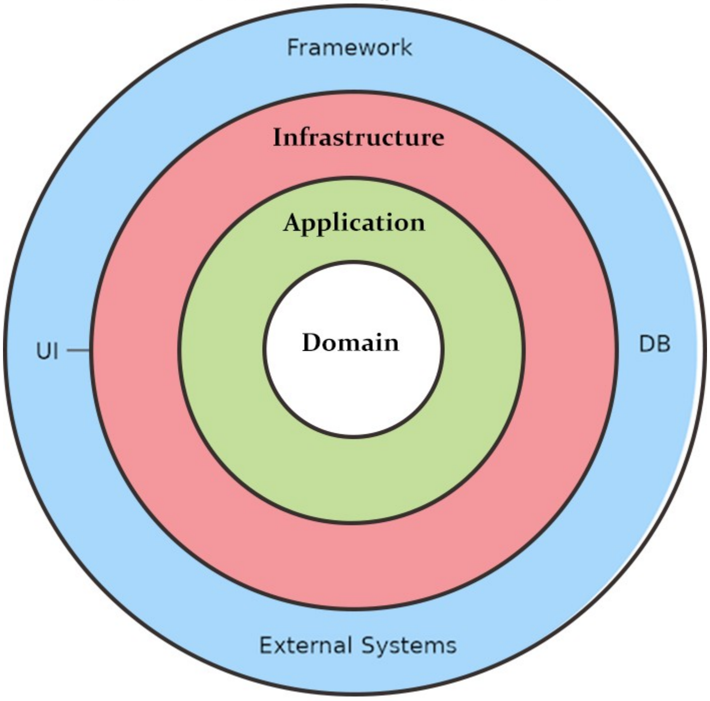
Introduction

In this chapter, we will cover clean architecture practically. Clean architecture is the software architecture that helps us to keep the entire application code under control. The main goal of the clean architecture is the code/logic, which is unlikely to change, and has to be written without any direct dependency this means that if I want to change my development framework OR User Interface (UI) of the system the core of the system should not be changed. It means that our external dependencies are completely replaceable.

What is Clean Architecture

Clean architecture has a domain layer, Application Layer, Infrastructure Layer, and Framework Layer. The domain and application layer are always the center of the design and are known as the core of the system. The core will be independent of the data access and infrastructure concerns. And we can achieve this goal by using the Interfaces and abstraction within the core system but implementing them outside of the core system.

Clean Architecture In Asp.Net Core Web API



All Rights Reserved By Sardar Mudassar Ali Khan

Layer In Clean Architecture

Clean architecture has a domain layer, Application Layer, Infrastructure Layer, and Presentation Layer. The domain and application layer are always the center of the design and are known as the core of the system.

In Clean architecture, all the dependencies of the application are Independent /Inwards, and the Core system has no dependencies on any other layer of the system. So, in the future, if we want to change the UI/ OR framework of the system we can do it easily because our all-other dependencies of the system are not dependent on the core of the system.

Domain Layer

The domain layer in the clean architecture contains the enterprise logic Like the Entities and their specifications This layer lies in the center of the architecture where we have application entities which are the application model classes or database model classes using the code first approach in the application development using Asp.net core these entities are used to create the tables in the database.

Application Layer

The application layer contains the business logic all the business logic will be written in this layer. In this layer services interfaces are kept separate from their implementation for loose coupling and separation of concerns.

Infrastructure Layer

In the infrastructure layer, we have model objects we will maintain all the database migrations and database context Objects in this layer. In this layer, we have the repositories of all the domain model objects.

Presentation Layer

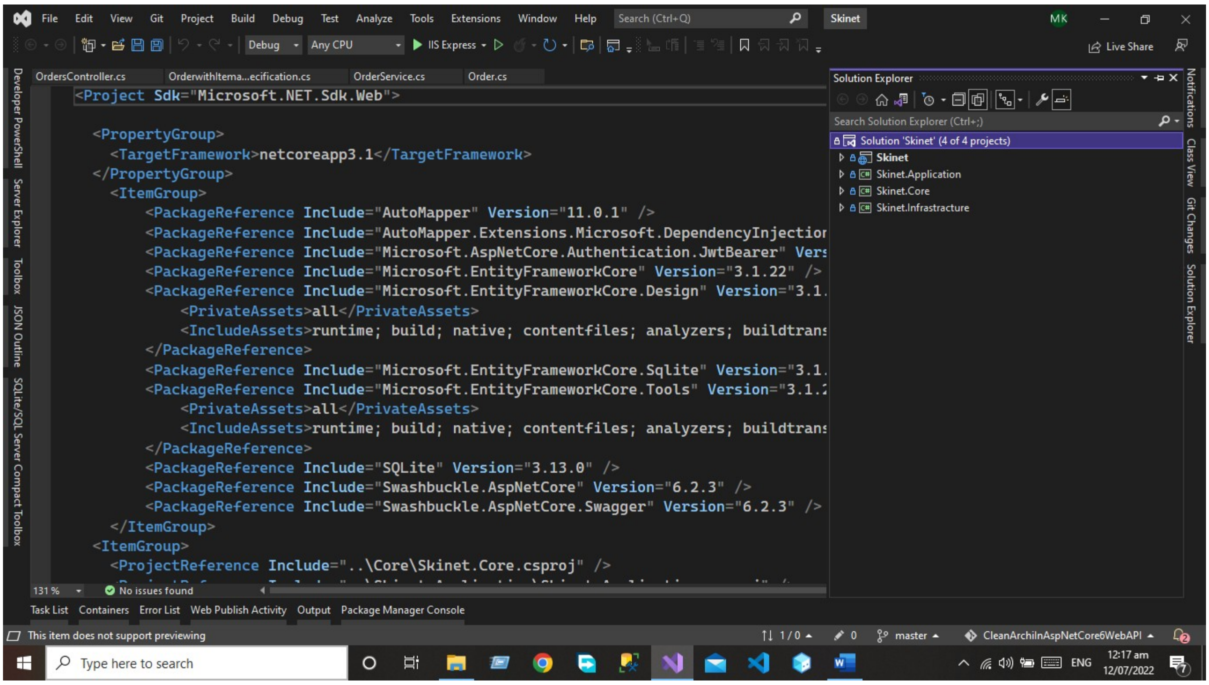
In the case of the API Presentation layer that presents us the object data from the database using the HTTP request in the form of JSON Object. But in the case of front-end applications, we present the data using the UI by consuming the APIS.

Advantages of Clean Architecture

- The immediate implementation you can implement this architecture with any programming language.
- The domain and application layer are always the center of the design and are known as the core of the system that why the core of the system is not dependent on external systems.
- This architecture allows you to change the external system without affecting the core of the system.
- In a highly testable environment, you can test your code quickly and easily.
- You can create a highly scalable and quality product.

Implementation of Clean Architecture

Now we are going to implement the clean architecture. First, you need to create the Asp.net Core API Project using the visual studio after that we will add the layer into our solution so after adding all the layers in the system our project structure will be like this.



```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>
    <ItemGroup>
        <PackageReference Include="AutoMapper" Version="11.0.1" />
        <PackageReference Include="AutoMapper.Extensions.Microsoft.DependencyInjectior
        <PackageReference Include="Microsoft.AspNetCore.Authentication.JwtBearer" Vers
        <PackageReference Include="Microsoft.EntityFrameworkCore" Version="3.1.22" />
        <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="3.1.
            <PrivateAssets>all</PrivateAssets>
            <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtrans
        </PackageReference>
        <PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite" Version="3.1.
        <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="3.1.2
            <PrivateAssets>all</PrivateAssets>
            <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtrans
        </PackageReference>
        <PackageReference Include="SQLite" Version="3.13.0" />
        <PackageReference Include="Swashbuckle.AspNetCore" Version="6.2.3" />
        <PackageReference Include="Swashbuckle.AspNetCore.Swagger" Version="6.2.3" />
    </ItemGroup>
  <ItemGroup>
    <ProjectReference Include="..\Core\Skinet.Core.csproj" />
```

Now you can see that our project structure will be like in the above picture.

Let's Implement the layer practically.

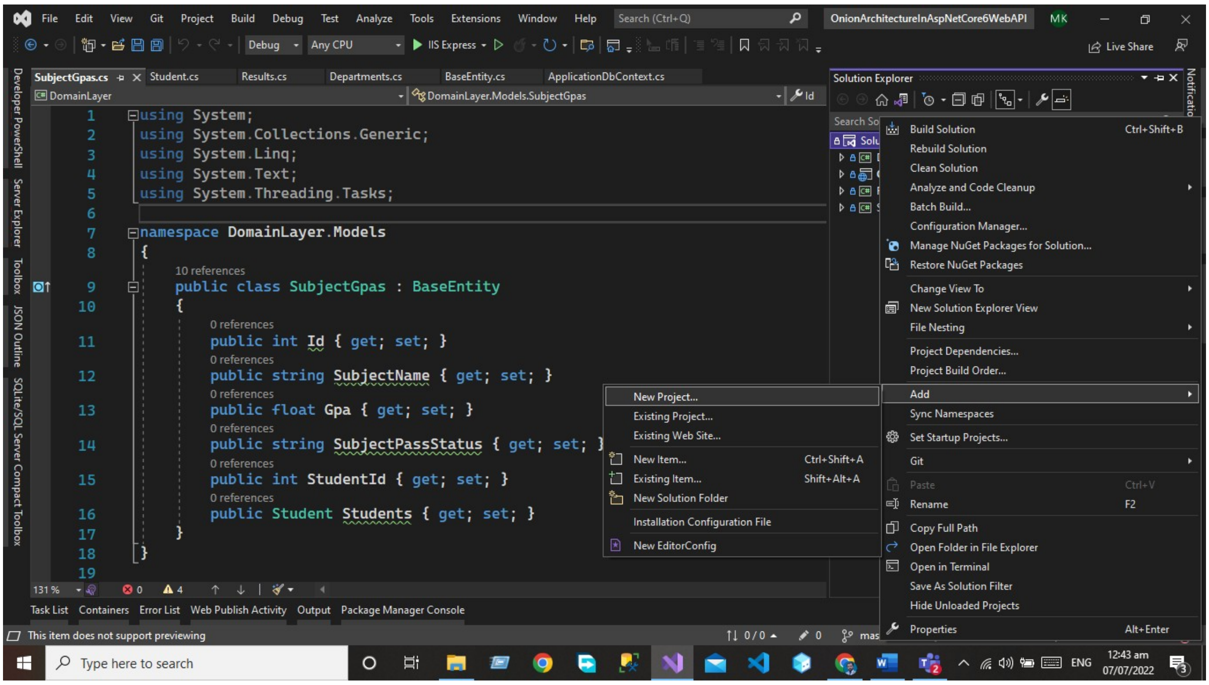
Domain Layer

The domain layer in the clean architecture contains the enterprise logic Like the Entities and their specifications This layer lies in the center of the architecture where we have application entities which are the application model classes or database model classes using the code first approach in the application development using Asp.net core these entities are used to create the tables in the database.

Implementation of Domain layer

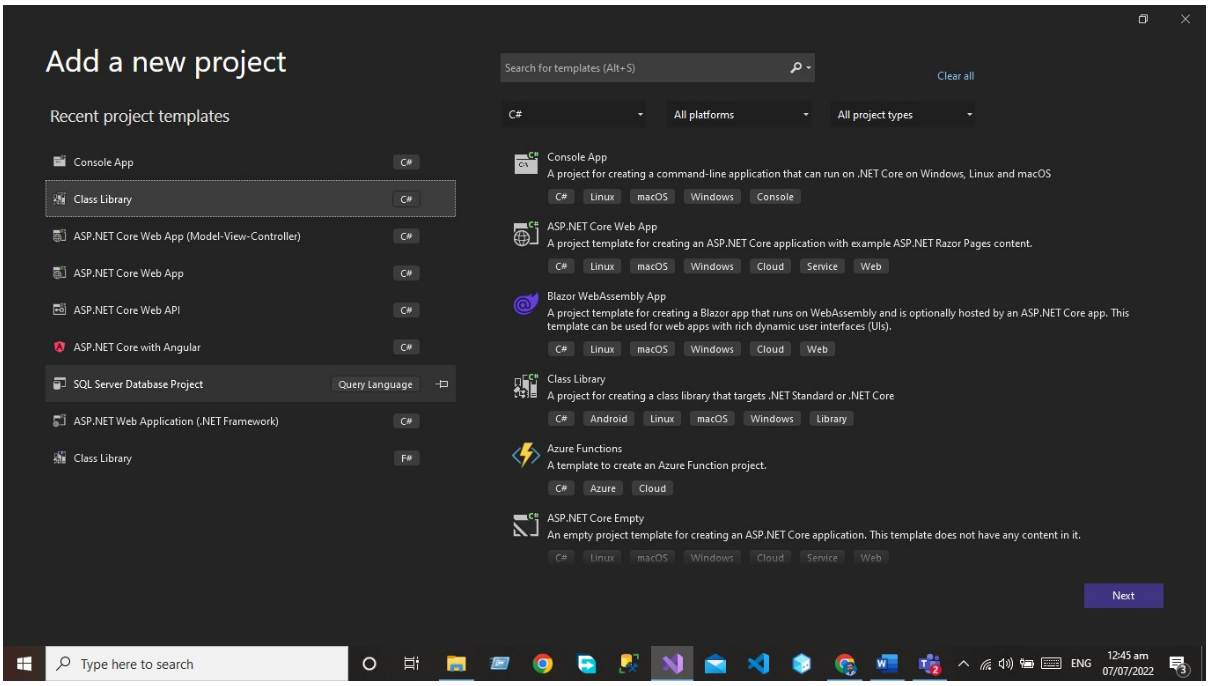
First, you need to add the library project to your system so let’s add the library project to your system.

Write click on the Solution and then click on add option.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DomainLayer.Models
{
    10 references
    public class SubjectGpas : BaseEntity
    {
        0 references
        public int Id { get; set; }
        0 references
        public string SubjectName { get; set; }
        0 references
        public float Gpa { get; set; }
        0 references
        public string SubjectPassStatus { get; set; }
        0 references
        public int StudentId { get; set; }
        0 references
        public Student Students { get; set; }
    }
}
```

Add the library project to your solution



Add a new project

Recent project templates

- Console App — C#
- Class Library — C#
- ASP.NET Core Web App (Model-View-Controller) — C#
- ASP.NET Core Web App — C#
- ASP.NET Core Web API — C#
- ASP.NET Core with Angular — C#
- SQL Server Database Project — Query Language
- ASP.NET Web Application (.NET Framework) — C#
- Class Library — F#

Search for templates (Alt+S)

Clear all

C# | All platforms | All project types

Console App
A project for creating a command-line application that can run on .NET Core on Windows, Linux and macOS
C# Linux macOS Windows Console

ASP.NET Core Web App
A project template for creating an ASP.NET Core application with example ASP.NET Razor Pages content.
C# Linux macOS Windows Cloud Service Web

Blazor WebAssembly App
A project template for creating a Blazor app that runs on WebAssembly and is optionally hosted by an ASP.NET Core app. This template can be used for web apps with rich dynamic user interfaces (UIs).
C# Linux macOS Windows Cloud Web

Class Library
A project for creating a class library that targets .NET Standard or .NET Core
C# Android Linux macOS Windows Library

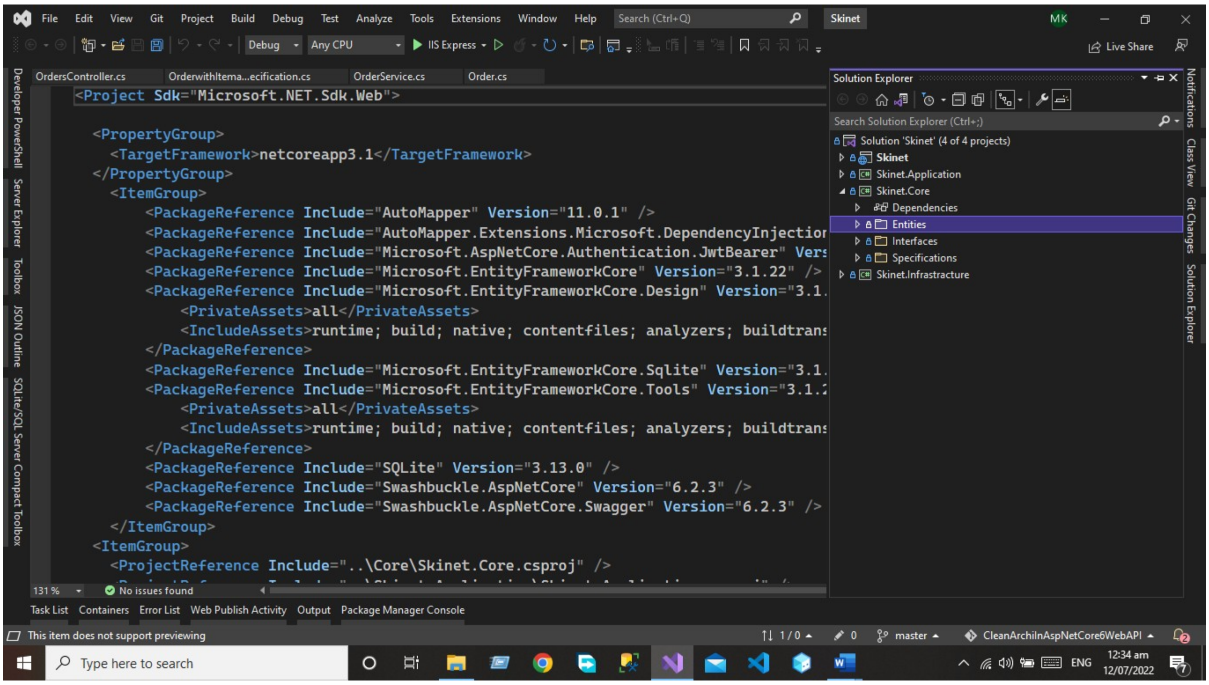
Azure Functions
A template to create an Azure Function project.
C# Azure Cloud

ASP.NET Core Empty
An empty project template for creating an ASP.NET Core application. This template does not have any content in it.
C# Linux macOS Windows Cloud Service Web

Name this project as Domain Layer

Entities Folder

First, you need to add the Models folder that will be used to create the database entities. In the Models folder, we will create the following database entities.



File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help
Search (Ctrl+Q)
Skinet
Debug
Any CPU
IIS Express
Live Share
OrdersController.cs
OrderwithItema...ecification.cs
OrderService.cs
Order.cs
Developer PowerShell
Server Explorer
Toolbox
JSON Outline
SQLite/SQL Server Compact Toolbox
Notifications
Class View
Git Changes
Solution Explorer
131 %
No issues found
Task List Containers Error List Web Publish Activity Output Package Manager Console
This item does not support previewing
1 / 0
0
master
CleanArchiInAspNetCore6WebAPI
Type here to search
ENG
12:34 am
12/07/2022

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>
    <ItemGroup>
        <PackageReference Include="AutoMapper" Version="11.0.1" />
        <PackageReference Include="AutoMapper.Extensions.Microsoft.DependencyInjectior
        <PackageReference Include="Microsoft.AspNetCore.Authentication.JwtBearer" Vers
        <PackageReference Include="Microsoft.EntityFrameworkCore" Version="3.1.22" />
        <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="3.1.
            <PrivateAssets>all</PrivateAssets>
            <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtrans
        </PackageReference>
        <PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite" Version="3.1.
        <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="3.1.2
            <PrivateAssets>all</PrivateAssets>
            <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtrans
        </PackageReference>
        <PackageReference Include="SQLite" Version="3.13.0" />
        <PackageReference Include="Swashbuckle.AspNetCore" Version="6.2.3" />
        <PackageReference Include="Swashbuckle.AspNetCore.Swagger" Version="6.2.3" />
    </ItemGroup>
  <ItemGroup>
    <ProjectReference Include="..\Core\Skinet.Core.csproj" />
```

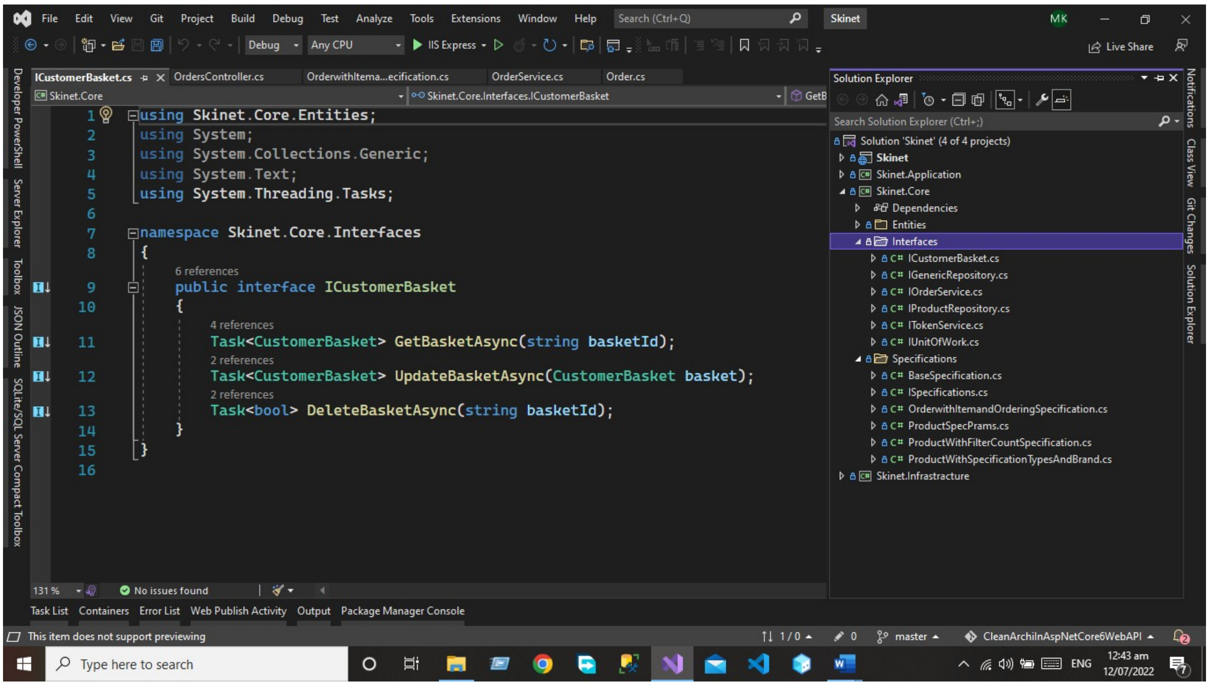
Solution Explorer

Search Solution Explorer (Ctrl+;)

- Solution 'Skinet' (4 of 4 projects)
  - Skinet
  - Skinet.Application
  - Skinet.Core
    - Dependencies
    - Entities
    - Interfaces
    - Specifications
  - Skinet.Infrastracture

Interface Folder

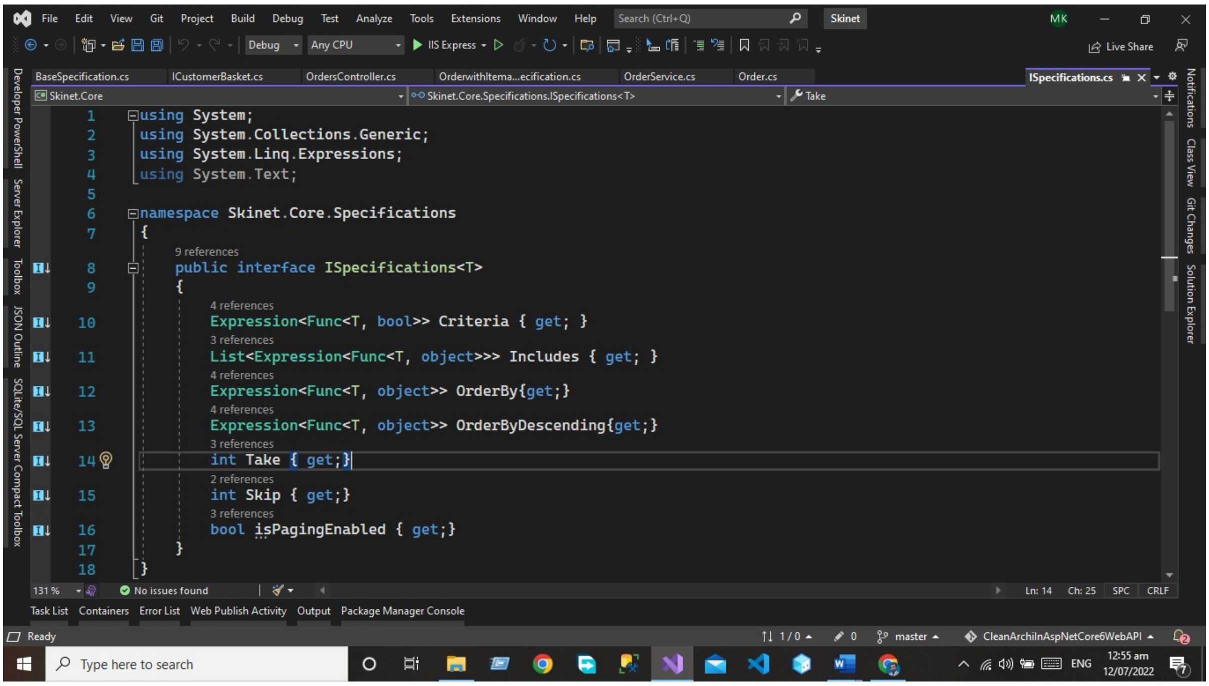
This folder is used to add the interfaces of the entities that you want to add the specific methods in your Interface.



```
using Skinet.Core.Entities;
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading.Tasks;

namespace Skinet.Core.Interfaces
{
    public interface ICustomerBasket
    {
        Task<CustomerBasket> GetBasketAsync(string basketId);
        Task<CustomerBasket> UpdateBasketAsync(CustomerBasket basket);
        Task<bool> DeleteBasketAsync(string basketId);
    }
}
```

Specification Folder

This folder is used to add all the specifications lets us take the example if you want the result of the API in ascending OR in descending Order OR want the result in the specific criteria OR want the result in the form of pagination then you need to add the specification class. Let’s take the example


File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help
Search (Ctrl+Q)
Skinet
Debug
Any CPU
IIS Express
Live Share
BaseSpecification.cs
ICustomerBasket.cs
OrdersController.cs
OrderwithItema...ecification.cs
OrderService.cs
Order.cs
ISpecifications.cs
Skinet.Core
Skinet.Core.Specifications.ISpecifications<T>
Take
131 %
No issues found
Ln: 14 Ch: 25 SPC CRLF
Task List Containers Error List Web Publish Activity Output Package Manager Console
Ready
master
CleanArchinAspNetCore6WebAPI
Type here to search
ENG
12:55 am
12/07/2022

```
using System;
using System.Collections.Generic;
using System.Linq.Expressions;
using System.Text;

namespace Skinet.Core.Specifications
{
    9 references
    public interface ISpecifications<T>
    {
        4 references
        Expression<Func<T, bool>> Criteria { get; }
        3 references
        List<Expression<Func<T, object>>> Includes { get; }
        4 references
        Expression<Func<T, object>> OrderBy{get;}
        4 references
        Expression<Func<T, object>> OrderByDescending{get;}
        3 references
        int Take { get;}
        2 references
        int Skip { get;}
        3 references
        bool isPagingEnabled { get;}
    }
}
```

Note: Complete Code of the Project will be available on My GitHub

Code Example of Specification Class

```
using System;
using System.Collections.Generic;
using System.Linq.Expressions;
using System.Text;

namespace Skinet.Core.Specifications
{
    public interface ISpecifications<T>
    {
        Expression<Func<T, bool>> Criteria { get; }
        List<Expression<Func<T, object>>> Includes { get; }
        Expression<Func<T, object>> OrderBy{get;}
        Expression<Func<T, object>> OrderByDescending{get;}
        int Take { get;}
        int Skip { get;}
        bool isPagingEnabled { get;}
    }
}
```

Base Specification Class

```
using System;
using System.Collections.Generic;
using System.Linq.Expressions;
using System.Text;

namespace Skinet.Core.Specifications
{
    public class BaseSpecification<T> : ISpecifications<T>
    {
        public Expression<Func<T, bool>> Criteria { get; }
        public BaseSpecification()
        {


        }
        public BaseSpecification(Expression<Func<T, bool>> Criteria)
        {
            this.Criteria = Criteria;
        }
        public List<Expression<Func<T, object>>> Includes { get; }
        = new List<Expression<Func<T, object>>>();

        public Expression<Func<T, object>> OrderBy { get; private set; }

        public Expression<Func<T, object>> OrderByDescending { get; private set; }

        public int Take { get; private set; }

        public int Skip { get; private set; }

        public bool isPagingEnabled { get; private set; }

        protected void AddInclude(Expression<Func<T,object>> includeExpression)
        {
            Includes.Add(includeExpression);
        }

        public void AddOrderBy(Expression<Func<T, object>> OrderByexpression)
        {
            OrderBy = OrderByexpression;
        }
        public void AddOrderByDecending(Expression<Func<T, object>> OrderByDecending)
        {
            OrderByDescending = OrderByDecending;
        }
        public void ApplyPagging(int take, int skip)
        {
            Take = take;
            //Skip = skip;
            isPagingEnabled = true;
        }
    }
}
```

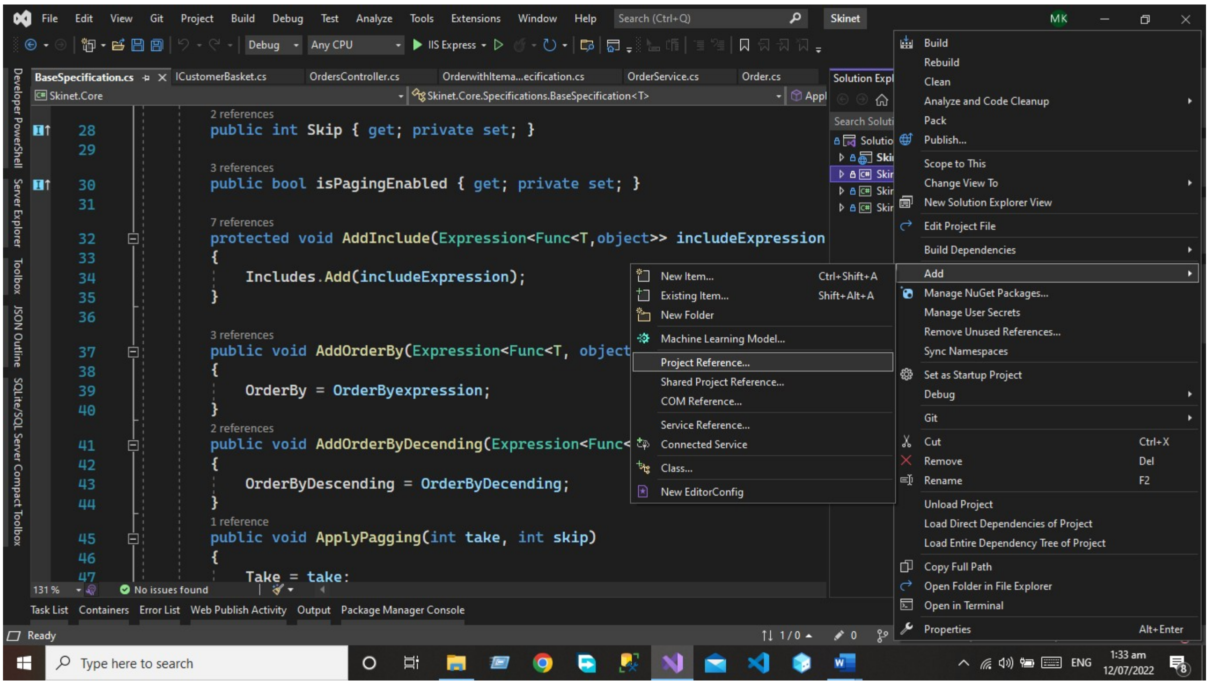
Application Layer

The application layer contains the business logic all the business logic will be written in this layer. In this layer services interfaces are kept separate from their implementation for loose coupling and separation of concerns.

Now we will add the application layer to our application

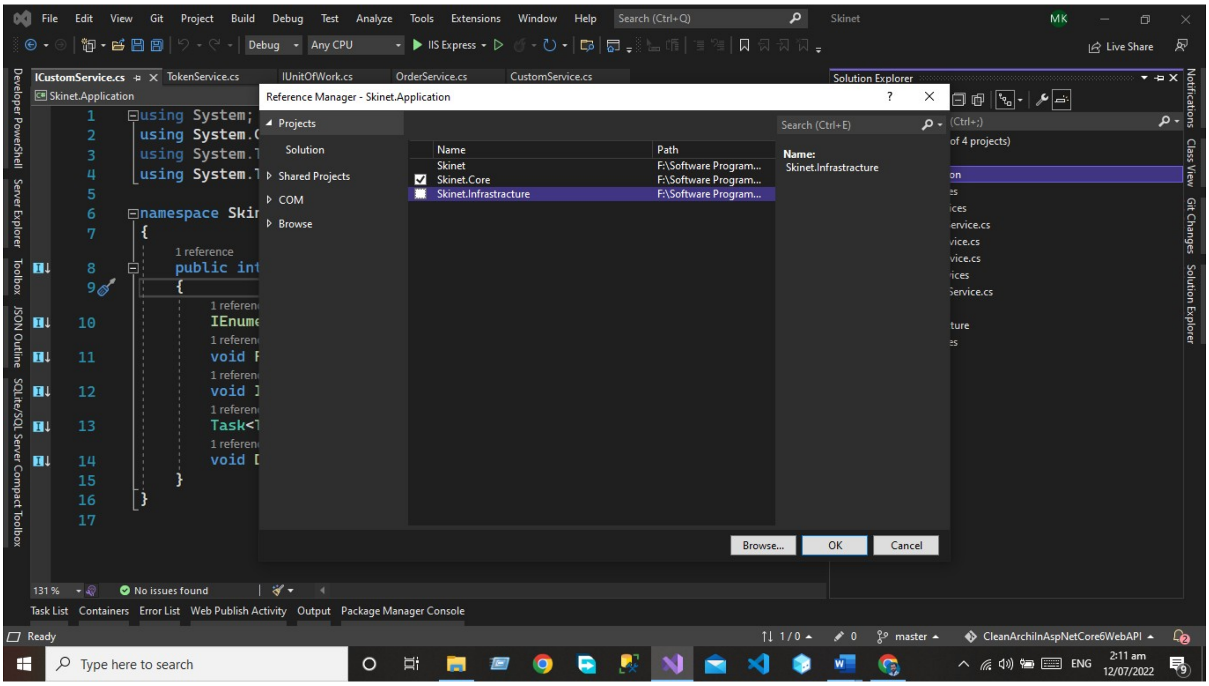
Now we will work on the application Layer we will follow the same process as we did for the Domain layer add the library project in your applicant and give a name to that project Repository layer.

But here we need to add the project reference of the Domain layer in the application layer. Write click on the project and then click the Add button after that we will add the project references in the application layer.



```
        2 references
        public int Skip { get; private set; }

        3 references
        public bool isPagingEnabled { get; private set; }

        7 references
        protected void AddInclude(Expression<Func<T,object>> includeExpression
        {
            Includes.Add(includeExpression);
        }

        3 references
        public void AddOrderBy(Expression<Func<T, object
        {
            OrderBy = OrderByexpression;
        }
        2 references
        public void AddOrderByDecending(Expression<Func<
        {
            OrderByDescending = OrderByDecending;
        }
        1 reference
        public void ApplyPagging(int take, int skip)
        {
            Take = take;
```

Now select the core project to add the project reference of the domain layer.



☐	Name	Path
	Skinet	F:\Software Program...
☑	Skinet.Core	F:\Software Program...
☐	Skinet.Infrastructure	F:\Software Program...

Name:
Skinet.Infrastructure

```
using System;
using System.(
using System.T
using System.T

namespace Skir
{
    1 reference
    public int
    {
        1 referen
        IEnume
        1 referen
        void F
        1 referen
        void I
        1 referen
        Task<T
        1 referen
        void [
    }
}
```

Click the OK button After that our project reference will be added to our system.

Now let’s Create the Desired Folders in our Projects

I Custom Services

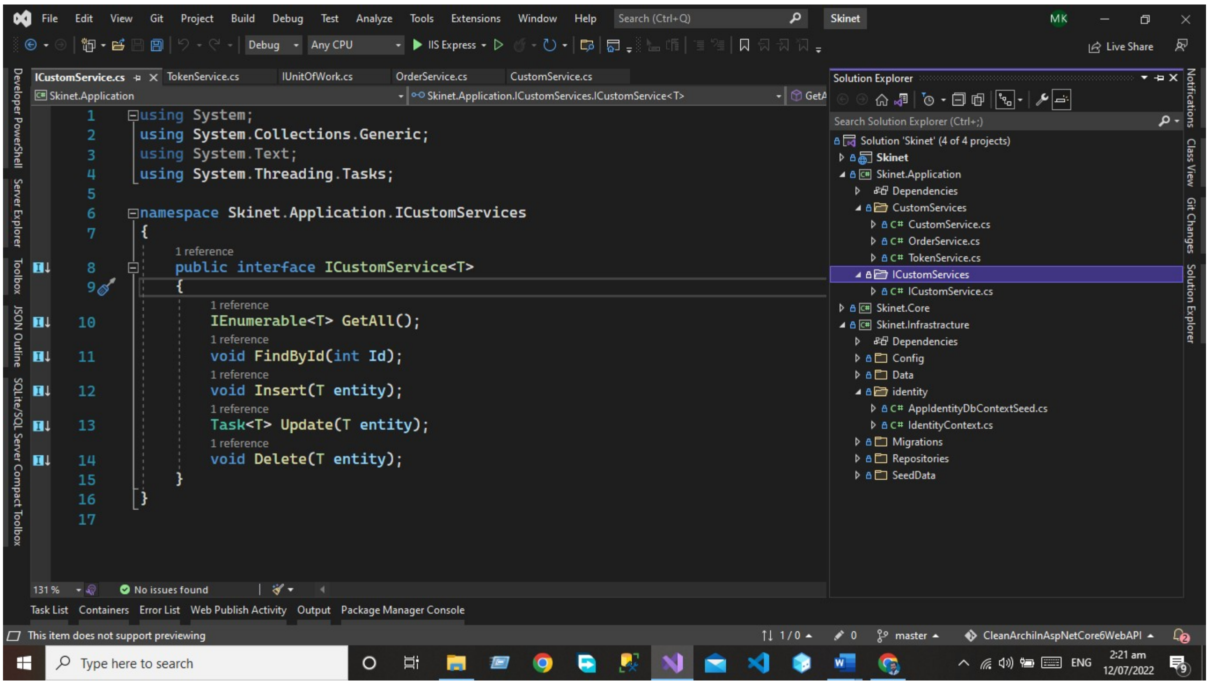
Let’s add the ICustom services folder in our application in this folder we will add the ICustom Service Interfaced that will be Inherited by all the services we will add in our Customer Service folder.

Note: Complete Code of the Project will be available on My GitHub

Let's add some services to our project

Code Of the Custom Service

Let us add the Token service that inherits the token service interface from the Core



```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading.Tasks;

namespace Skinet.Application.ICustomServices
{
    1 reference
    public interface ICustomService<T>
    {
        1 reference
        IEnumerable<T> GetAll();
        1 reference
        void FindById(int Id);
        1 reference
        void Insert(T entity);
        1 reference
        Task<T> Update(T entity);
        1 reference
        void Delete(T entity);
    }
}
```

Code Of ICustom Interface

```
 using System;
using System.Collections.Generic;
using System.Text;
using System.Threading.Tasks;

namespace Skynet.Application.ICustomServices
{
    public interface ICustomService<T>
    {
        IEnumerable<T> GetAll();
        void FindById(int Id);
        void Insert(T entity);
        Task<T> Update(T entity);
        void Delete(T entity);
    }
}
```

Code of the ICustomerBasket

```
using Skinet.Core.Entities;
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading.Tasks;

namespace Skinet.Core.Interfaces
{
    public interface ICustomerBasket
    {
        Task<CustomerBasket> GetBasketAsync(string basketId);
        Task<CustomerBasket> UpdateBasketAsync(CustomerBasket basket);
        Task<bool> DeleteBasketAsync(string basketId);
    }
}
```

Custom Services Folder

This folder will be used to add the custom services to our system and lets us create some custom services for our project so that our concept will be clear about Custom services. All the custom services will inherit the I Custom services interface using that interface we will add the CRUD Operation in our system.

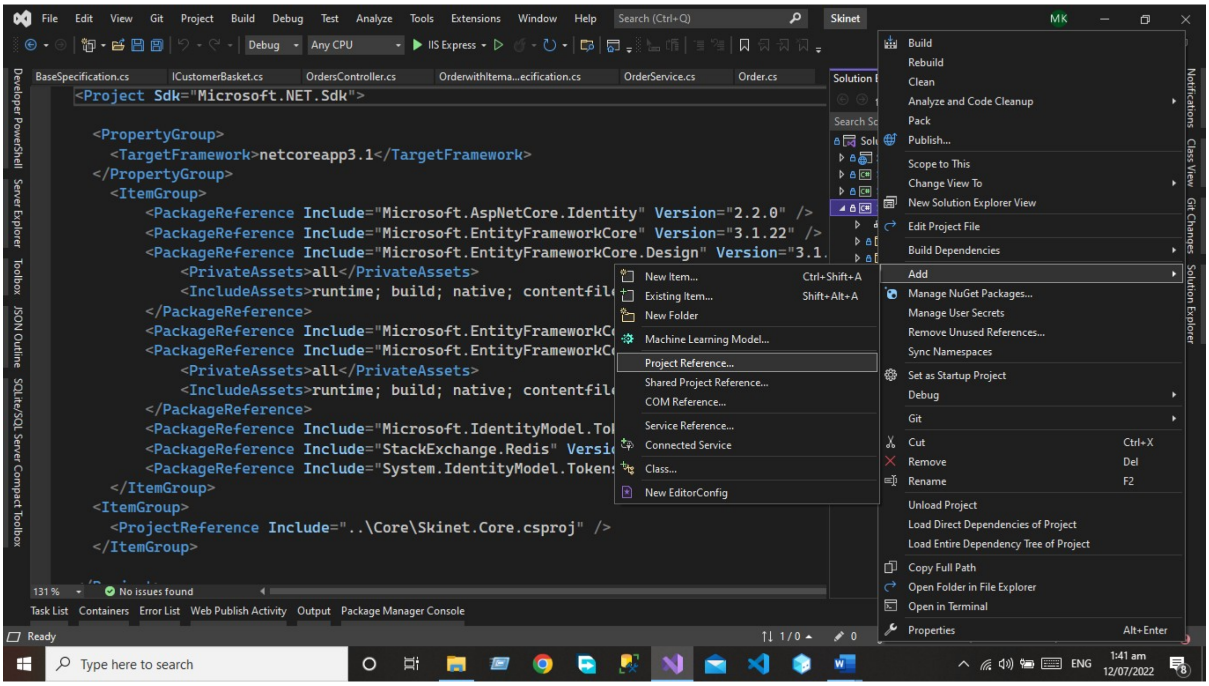
Infrastructure Layer

In the infrastructure layer, we have model objects we will maintain all the database migrations and database context Objects in this layer. In this layer, we have the repositories of all the domain model objects.

Now we will add the infrastructure layer to our application

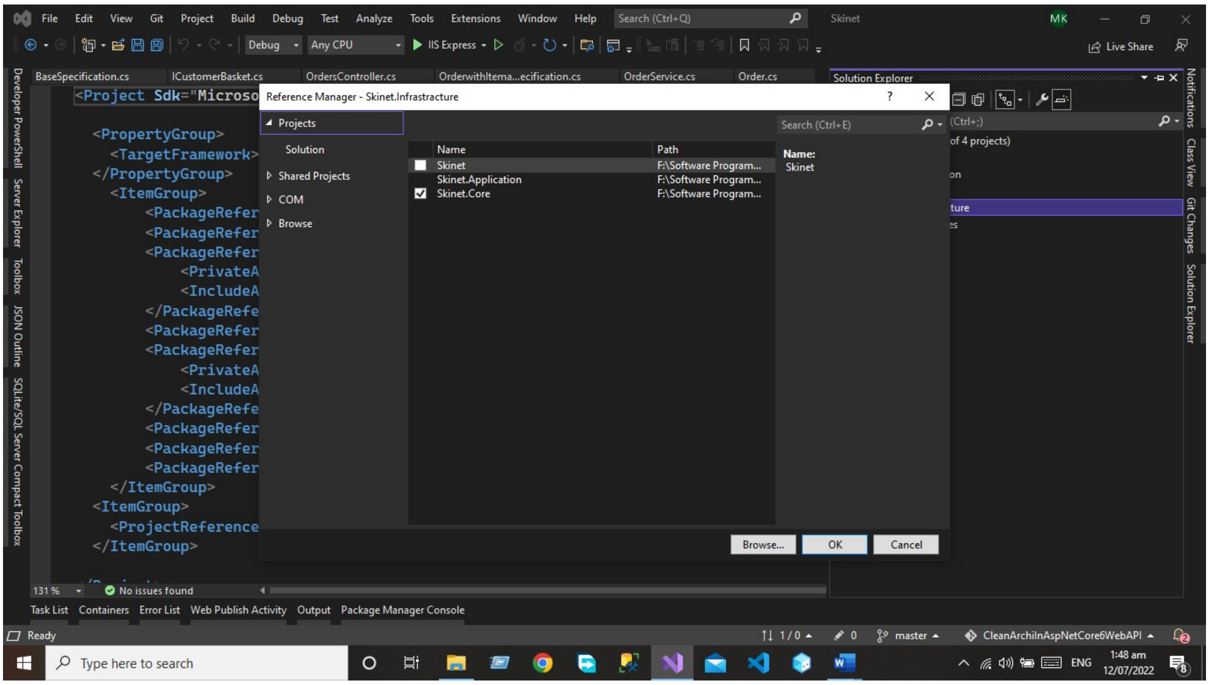
Now we will work on the infrastructure Layer we will follow the same process as we did for the Domain layer add the library project in your applicant and give a name to that project infrastructure layer.

But here we need to add the project reference of the Domain layer in the infrastructure layer. Write click on the project and then click the Add button after that we will add the project references in the infrastructure layer.



```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>
    <ItemGroup>
        <PackageReference Include="Microsoft.AspNetCore.Identity" Version="2.2.0" />
        <PackageReference Include="Microsoft.EntityFrameworkCore" Version="3.1.22" />
        <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="3.1.
            <PrivateAssets>all</PrivateAssets>
            <IncludeAssets>runtime; build; native; contentfil
        </PackageReference>
        <PackageReference Include="Microsoft.EntityFrameworkC
        <PackageReference Include="Microsoft.EntityFrameworkC
            <PrivateAssets>all</PrivateAssets>
            <IncludeAssets>runtime; build; native; contentfil
        </PackageReference>
        <PackageReference Include="Microsoft.IdentityModel.Tol
        <PackageReference Include="StackExchange.Redis" Versic
        <PackageReference Include="System.IdentityModel.Token
    </ItemGroup>
  <ItemGroup>
    <ProjectReference Include="..\Core\Skinet.Core.csproj" />
  </ItemGroup>
```

Now select the Core for adding the domain layer reference in our project.



Reference Manager - Skinet.Infrastructure

	Name	Path
□	Skinet	F:\Software Program...
	Skinet.Application	F:\Software Program...
☑	Skinet.Core	F:\Software Program...

Name:
Skinet

Implementation of Infrastructure Layer

Let’s implement the infrastructure layer in our system.

First, you need to add the Data folder to your infrastructure layer.

Data Folder

Add the Data folder in the infrastructure layer that is used to add the database context class. The database context class is used to maintain the session with the underlying database using which you can perform the CRUD operation.

In our project, we will add the store context class that will handle the session with our database.

Code of the Database Context Class

```
  using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage.ValueConversion;
using Skinet.Core.Entities;
using Skinet.Core.Entities.OrderAggregate;
using System;
using System.Linq;
using System.Reflection;

namespace Skinet.Infrastracture.Data
{
    public class StoreContext : DbContext
    {
        public StoreContext(DbContextOptions<StoreContext> options) : base(options)
        {


        }
        public DbSet<Products> Products { get; set; }
        public DbSet<ProductType> ProductTypes { get; set; }
        public DbSet<ProductBrand> ProductBrands { get; set; }
        public DbSet<Order> Orders { get; set; }
        public DbSet<DeliveryMethod> DeliveryMethods { get; set; }
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);
            modelBuilder.ApplyConfigurationsFromAssembly(Assembly.GetExecutingAssembly());
            if(Database.ProviderName =="Microsoft.EntityFramework.Sqlite")
            {
                foreach (var entity in modelBuilder.Model.GetEntityTypes())
                {
                    var properties = entity.ClrType.GetProperties()
                        .Where(p => p.PropertyType == typeof(decimal));
                    var dateandtimepropertise = entity.ClrType.GetProperties()
                        .Where(t => t.PropertyType == typeof(DateTimeOffset));
                    foreach (var property in properties)
                    {
                        modelBuilder.Entity(entity.Name).Property(property.Name)
                            .HasConversion<double>();
                    }
                    foreach (var property in dateandtimepropertise)
                    {
                        modelBuilder.Entity(entity.Name).Property(property.Name)
                            .HasConversion(new DateTimeOffsetToBinaryConverter());
                    }
                }
            }
        }



    }
}
```

Repositories Folder

The repositories folder is used to add the repositories of the domain classes because we are going to implement the repository pattern in our solution.

Note: Complete Code of the Project will be available on My GitHub

Let's use add some repositories to our solution so that our concept about the repositories will be clear.

Basket Repository

Let's create the basket as an example to clarify the concept of repositories. That will inherit the Interface of IBasket Interface from the core layer.

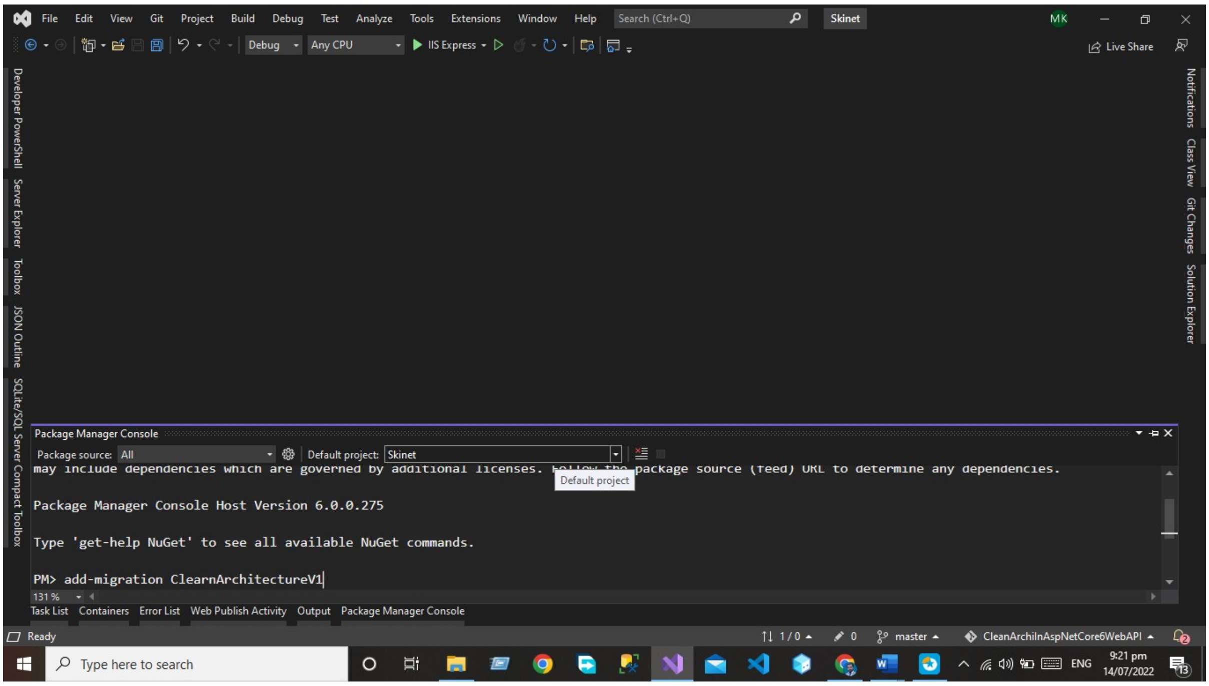
```
using Skinet.Core.Entities;
using Skinet.Core.Interfaces;
using StackExchange.Redis;
using System;
using System.Collections.Generic;
using System.Text;
using System.Text.Json;
using System.Threading.Tasks;
```

```
namespace Skinet.Infrastracture.Repositories
{
    public class BasketRepository : ICustomerBasket
    {
        private readonly IDatabase _database;
        public  BasketRepository(IConnectionMultiplexer  radis)
        {
            _database= radis.GetDatabase();
        }
        public async Task<bool> DeleteBasketAsync(string basketId)
        {
            return await _database.KeyDeleteAsync(basketId);
        }

        public async Task<CustomerBasket> GetBasketAsync(string basketId)
        {
            var data = await _database.StringGetAsync(basketId);
            return data.IsNullOrEmpty ? null : JsonSerializer.Deserialize<CustomerBasket>(data);
        }

        public async Task<CustomerBasket> UpdateBasketAsync(CustomerBasket basket)
        {
            var created = await _database.StringSetAsync(basket.Id,
                JsonSerializer.Serialize(basket),TimeSpan.FromDays(15));
            if (!created)
            {
                return null;
            }
            return await GetBasketAsync(basket.Id);
        }
    }
}
```

Migrations

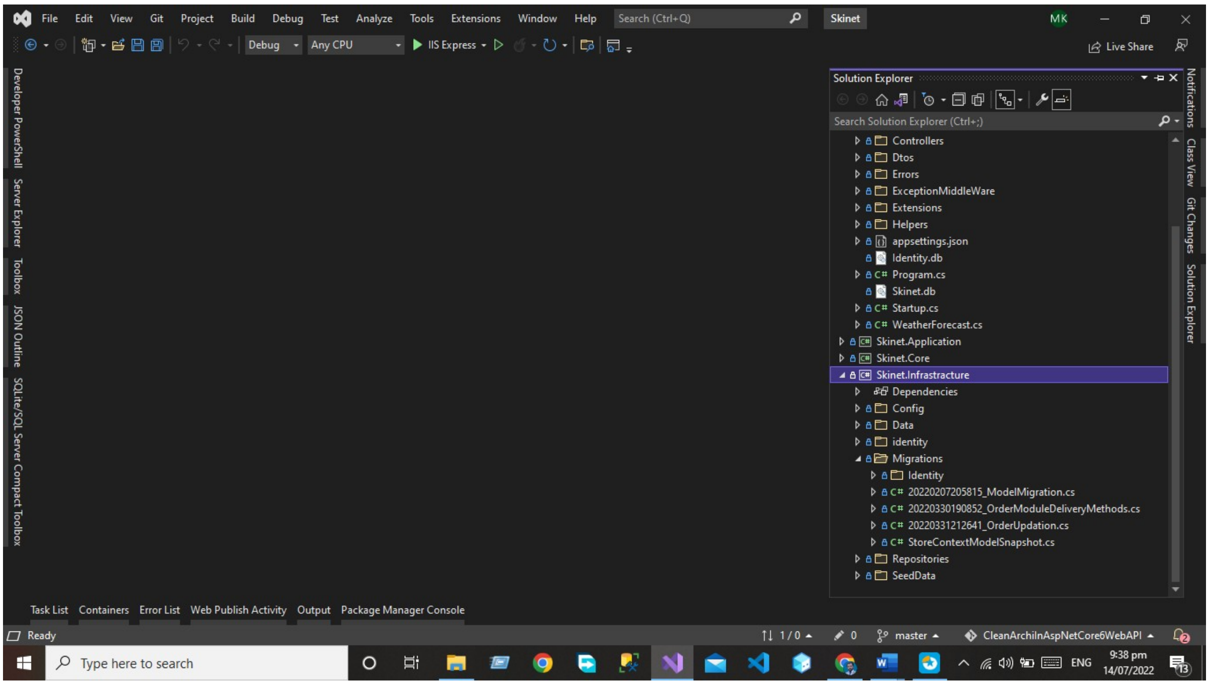
After Adding the DbSet properties we need to add the migration using the package manager console and run the command Add-Migration.

```
add-migration ClearnArchitectureV1
```



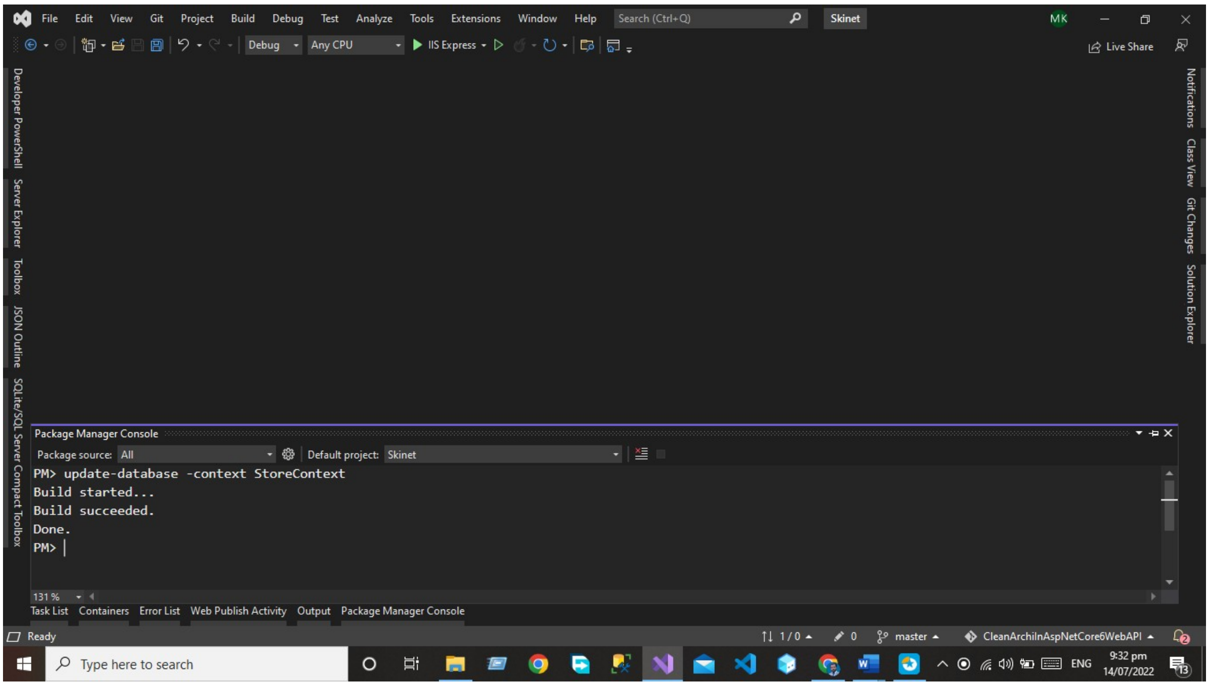
```
may include dependencies which are governed by additional licenses. Follow the package source (feed) URL to determine any dependencies.

Package Manager Console Host Version 6.0.0.275

Type 'get-help NuGet' to see all available NuGet commands.

PM> add-migration ClearnArchitectureV1
```

After executing the Add-Migration Command we have the following auto-generated classes in our migration folder.



```
Controllers
Dtos
Errors
ExceptionMiddleWare
Extensions
Helpers
appsettings.json
Identity.db
Program.cs
Skinet.db
Startup.cs
WeatherForecast.cs
Skinet.Application
Skinet.Core
Skinet.Infrastracture
  Dependencies
  Config
  Data
  identity
  Migrations
    Identity
    20220207205815_ModelMigration.cs
    20220330190852_OrderModuleDeliveryMethods.cs
    20220331212641_OrderUpdation.cs
    StoreContextModelSnapshot.cs
  Repositories
  SeedData
```

After executing the commands now, you need to update the database by executing the **update-database** Command



```
PM> update-database -context StoreContext
Build started...
Build succeeded.
Done.
PM> |
```

Presentation Layer

The presentation layer is our final layer that presents the data to the front-end user on every HTTP request.

In the case of the API presentation layer that presents us the object data from the database using the HTTP request in the form of JSON Object. But in the case of front-end applications, we present the data using the UI by consuming the APIS.

Note: Complete Code of the Project will be available on My GitHub

Extensions Folder

Extensions Folder is used for extension methods / Classes we extend functionality C# extension method is a static method of a static class, where the "this" modifier is applied to the first parameter. The type of the first parameter will be the type that is extended. Extension methods are only in scope when you explicitly import the namespace into your source code with a using directive.

Let's Create the static ApplicationServicesExtensions class where we will create the extension method for registering all services we have created during the entire project

Now we need to add the dependency Injection of our all services in the Extensions Classes and then we will use these extensions classes and methods in our startup class.

Code of Extension Method

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.DependencyInjection;
using Skinet.Application.CustomServices;
using Skinet.Core.Interfaces;
using Skinet.Errors;
using Skinet.Infrastructure.Data;
using Skinet.Infrastructure.Repositories;
using Skinet.Infrastructure.SeedData;
using System.Linq;

namespace Skinet.Controllers.Extensions
{
    public static class ApplicationServicesExtensions
    {
        public static IServiceCollection AddApplicationServices(this IServiceCollection services)
        {
            services.AddScoped<ITokenService, TokenService>();
            services.AddScoped<StoreContext, StoreContext>();
            services.AddScoped<StoreContextSeed, StoreContextSeed>();
            services.AddScoped<IProductRepository, ProductRepository>();
            services.AddScoped<ICustomerBasket, BasketRepository>();
            services.AddScoped<IUnitOfWork, UnitOfWork>();
            services.AddScoped<IOrderService, OrderService>();
            services.AddScoped(typeof(IGenericRepository<>), typeof(GenericRepository<>));
            services.Configure<ApiBehaviorOptions>(options =>
            options.InvalidModelStateResponseFactory = ActionContext =>
              {
                  var error = ActionContext.ModelState
                              .Where(e => e.Value.Errors.Count > 0)
                              .SelectMany(e => e.Value.Errors)
                              .Select(e => e.ErrorMessage).ToArray();
                  var errorresponce = new APIValidationErrorResponce
                  {
                      Errors = error
                  };
                  return new BadRequestObjectResult(error);
              }
            );
            return services;
        }
    }
}
```

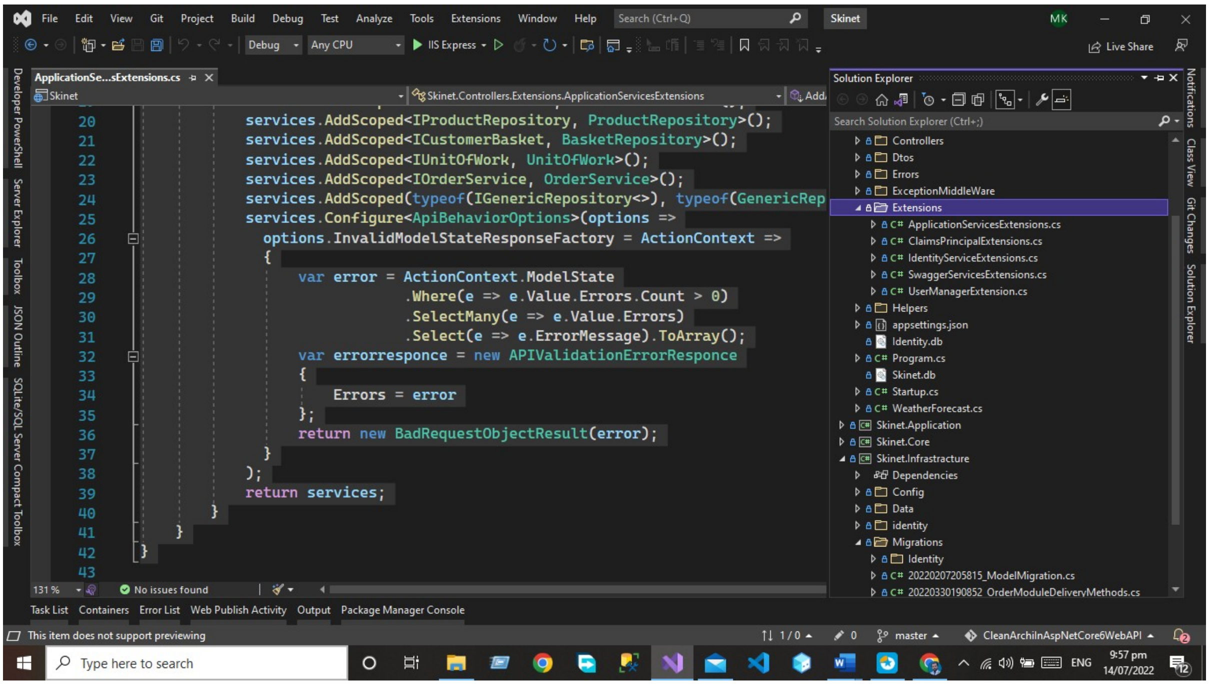
Modify The Startup.cs Class

In Startup.cs Class we will use our extensions method that we have created in extensions folders.

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Skinet.Core.Entities;
using Skinet.Core.Interfaces;
using Skinet.Infrastracture.Data;
using Skinet.Infrastracture.Repositories;
using Microsoft.EntityFrameworkCore;
using Skinet.Infrastracture.SeedData;
using Skinet.Helpers;
using Skinet.ExceptionMiddleWare;
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using Skinet.Errors;
using Microsoft.OpenApi.Models;
using Skinet.Controllers.Extensions;
using StackExchange.Redis;
using Skinet.Infrastracture.identity;
using Skinet.Extensions;

namespace Skinet
{
    public class Startup
    {
        public IConfiguration Configuration { get; }

        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }
        // This method gets called by the runtime. Use this method to add services to the container.
        public void ConfigureServices(IServiceCollection services)
```

```
{
  services.AddDbContext<StoreContext>(options =>
  options.UseSqlite(Configuration.GetConnectionString("DefaultConnection")));
  services.AddDbContext<IdentityContext>(options =>
  options.UseSqlite(Configuration.GetConnectionString("DefaultIdentityConnection")));
  services.AddSingleton<IConnectionMultiplexer>(c =>
  {
    var configuration = ConfigurationOptions.
    Parse(Configuration.GetConnectionString("Radis"), true);
    return ConnectionMultiplexer.Connect(configuration);
  });

  services.AddAutoMapper(typeof(MappingProfiles));
  services.AddControllers();
  services.AddApplicationServices();
  services.AddSwaggerDocumentation();
  services.AddIdentityService(Configuration);
  services.AddCors(options =>
  {
     options.AddPolicy("CorsPolicy", policy =>
     {
       policy.AllowAnyHeader().AllowAnyMethod().WithOrigins("*");
     });
  });
}

// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
  app.UseMiddleware<ExceptionMiddle>();
  app.UseStatusCodePagesWithReExecute("/error/{0}");
  app.UseCors("CorsPolicy");
  app.UseHttpsRedirection();
  app.UseSwaggerGen();
  app.UseRouting();
  app.UseStaticFiles();
  app.UseAuthentication();
  app.UseAuthorization();

  app.UseEndpoints(endpoints =>
  {
    endpoints.MapControllers();
```

```
            });
        }
    }
}
```



```
                services.AddScoped<IProductRepository, ProductRepository>();
                services.AddScoped<ICustomerBasket, BasketRepository>();
                services.AddScoped<IUnitOfWork, UnitOfWork>();
                services.AddScoped<IOrderService, OrderService>();
                services.AddScoped(typeof(IGenericRepository<>), typeof(GenericRep
                services.Configure<ApiBehaviorOptions>(options =>
                  options.InvalidModelStateResponseFactory = ActionContext =>
                  {
                      var error = ActionContext.ModelState
                                  .Where(e => e.Value.Errors.Count > 0)
                                  .SelectMany(e => e.Value.Errors)
                                  .Select(e => e.ErrorMessage).ToArray();
                      var errorresponce = new APIValidationErrorResponce
                      {
                          Errors = error
                      };
                      return new BadRequestObjectResult(error);
                  }
                );
                return services;
            }
        }
    }
```

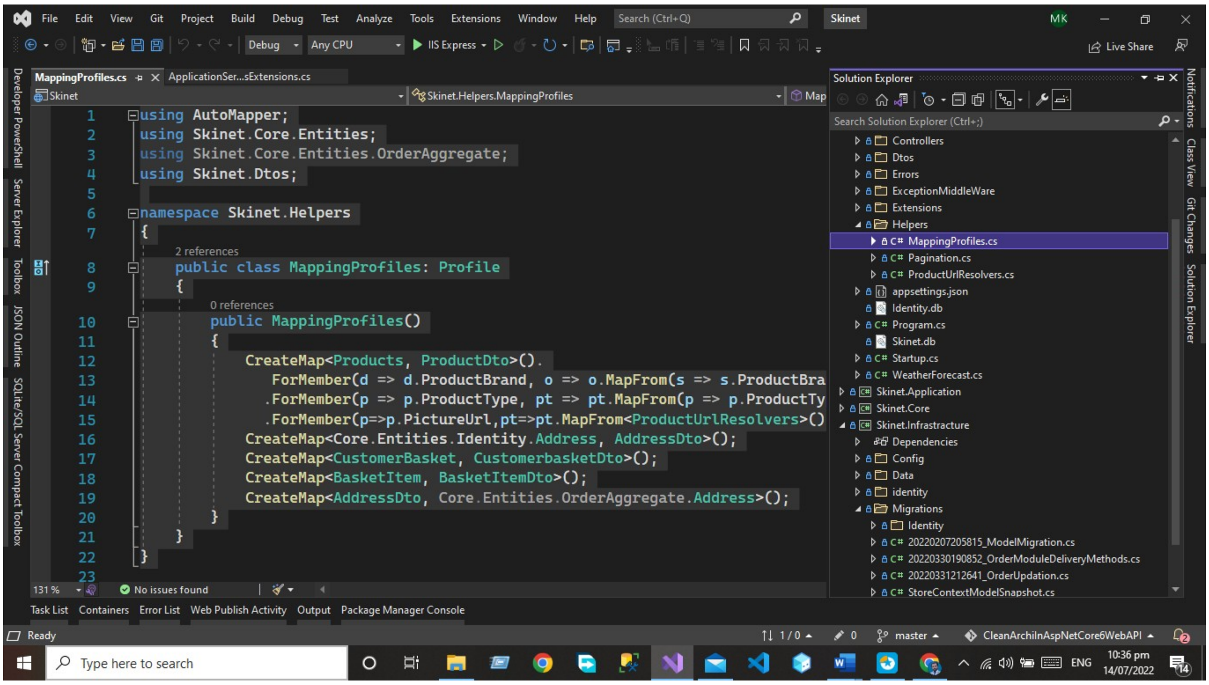
File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help
Search (Ctrl+Q)
Skinet
Debug
Any CPU
IIS Express
Live Share
ApplicationSe...sExtensions.cs
Skinet
Skinet.Controllers.Extensions.ApplicationServicesExtensions
Solution Explorer
Search Solution Explorer (Ctrl+;)
Controllers
Dtos
Errors
ExceptionMiddleWare
Extensions
ApplicationServicesExtensions.cs
ClaimsPrincipalExtensions.cs
IdentityServiceExtensions.cs
SwaggerServicesExtensions.cs
UserManagerExtension.cs
Helpers
appsettings.json
Identity.db
Program.cs
Skinet.db
Startup.cs
WeatherForecast.cs
Skinet.Application
Skinet.Core
Skinet.Infrastructure
Dependencies
Config
Data
identity
Migrations
Identity
20220207205815_ModelMigration.cs
20220330190852_OrderModuleDeliveryMethods.cs
131 %
No issues found
Task List Containers Error List Web Publish Activity Output Package Manager Console
This item does not support previewing
master
CleanArchitectureAspNetCore6WebAPI
Type here to search
ENG
9:57 pm
14/07/2022

Helpers

Helpers’ folder here we create the auto mapper profiles for mapping the entities with each other.

Code Example of Auto Mapper

```
using AutoMapper;
using Skinet.Core.Entities;
using Skinet.Core.Entities.OrderAggregate;
using Skinet.Dtos;

namespace Skinet.Helpers
{
    public class MappingProfiles: Profile
    {
        public MappingProfiles()
        {
            CreateMap<Products, ProductDto>().
              ForMember(d => d.ProductBrand, o => o.MapFrom(s => s.ProductBrand.Name))
             .ForMember(p => p.ProductType, pt => pt.MapFrom(p => p.ProductType.Name))
             .ForMember(p=>p.PictureUrl,pt=>pt.MapFrom<ProductUrlResolvers>());
            CreateMap<Core.Entities.Identity.Address, AddressDto>();
            CreateMap<CustomerBasket, CustomerbasketDto>();
            CreateMap<BasketItem, BasketItemDto>();
            CreateMap<AddressDto, Core.Entities.OrderAggregate.Address>();
        }
    }
}
```



```
using AutoMapper;
using Skinet.Core.Entities;
using Skinet.Core.Entities.OrderAggregate;
using Skinet.Dtos;

namespace Skinet.Helpers
{
    2 references
    public class MappingProfiles: Profile
    {
        0 references
        public MappingProfiles()
        {
            CreateMap<Products, ProductDto>().
              ForMember(d => d.ProductBrand, o => o.MapFrom(s => s.ProductBra
             .ForMember(p => p.ProductType, pt => pt.MapFrom(p => p.ProductTy
             .ForMember(p=>p.PictureUrl,pt=>pt.MapFrom<ProductUrlResolvers>()
            CreateMap<Core.Entities.Identity.Address, AddressDto>();
            CreateMap<CustomerBasket, CustomerbasketDto>();
            CreateMap<BasketItem, BasketItemDto>();
            CreateMap<AddressDto, Core.Entities.OrderAggregate.Address>();
        }
    }
}
```

Data Transfer Object Folder (DTOs)

DTO stands for **data transfer object**. As the name suggests, a DTO is an object made to transfer data. We create the data transfer object class for mapping the incoming request data.

Code of DTOs

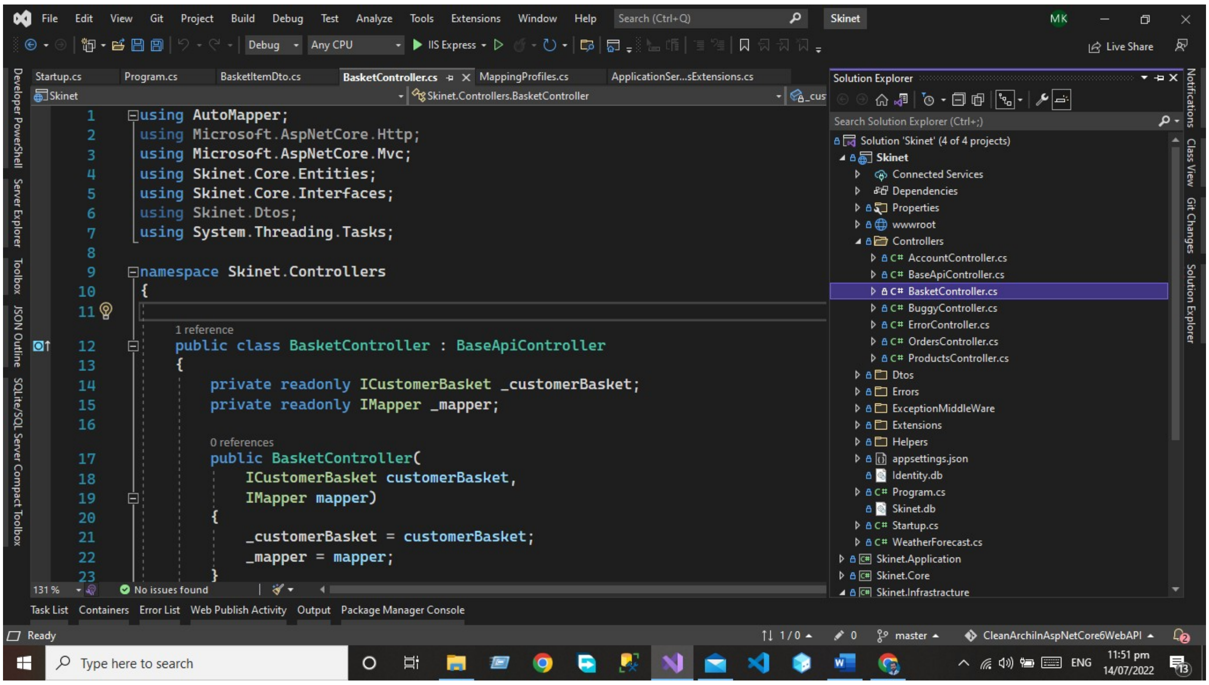
Lets us Create the Basket DTOs for clearing the concept about

```
using System.ComponentModel.DataAnnotations;

namespace Skinet.Dtos
{
    public class BasketItemDto
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public decimal Price { get; set; }
        public int Quantity { get; set; }
        public string PictureUrl { get; set; }
        public string Brand { get; set; }
        public string Type { get; set; }
    }
}
```

Controllers

Controllers are used to handle the HTTP request. Now we need to add the student controller that will interact will our service layer and display the data to the users.

In this chapter, we will focus on Basket controllers because in the whole chapter we talk about Basket.



```
using AutoMapper;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Skinet.Core.Entities;
using Skinet.Core.Interfaces;
using Skinet.Dtos;
using System.Threading.Tasks;

namespace Skinet.Controllers
{
    1 reference
    public class BasketController : BaseApiController
    {
        private readonly ICustomerBasket _customerBasket;
        private readonly IMapper _mapper;

        0 references
        public BasketController(
            ICustomerBasket customerBasket,
            IMapper mapper)
        {
            _customerBasket = customerBasket;
            _mapper = mapper;
        }
```

Code of Basket Controller

```
using AutoMapper;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Skinet.Core.Entities;
using Skinet.Core.Interfaces;
using Skinet.Dtos;
using System.Threading.Tasks;

namespace Skinet.Controllers
{
    public class BasketController : BaseApiController
    {
        private readonly ICustomerBasket _customerBasket;
        private readonly IMapper _mapper;

        public BasketController(
            ICustomerBasket customerBasket,
            IMapper mapper)
        {
            _customerBasket = customerBasket;
            _mapper = mapper;
        }
        [HttpGet(nameof(GetBasketElement))]
        public async Task<ActionResult<CustomerBasket>> GetBasketElement([FromQuery]string Id)
        {
            var basketelements = await _customerBasket.GetBasketAsync(Id);
            return Ok(basketelements??new CustomerBasket(Id));
        }

        [HttpPost(nameof(UpdateProduct))]
        public async Task<ActionResult<CustomerBasket>> UpdateProduct(CustomerBasket product)
        {
            //var customerbasket = _mapper.Map<CustomerbasketDto, CustomerBasket>(product);
            var data =  await _customerBasket.UpdateBasketAsync(product);
            return Ok(data);
        }
        [HttpDelete(nameof(DeleteProduct))]
        public async Task DeleteProduct(string Id)
        {
            await _customerBasket.DeleteBasketAsync(Id);
        }
    }
}
```

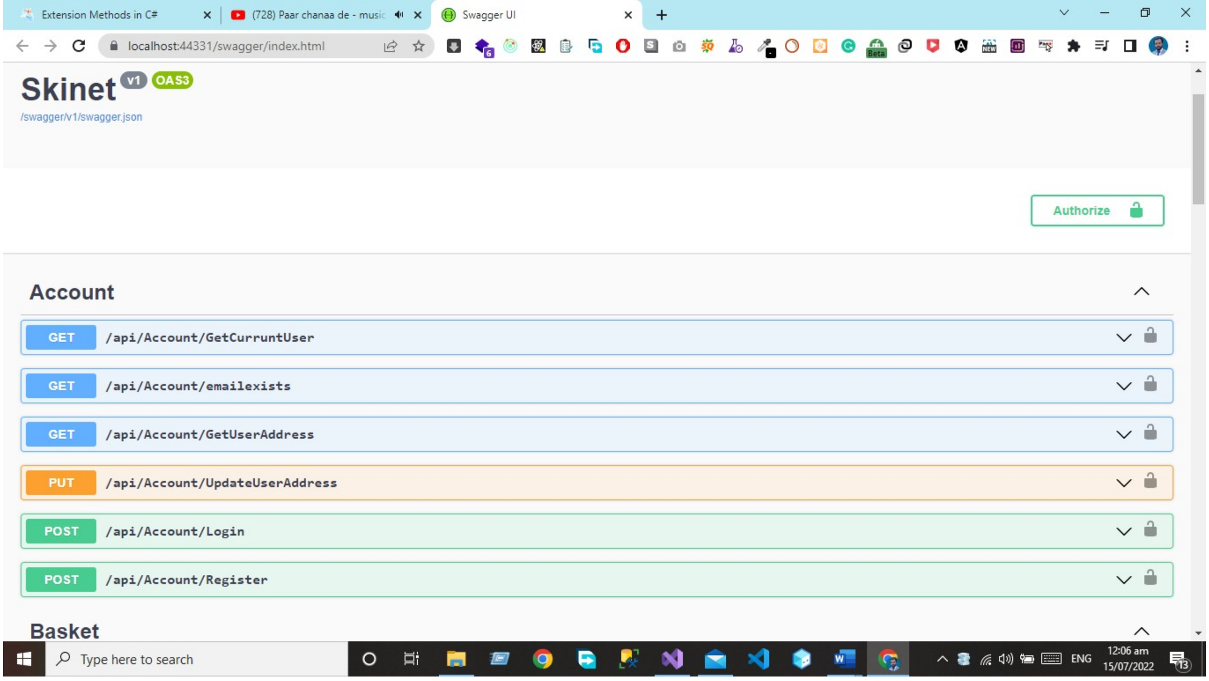
Let’s us Take another example of Product Controller

Code of the Product Controller

```
using AutoMapper;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Skinet.Core.Entities;
using Skinet.Core.Interfaces;
using Skinet.Core.Specifications;
using Skinet.Dtos;
using Skinet.Errors;
using Skinet.Helpers;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace Skinet.Controllers
{

    public class ProductsController : BaseApiController
    {
        private readonly IGenericRepository<Products> _productRepository;
        private readonly IGenericRepository<ProductBrand> _brandRepository;
        private readonly IGenericRepository<ProductType> _productType;
        public IMapper _Mapper;

        public ProductsController(IGenericRepository<Products> productRepository,
            IGenericRepository<ProductBrand> BrandRepository,
            IGenericRepository<ProductType> productType, IMapper mapper)
        {
            _productRepository = productRepository;
            _brandRepository = BrandRepository;
            _productType = productType;
            _Mapper = mapper;
        }
        [HttpGet(nameof(GetProducts))]
        public async Task<ActionResult<Pagination<ProductDto>>> GetProducts(
            [FromQuery]ProductSpecPrams productSpecPrams)
        {
            var spec = new ProductWithSpecificationTypesAndBrand(productSpecPrams);
            var speccount = new ProductWithFilterCountSpecification(productSpecPrams);
            var total = await _productRepository.CountAsync(spec);
            var products = await _productRepository.ListAsync(spec);
            var data = _Mapper.Map<IReadOnlyList<Products>, IReadOnlyList<ProductDto>>(products);
            return Ok(new Pagination<ProductDto>(productSpecPrams.pageIndex, productSpecPrams.pageSize, total,
data));
        }

        [HttpGet(nameof(GetProductById))]
        [ProducesResponseType(StatusCodes.Status200OK)]
        [ProducesResponseType(typeof(APIResponce),StatusCodes.Status404NotFound)]
        public async Task<ProductDto> GetProductById([FromQuery] int Id)
        {
            var spec = new ProductWithSpecificationTypesAndBrand(Id);
            var products = await _productRepository.GetByIdAsync(Id);
            return _Mapper.Map<ProductDto>(products);
        }

        [HttpGet(nameof(GetProductBrand))]
        public async Task<ActionResult<List<ProductBrand>>> GetProductBrand()
        {
            var obj = await _brandRepository.GetAllAsync();
            return Ok(obj);
        }

        [HttpGet(nameof(GetProductTypes))]
        public async Task<ActionResult<List<ProductType>>> GetProductTypes()
        {
            var obj = await _productRepository.GetAllAsync();
            return Ok(obj);
        }
    }
}
```

Output

Now we will run the project and will see the output using the swagger.



Skinet v1 OAS3

/swagger/v1/swagger.json

Authorize

Account

GET /api/Account/GetCurruntUser

GET /api/Account/emailexists

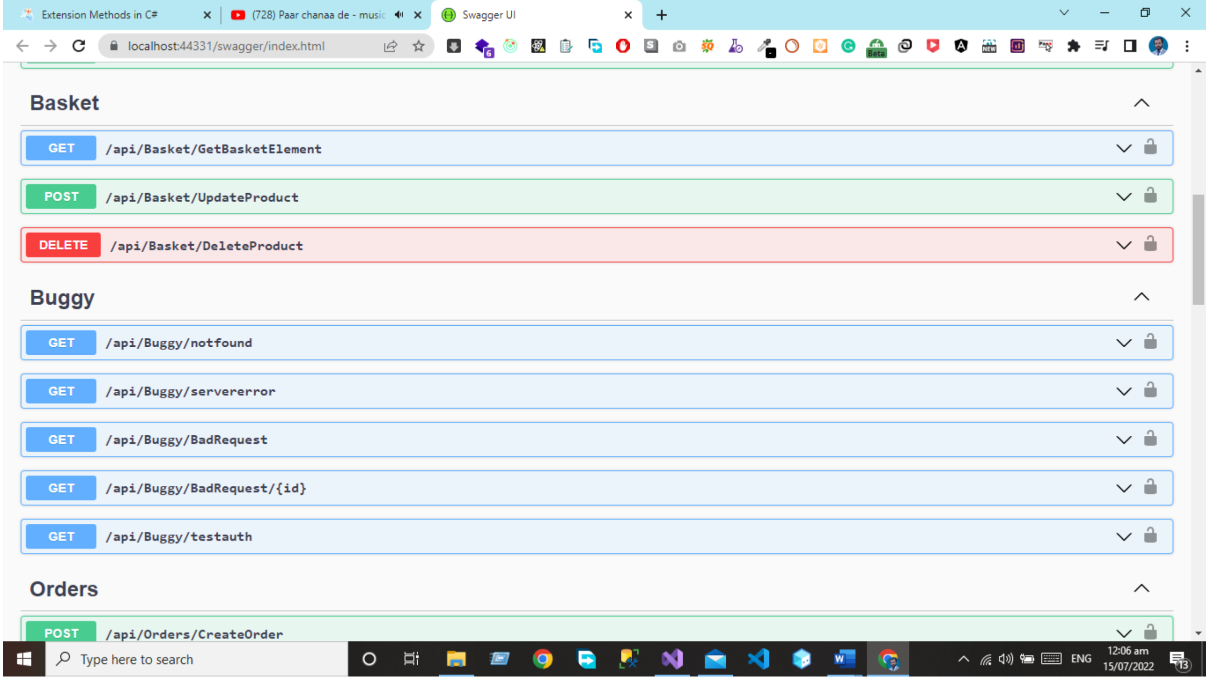
GET /api/Account/GetUserAddress

PUT /api/Account/UpdateUserAddress

POST /api/Account/Login

POST /api/Account/Register

Basket



Basket

GET /api/Basket/GetBasketElement

POST /api/Basket/UpdateProduct

DELETE /api/Basket/DeleteProduct

Buggy

GET /api/Buggy/notfound

GET /api/Buggy/servererror

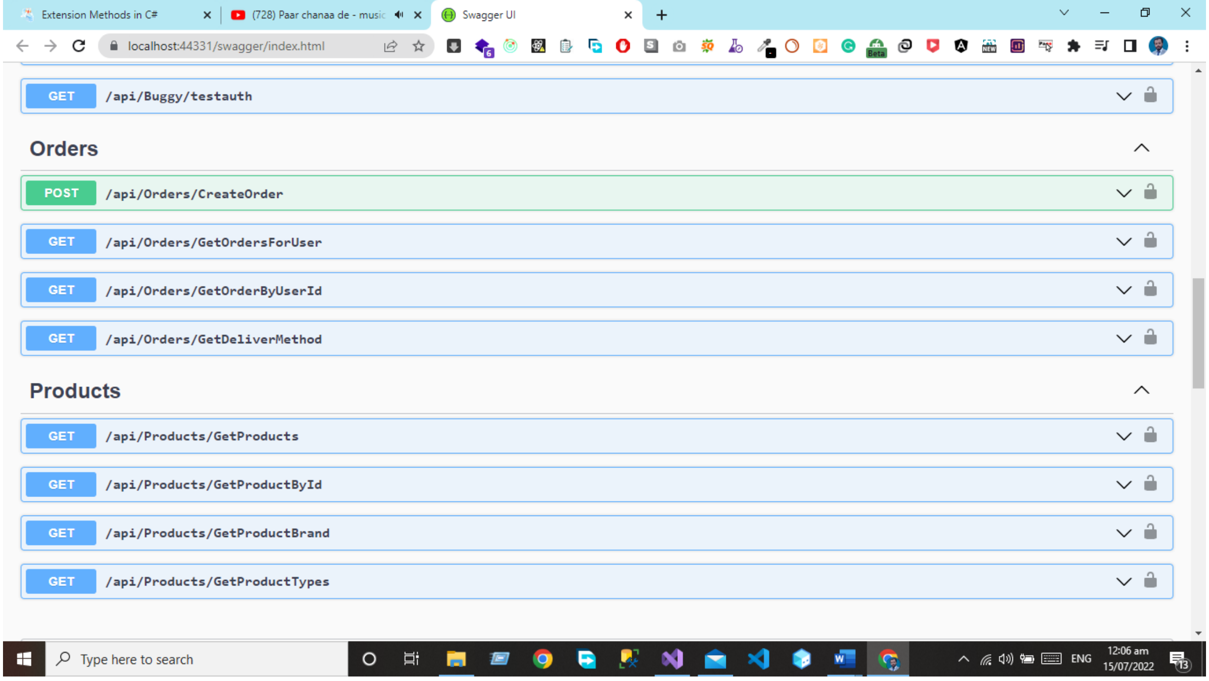
GET /api/Buggy/BadRequest

GET /api/Buggy/BadRequest/{id}

GET /api/Buggy/testauth

Orders

POST /api/Orders/CreateOrder



GET /api/Buggy/testauth

Orders

POST /api/Orders/CreateOrder

GET /api/Orders/GetOrdersForUser

GET /api/Orders/GetOrderByUserId

GET /api/Orders/GetDeliverMethod

Products

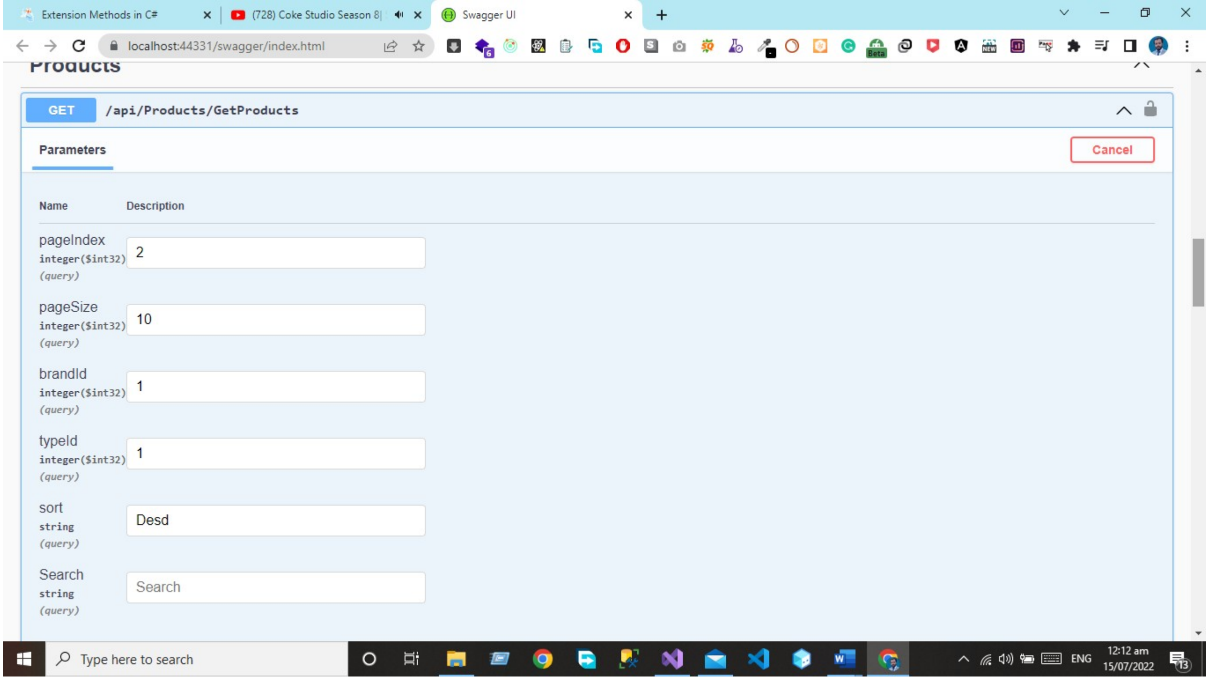
GET /api/Products/GetProducts

GET /api/Products/GetProductById

GET /api/Products/GetProductBrand

GET /api/Products/GetProductTypes

Let’s Get the Product using the Product Controller



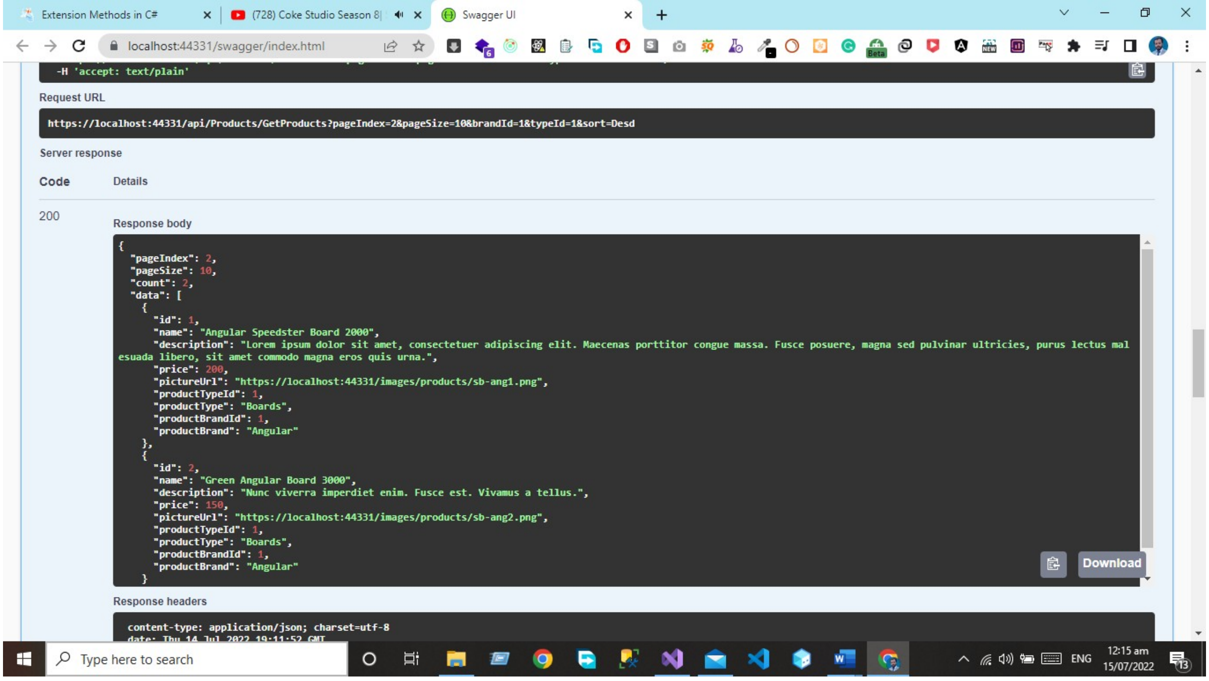
GET /api/Products/GetProducts

Parameters

Cancel

Name	Description
pageIndex integer($int32) (query)	2
pageSize integer($int32) (query)	10
brandId integer($int32) (query)	1
typeId integer($int32) (query)	1
sort string (query)	Desd
Search string (query)	Search

These are the parameter that is very helpful for getting the desired data.



```
-H 'accept: text/plain'
```

Request URL

```
https://localhost:44331/api/Products/GetProducts?pageIndex=2&pageSize=10&brandId=1&typeId=1&sort=Desd
```

Server response

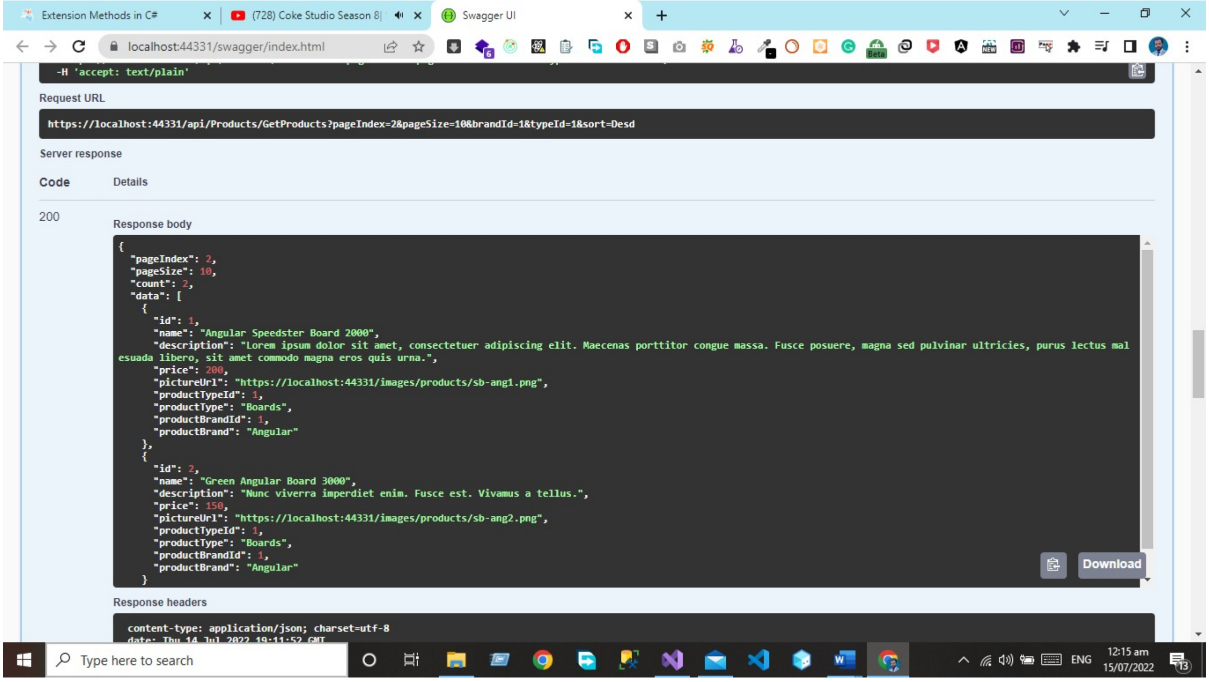
Code	Details
200	Response body

```
{
  "pageIndex": 2,
  "pageSize": 10,
  "count": 2,
  "data": [
    {
      "id": 1,
      "name": "Angular Speedster Board 2000",
      "description": "Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Maecenas porttitor congue massa. Fusce posuere, magna sed pulvinar ultricies, purus lectus mal
esuada libero, sit amet commodo magna eros quis urna.",
      "price": 200,
      "pictureUrl": "https://localhost:44331/images/products/sb-ang1.png",
      "productTypeId": 1,
      "productType": "Boards",
      "productBrandId": 1,
      "productBrand": "Angular"
    },
    {
      "id": 2,
      "name": "Green Angular Board 3000",
      "description": "Nunc viverra imperdiet enim. Fusce est. Vivamus a tellus.",
      "price": 150,
      "pictureUrl": "https://localhost:44331/images/products/sb-ang2.png",
      "productTypeId": 1,
      "productType": "Boards",
      "productBrandId": 1,
      "productBrand": "Angular"
    }
```

Response headers

```
content-type: application/json; charset=utf-8
```

Now we can see when we hit the get products endpoint, we can see the data of students from the database in the form of the JSON object.



```
-H 'accept: text/plain'
```

Request URL

```
https://localhost:44331/api/Products/GetProducts?pageIndex=2&pageSize=10&brandId=1&typeId=1&sort=Desd
```

Server response

Code Details

200 Response body

```
{
  "pageIndex": 2,
  "pageSize": 10,
  "count": 2,
  "data": [
    {
      "id": 1,
      "name": "Angular Speedster Board 2000",
      "description": "Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Maecenas porttitor congue massa. Fusce posuere, magna sed pulvinar ultricies, purus lectus mal
esuada libero, sit amet commodo magna eros quis urna.",
      "price": 200,
      "pictureUrl": "https://localhost:44331/images/products/sb-ang1.png",
      "productTypeId": 1,
      "productType": "Boards",
      "productBrandId": 1,
      "productBrand": "Angular"
    },
    {
      "id": 2,
      "name": "Green Angular Board 3000",
      "description": "Nunc viverra imperdiet enim. Fusce est. Vivamus a tellus.",
      "price": 150,
      "pictureUrl": "https://localhost:44331/images/products/sb-ang2.png",
      "productTypeId": 1,
      "productType": "Boards",
      "productBrandId": 1,
      "productBrand": "Angular"
    }
```

Response headers

```
content-type: application/json; charset=utf-8
```

Conclusion

In this chapter, we have implemented the clean architecture using the Entity Framework and Code First approach. We have now the knowledge of how the layer communicates with each other's in clean architecture and how we can write the Generic code for the Interface repository and services. Now we can develop our project using clean architecture for API Development OR MVC Core Based projects.

Complete Project Code

You can download the complete project code from the following GitHub repository. Follow the link below and down the complete project code and practice the code by yourself and learn how we can implement the clean architecture in asp.net code WebAPI.

The complete Code of the project will be available by following the given below link.

CLICK THE LINK FOR THE GIT HUB REPOSITORY

Chapter 5- Application Deployment on Server.

- ✓ Introduction
- ✓ Deployment Procedure
- ✓ Server Configuration
- ✓ Deployment Steps
- ✓ Conclusion

Introduction

In this chapter, we will learn about how to deploy the asp.net core 5 websites on a server using visual studio 2019.

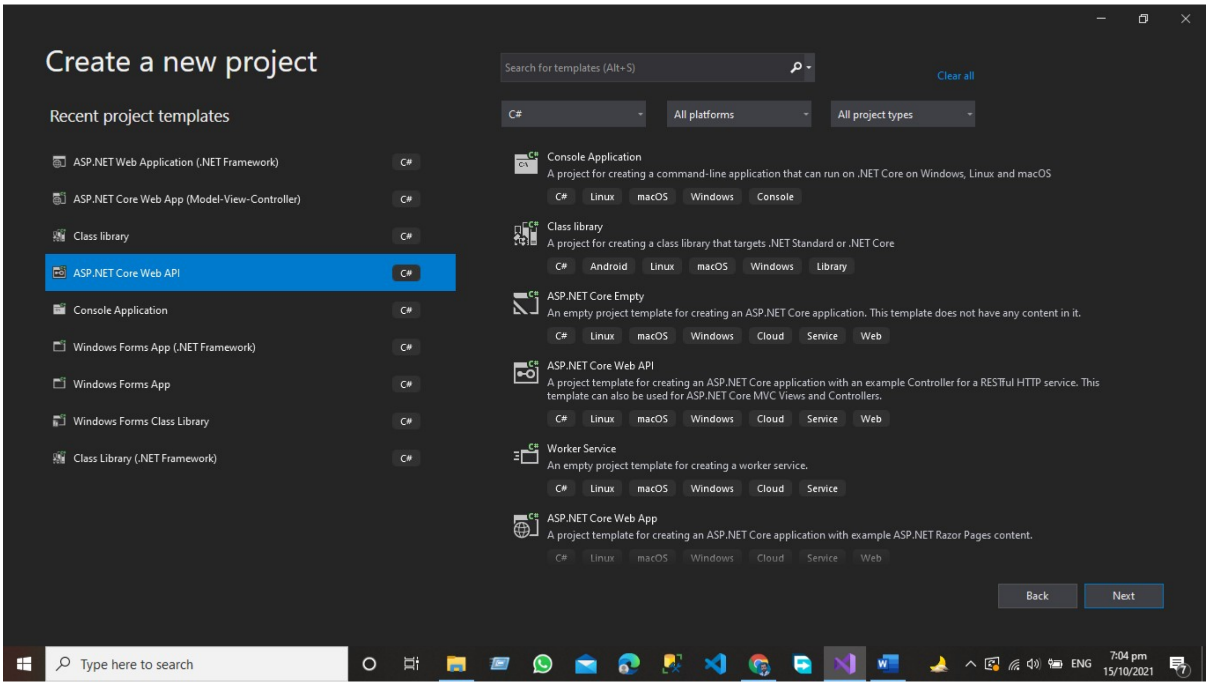
Step 1 Account Setup on Server

The first step for deployment of asp.net core web API is to create an account on the hosting provider website that supports the asp.net web application framework and then follow the steps in the given chapter.

- Create a website in the hosting panel.
- Then Create the SQL Server Website using the Hosting Panel

Step 2 Create the project asp.net Core 5 Web API using VS-2019

Create the project using visual studio and select the asp.net core version .Net 5.0 Current.



Create a new project

Recent project templates

- ASP.NET Web Application (.NET Framework) — C#
- ASP.NET Core Web App (Model-View-Controller) — C#
- Class library — C#
- ASP.NET Core Web API — C#
- Console Application — C#
- Windows Forms App (.NET Framework) — C#
- Windows Forms App — C#
- Windows Forms Class Library — C#
- Class Library (.NET Framework) — C#

Console Application
A project for creating a command-line application that can run on .NET Core on Windows, Linux and macOS
C# · Linux · macOS · Windows · Console

Class library
A project for creating a class library that targets .NET Standard or .NET Core
C# · Android · Linux · macOS · Windows · Library

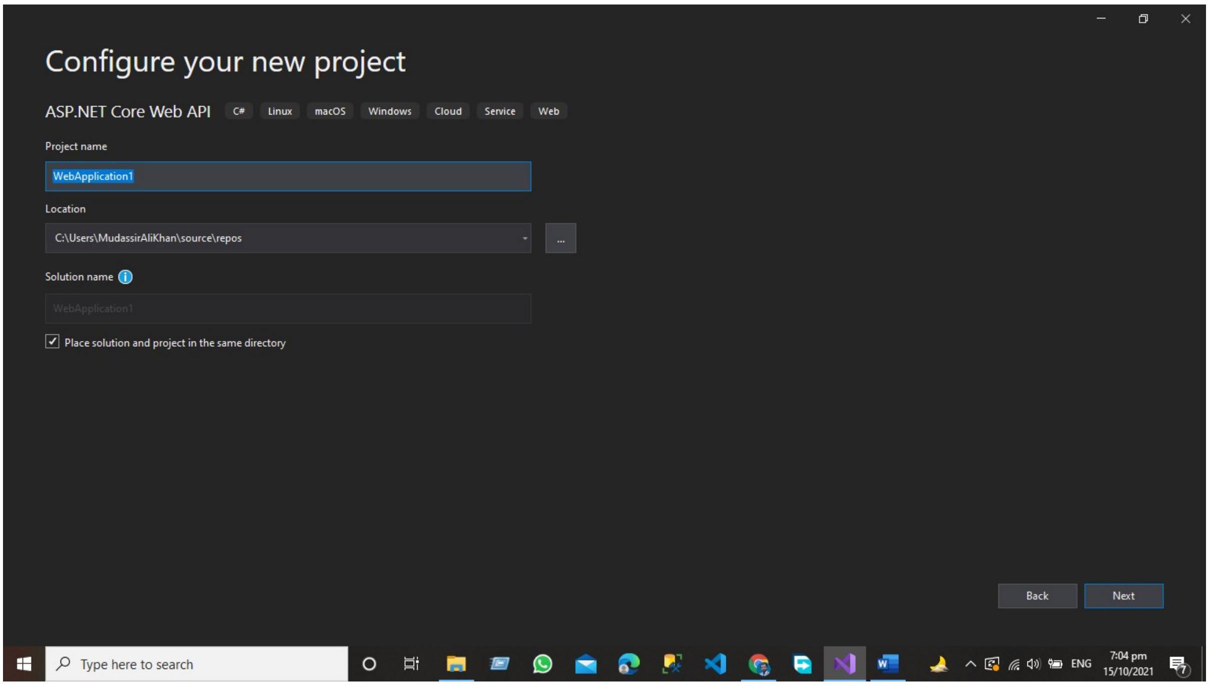
ASP.NET Core Empty
An empty project template for creating an ASP.NET Core application. This template does not have any content in it.
C# · Linux · macOS · Windows · Cloud · Service · Web

ASP.NET Core Web API
A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.
C# · Linux · macOS · Windows · Cloud · Service · Web

Worker Service
An empty project template for creating a worker service.
C# · Linux · macOS · Windows · Cloud · Service

ASP.NET Core Web App
A project template for creating an ASP.NET Core application with example ASP.NET Razor Pages content.
C# · Linux · macOS · Windows · Cloud · Service · Web

Give the Name of your project that you want to develop My Project is ONP (Online Prescription Application)



Configure your new project

ASP.NET Core Web API C# Linux macOS Windows Cloud Service Web

Project name

WebApplication1

Location

C:\Users\MudassirAliKhan\source\repos

Solution name

WebApplication1

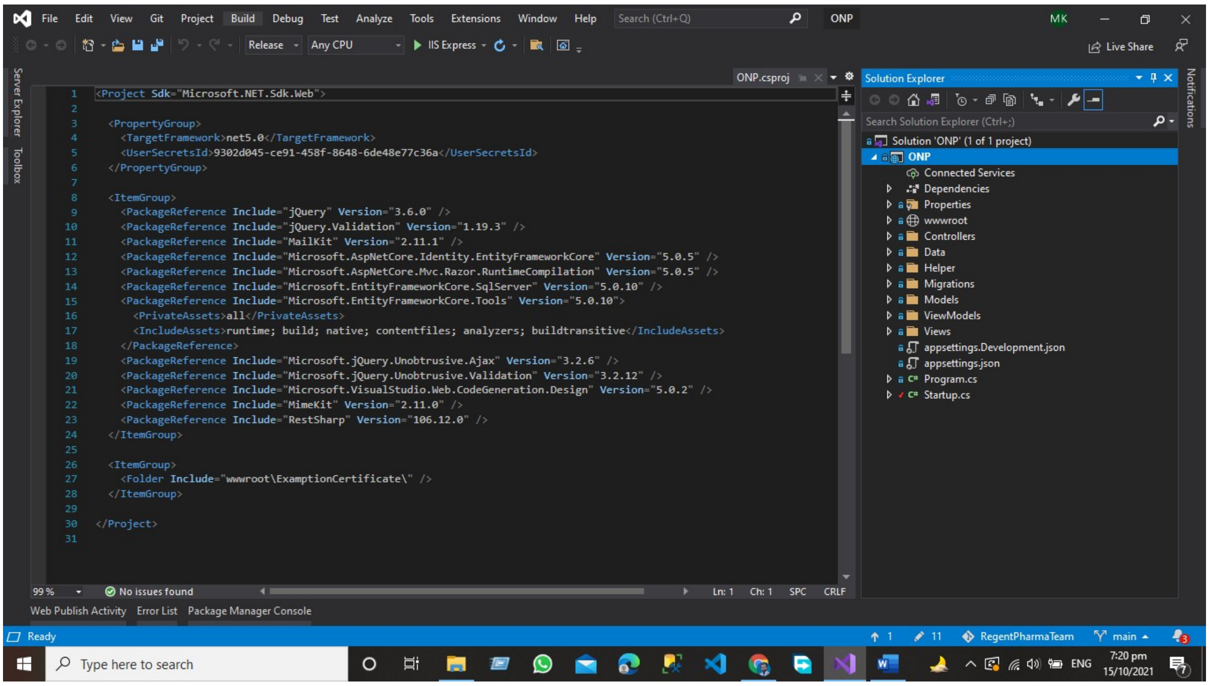
☑ Place solution and project in the same directory

Step 3 Set the Live Server Database Connection string in your application appsetting.json file

```
{

"DefaultConnection": "Data Source=SQL5108.site4now.net;Initial Catalog=YOUR ONLINE DB NAME;User Id=YOUR ONLINE DB USERNAME;Password=YOUR ONLINE DB PASSWORD"

 },
 "Logging": {
   "LogLevel": {
     "Default": "Information",
     "Microsoft": "Warning",
     "Microsoft.Hosting.Lifetime": "Information"
   }
 },
 "AllowedHosts": "*"
}
```

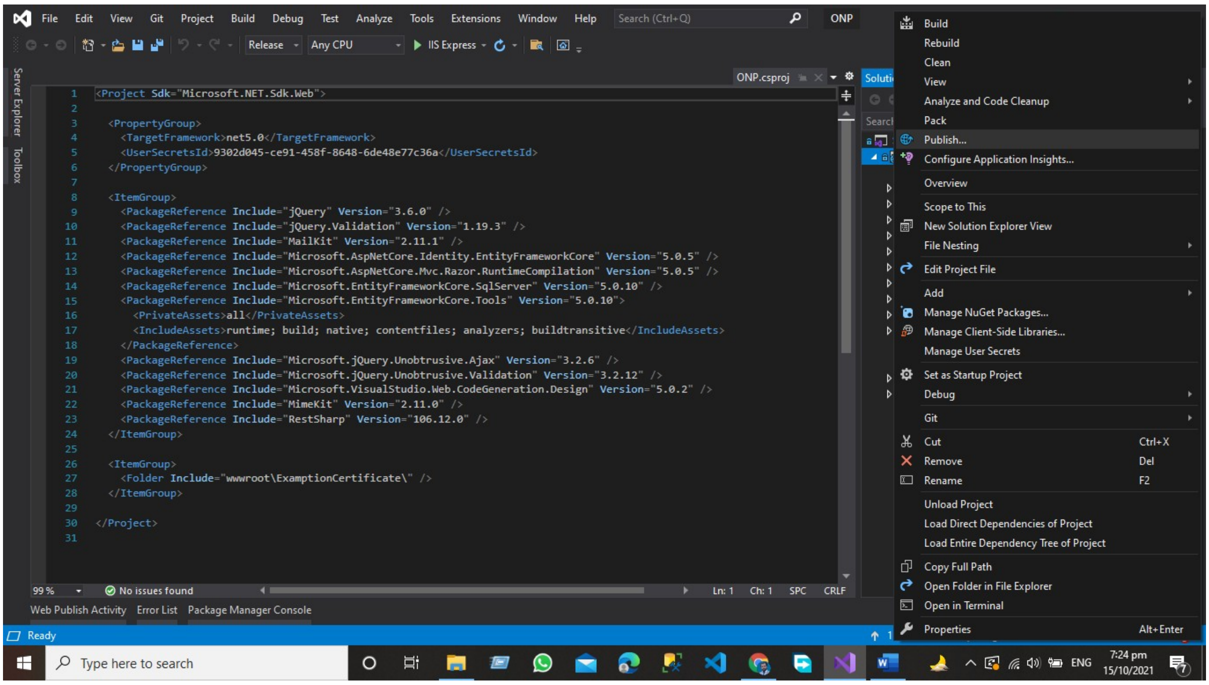
Step 4 Publish Your Project

After Creating your project and Final Testing from QA Section now it's time to deploy our website Now go to solution explorer in visual studio 2019.



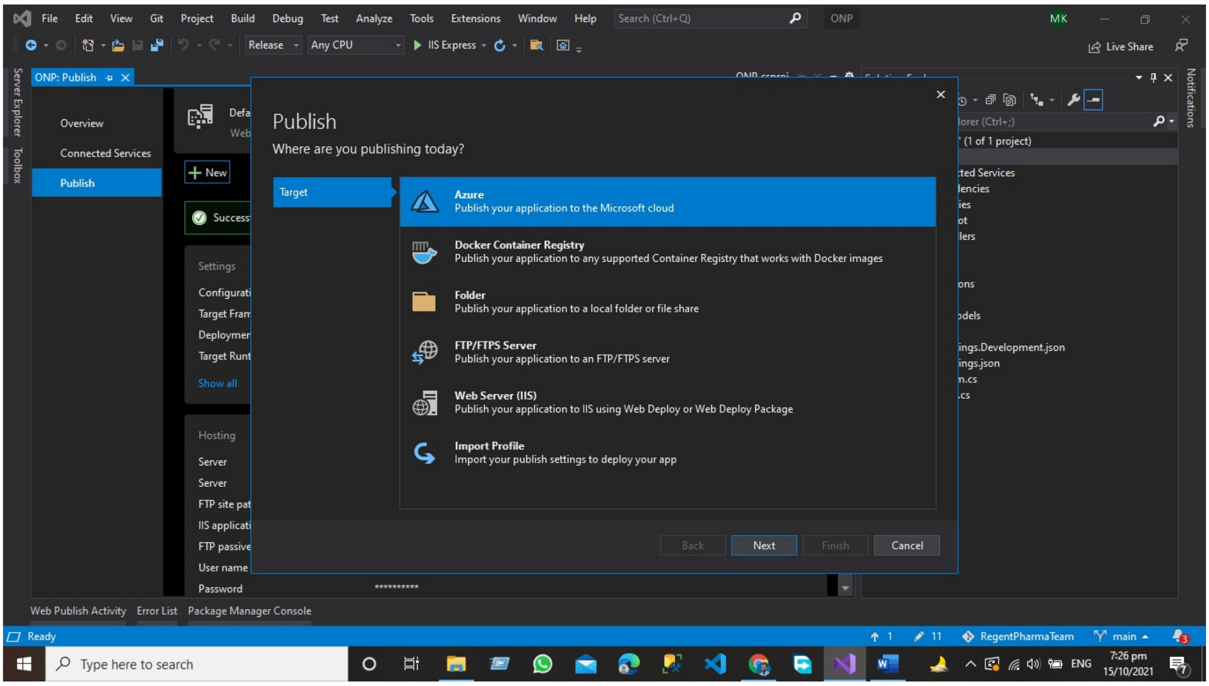
```xml
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
    <UserSecretsId>9302d045-ce91-458f-8648-6de48e77c36a</UserSecretsId>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="jQuery" Version="3.6.0" />
    <PackageReference Include="jQuery.Validation" Version="1.19.3" />
    <PackageReference Include="MailKit" Version="2.11.1" />
    <PackageReference Include="Microsoft.AspNetCore.Identity.EntityFrameworkCore" Version="5.0.5" />
    <PackageReference Include="Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation" Version="5.0.5" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="5.0.10" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="5.0.10">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
    </PackageReference>
    <PackageReference Include="Microsoft.jQuery.Unobtrusive.Ajax" Version="3.2.6" />
    <PackageReference Include="Microsoft.jQuery.Unobtrusive.Validation" Version="3.2.12" />
    <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="5.0.2" />
    <PackageReference Include="MimeKit" Version="2.11.0" />
    <PackageReference Include="RestSharp" Version="106.12.0" />
  </ItemGroup>

  <ItemGroup>
    <Folder Include="wwwroot\ExamptionCertificate\" />
  </ItemGroup>

</Project>
```

Now Right Click on your Project and select the option to publish.

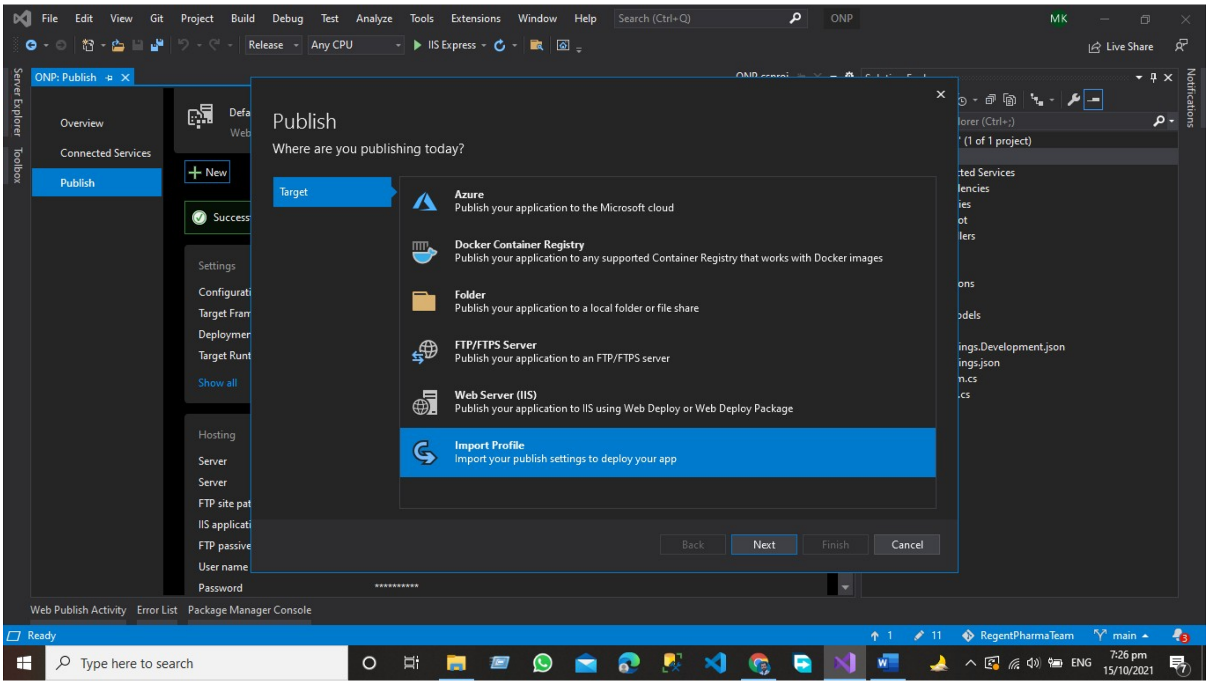


```xml
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
    <UserSecretsId>9302d045-ce91-458f-8648-6de48e77c36a</UserSecretsId>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="jQuery" Version="3.6.0" />
    <PackageReference Include="jQuery.Validation" Version="1.19.3" />
    <PackageReference Include="MailKit" Version="2.11.1" />
    <PackageReference Include="Microsoft.AspNetCore.Identity.EntityFrameworkCore" Version="5.0.5" />
    <PackageReference Include="Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation" Version="5.0.5" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="5.0.10" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="5.0.10">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
    </PackageReference>
    <PackageReference Include="Microsoft.jQuery.Unobtrusive.Ajax" Version="3.2.6" />
    <PackageReference Include="Microsoft.jQuery.Unobtrusive.Validation" Version="3.2.12" />
    <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="5.0.2" />
    <PackageReference Include="MimeKit" Version="2.11.0" />
    <PackageReference Include="RestSharp" Version="106.12.0" />
  </ItemGroup>

  <ItemGroup>
    <Folder Include="wwwroot\ExamptionCertificate\" />
  </ItemGroup>

</Project>
```

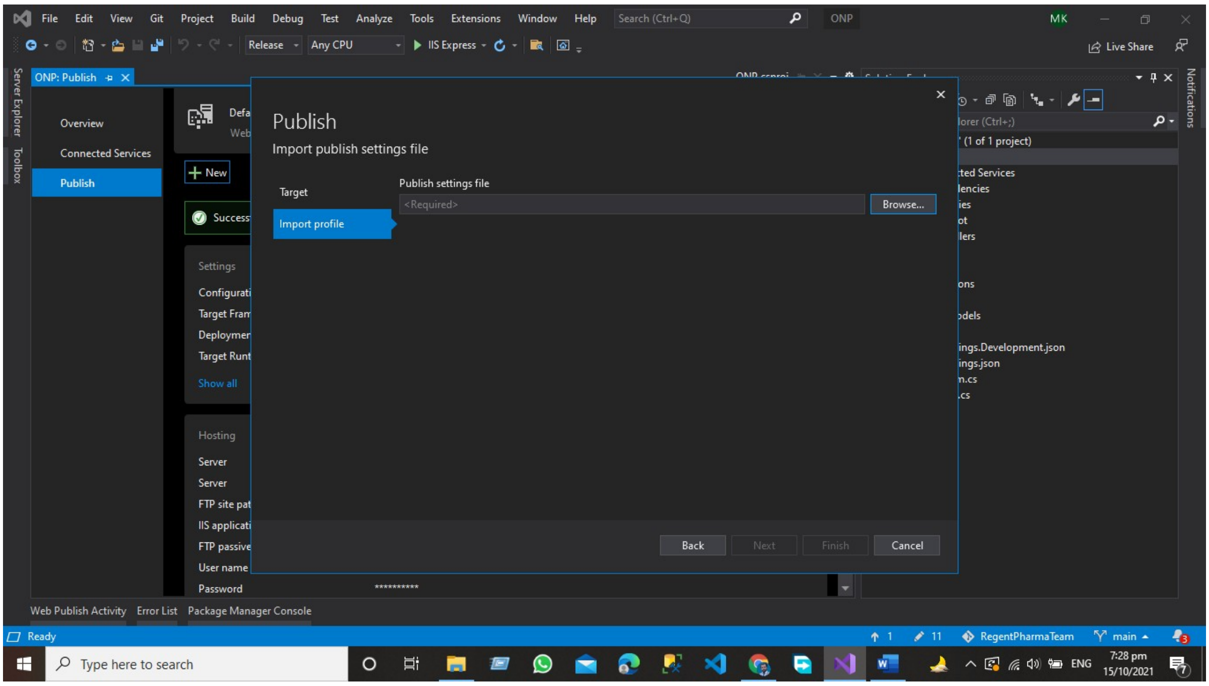
Click the Publish option a new window will appear now select the project publication option here you have 5 to 6 options given below.


Publish
Where are you publishing today?
Target
Azure
Publish your application to the Microsoft cloud
Docker Container Registry
Publish your application to any supported Container Registry that works with Docker images
Folder
Publish your application to a local folder or file share
FTP/FTPS Server
Publish your application to an FTP/FTPS server
Web Server (IIS)
Publish your application to IIS using Web Deploy or Web Deploy Package
Import Profile
Import your publish settings to deploy your app
Back
Next
Finish
Cancel

Now select the option Import to publish profile.


Publish
Where are you publishing today?
Target
Azure
Publish your application to the Microsoft cloud
Docker Container Registry
Publish your application to any supported Container Registry that works with Docker images
Folder
Publish your application to a local folder or file share
FTP/FTPS Server
Publish your application to an FTP/FTPS server
Web Server (IIS)
Publish your application to IIS using Web Deploy or Web Deploy Package
Import Profile
Import your publish settings to deploy your app
Back
Next
Finish
Cancel

After Selecting the Option Import Profile click next.



Publish

Import publish settings file

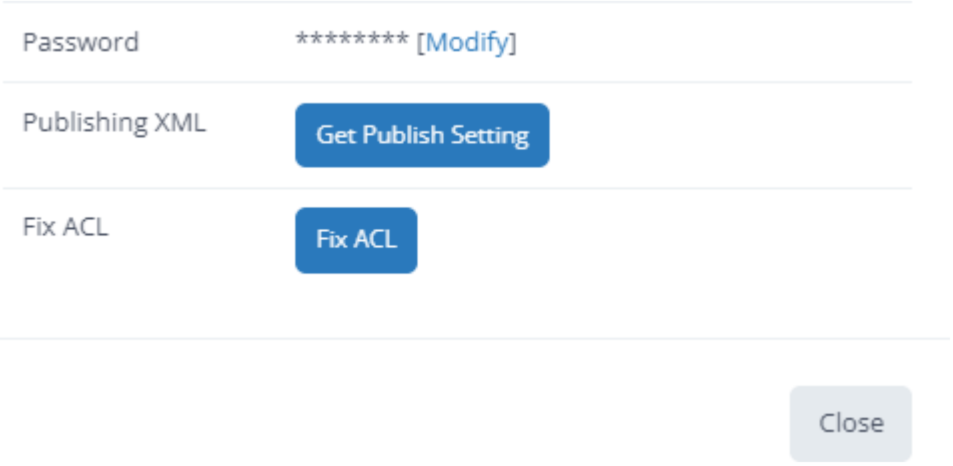
Target

Import profile

Publish settings file

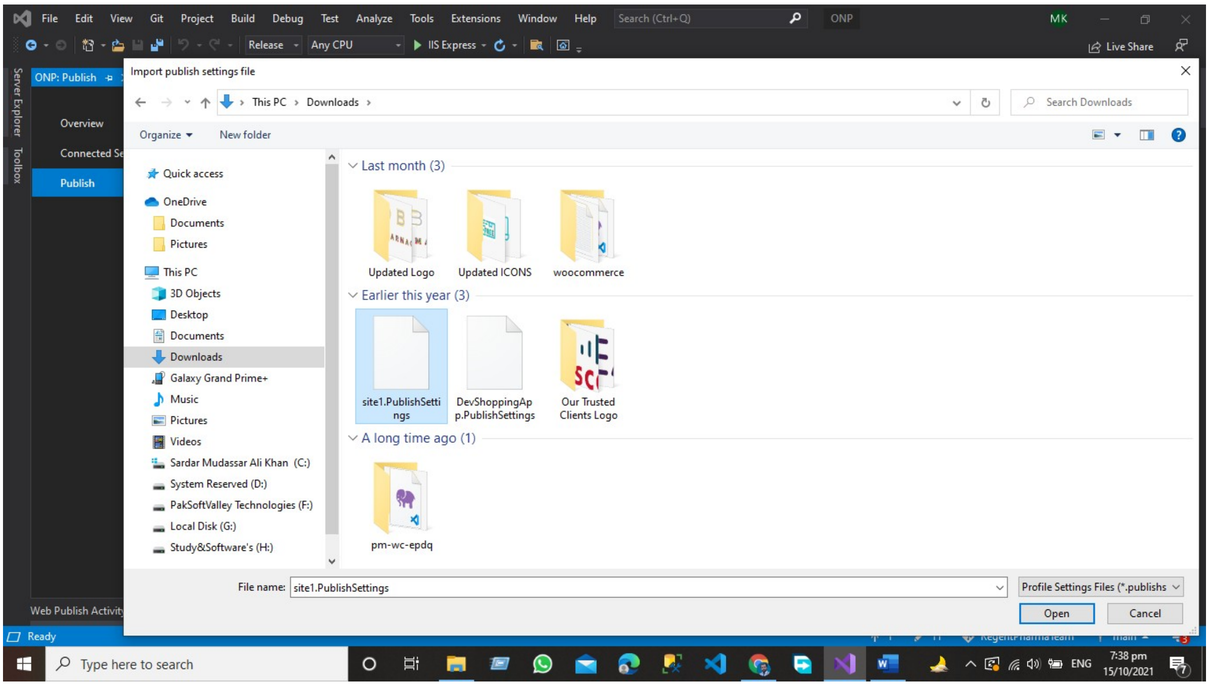
<Required>

But here question will raise in your mind how to get the publish profile just login to the hosting proving server that supports Asp.net core website applications and then go to the deployment option and get the publish profile file and download it in your local system.



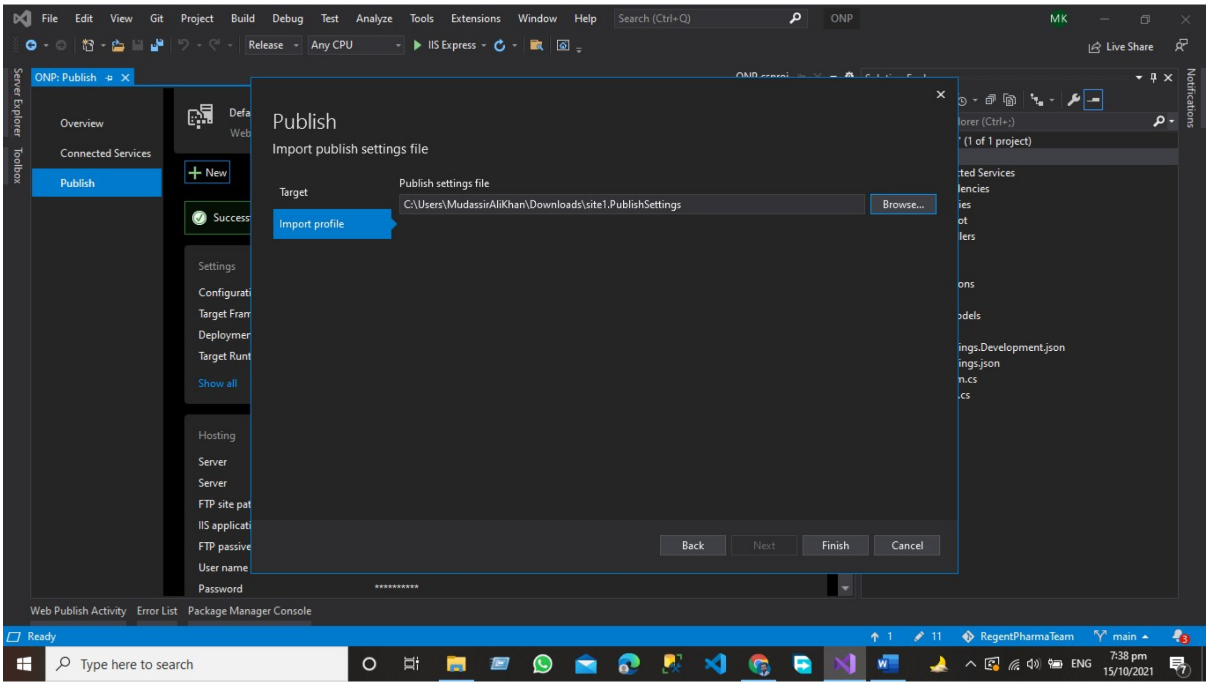
Password	******** [Modify]
Publishing XML	Get Publish Setting
Fix ACL	Fix ACL

You all get this type of option in your control panel just get the publish profile from and down that file in your system.



File name: site1.PublishSettings

Then upload the file in the visual studio using the browse option from the downloaded location.

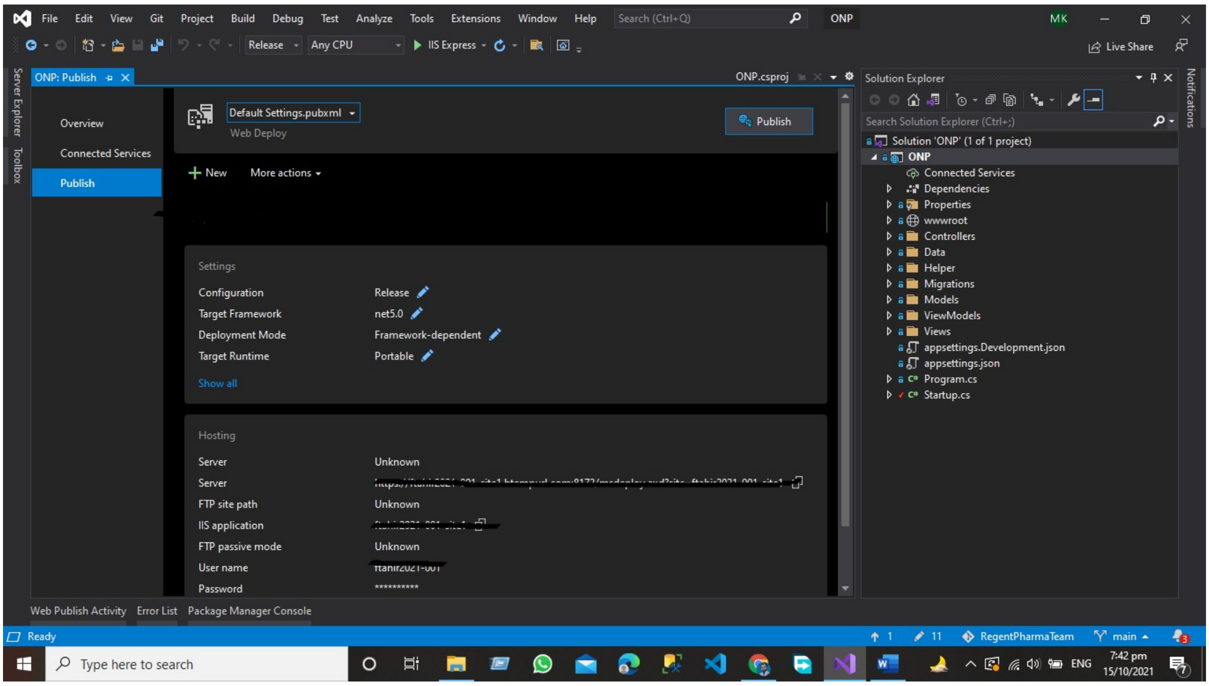


Publish

Import publish settings file

Publish settings file: C:\Users\MudassirAliKhan\Downloads\site1.PublishSettings

After Uploading the publish profile setting file in the visual studio just click on the Finish option and you will get given the below window.



[redacted]

Settings

Configuration Release

Target Framework net5.0

Deployment Mode Framework-dependent

Target Runtime Portable

Show all

Hosting

Server Unknown

Server [redacted]

FTP site path Unknown

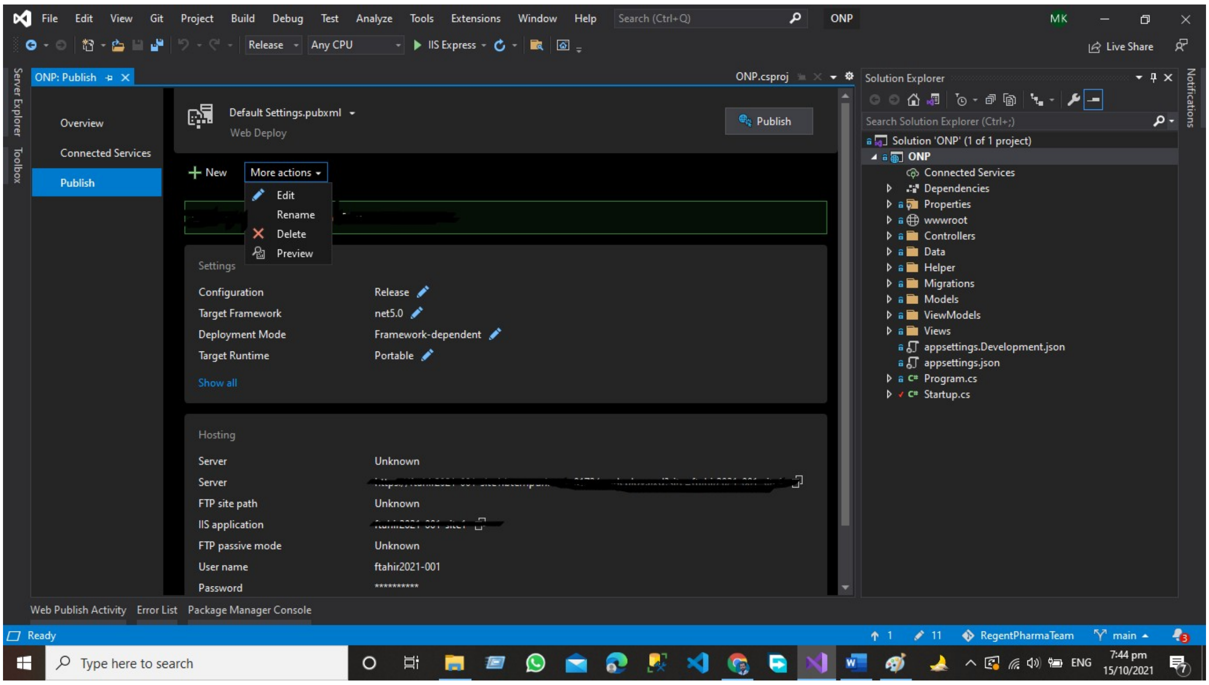
IIS application [redacted]

FTP passive mode Unknown

User name [redacted]

Password **********

Now Click on More Actions.



[redacted]

Settings

Configuration Release

Target Framework net5.0

Deployment Mode Framework-dependent

Target Runtime Portable

Show all

Hosting

Server Unknown

Server [redacted]

FTP site path Unknown

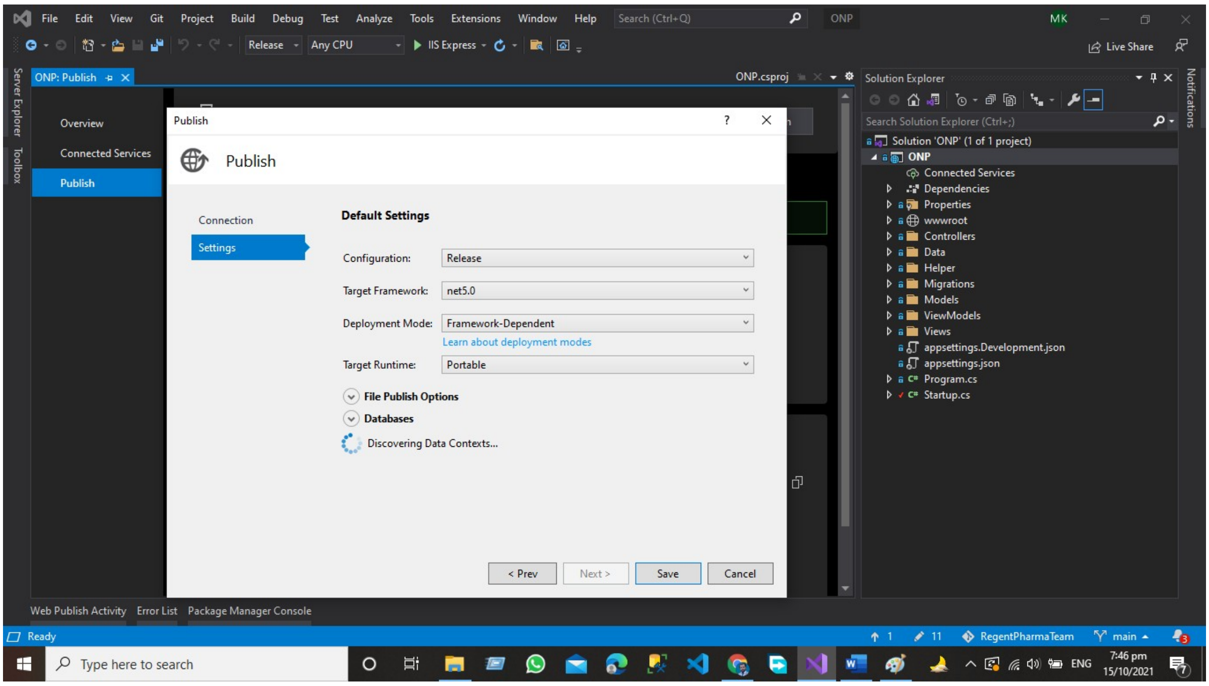
IIS application [redacted]

FTP passive mode Unknown

User name ftahir2021-001

Password **********

Click the Edit Option.



Publish

Connection

Settings

Default Settings

Configuration: Release

Target Framework: net5.0

Deployment Mode: Framework-Dependent

Learn about deployment modes

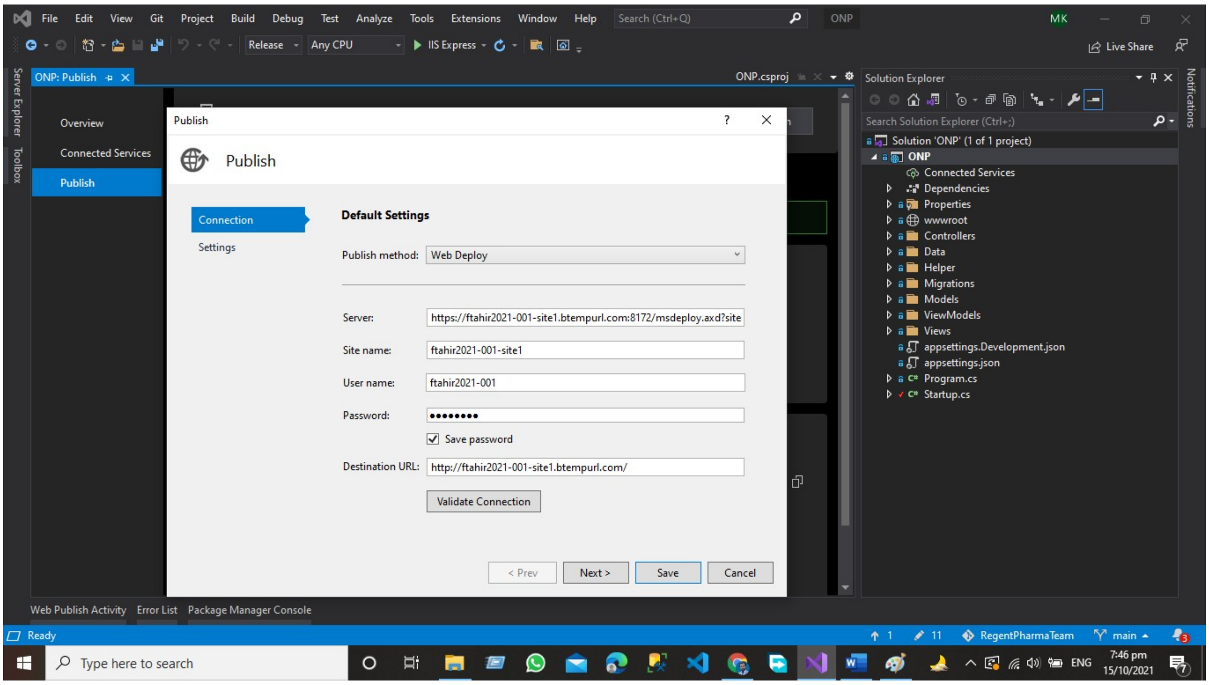
Target Runtime: Portable

File Publish Options

Databases

Discovering Data Contexts...

Click On Connection.



Publish

Connection

Settings

Default Settings

Publish method: Web Deploy

Server: https://ftahir2021-001-site1.btempurl.com:8172/msdeploy.axd?site

Site name: ftahir2021-001-site1

User name: ftahir2021-001

Password: ••••••••

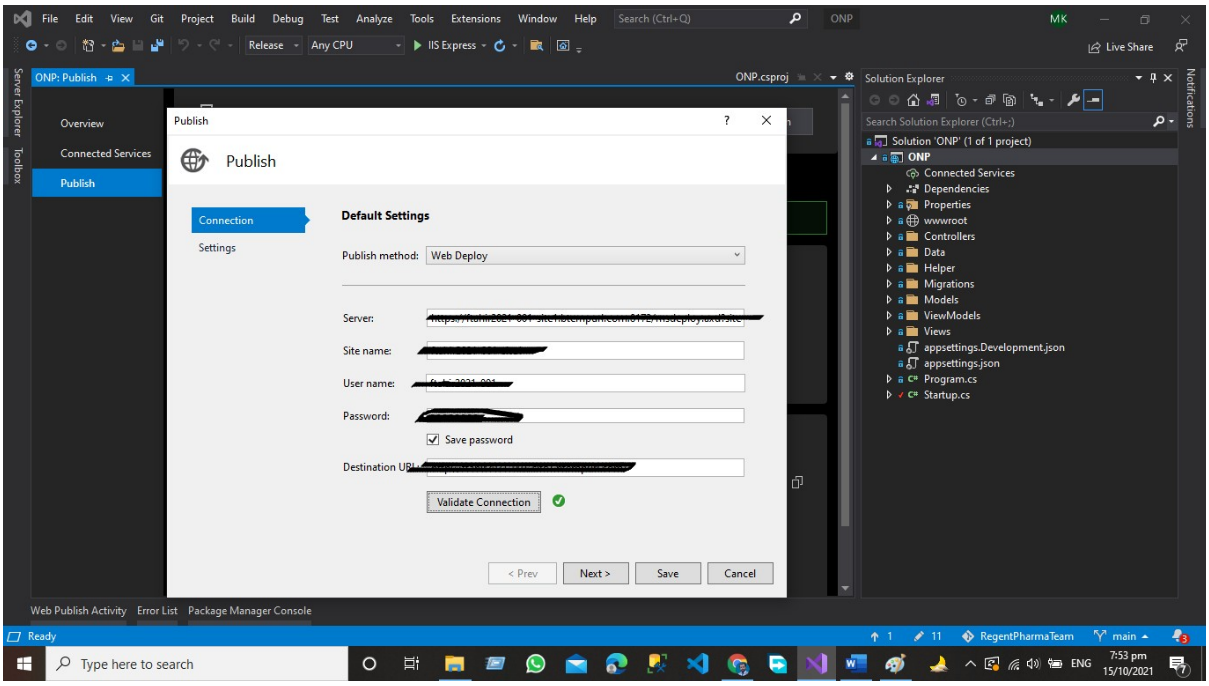
☑ Save password

Destination URL: http://ftahir2021-001-site1.btempurl.com/

All the information will automatically upload fetched from the uploaded publish profile and this information will automatically save in the visual studio for future deployments.

Step 5 Validate the Connection

In this step, visual studio validates the connection with the server and then you can publish your application on the server.



Publish

Connection

Settings

Default Settings

Publish method: Web Deploy

Server: [redacted]

Site name: [redacted]

User name: [redacted]

Password: [redacted]

☑ Save password

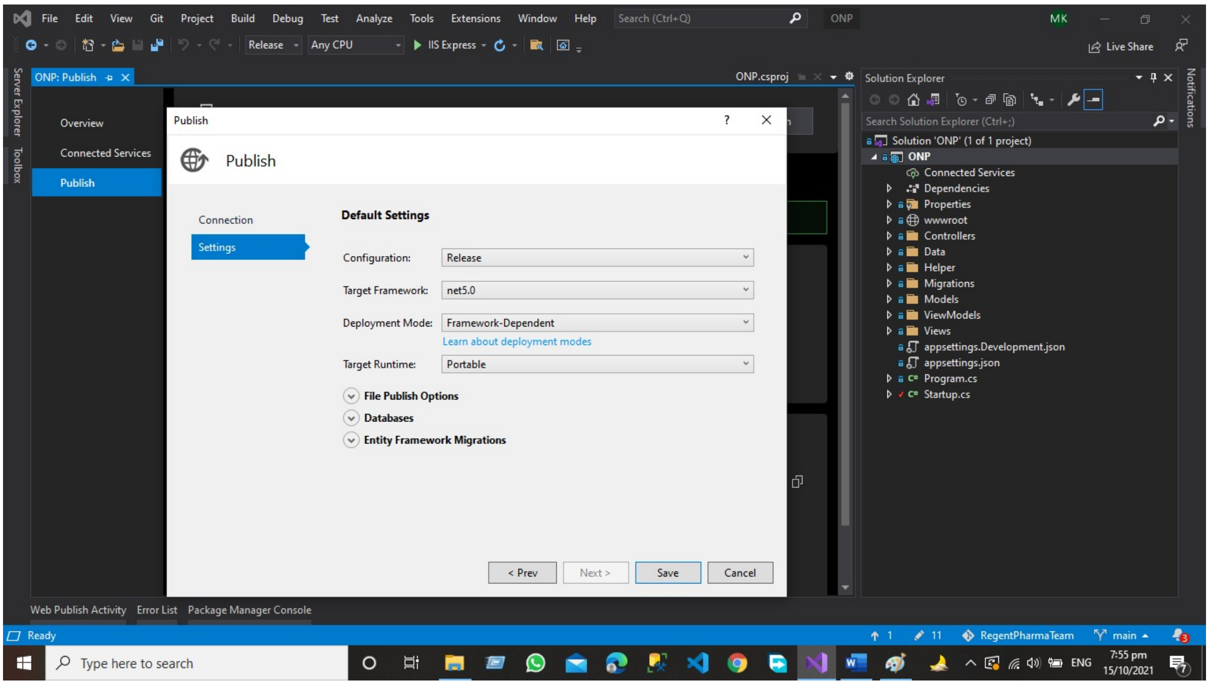
Destination URL: [redacted]

Validate Connection ✓

< Prev | Next > | Save | Cancel

Now you can see that our connection has been validated from visual studio with the server now we are ready to publish the application.

Now Click on Next



Publish

Connection

Settings

Default Settings

Configuration: Release

Target Framework: net5.0

Deployment Mode: Framework-Dependent

Learn about deployment modes

Target Runtime: Portable

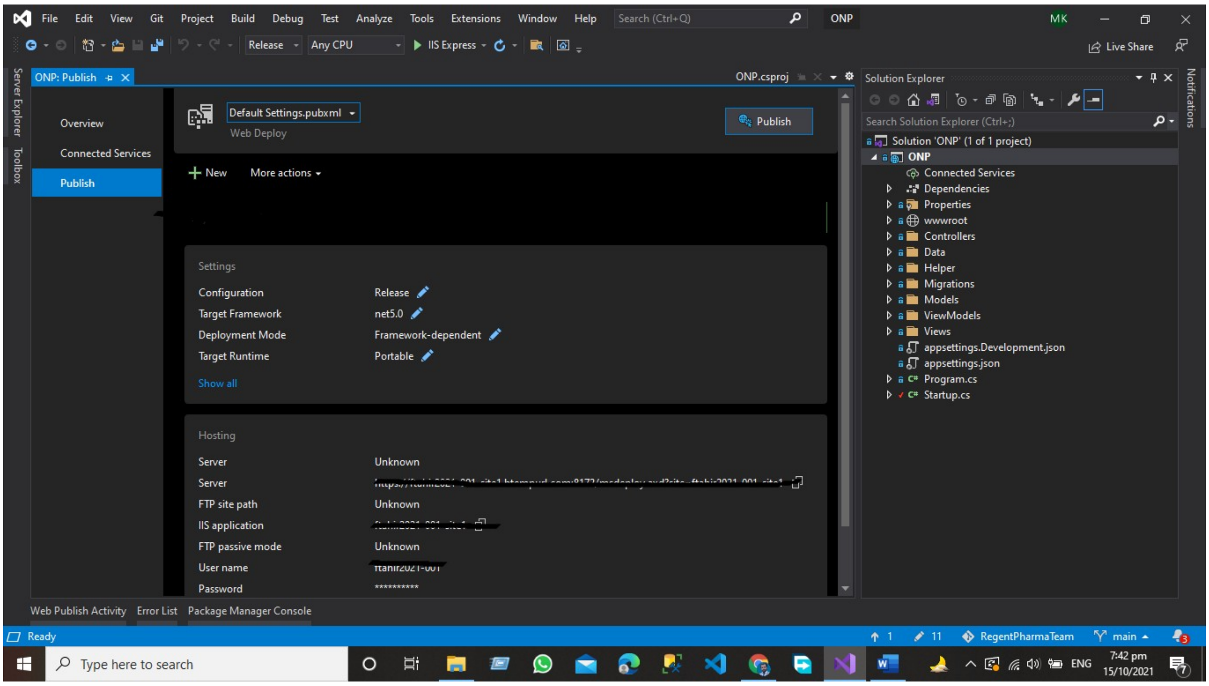
File Publish Options

Databases

Entity Framework Migrations

< Prev | Next > | Save | Cancel

Click Save the Setting



Default Settings.pubxml

Web Deploy

Settings

Configuration Release

Target Framework net5.0

Deployment Mode Framework-dependent

Target Runtime Portable

Hosting

Server Unknown

Server [redacted]

FTP site path Unknown

IIS application [redacted]

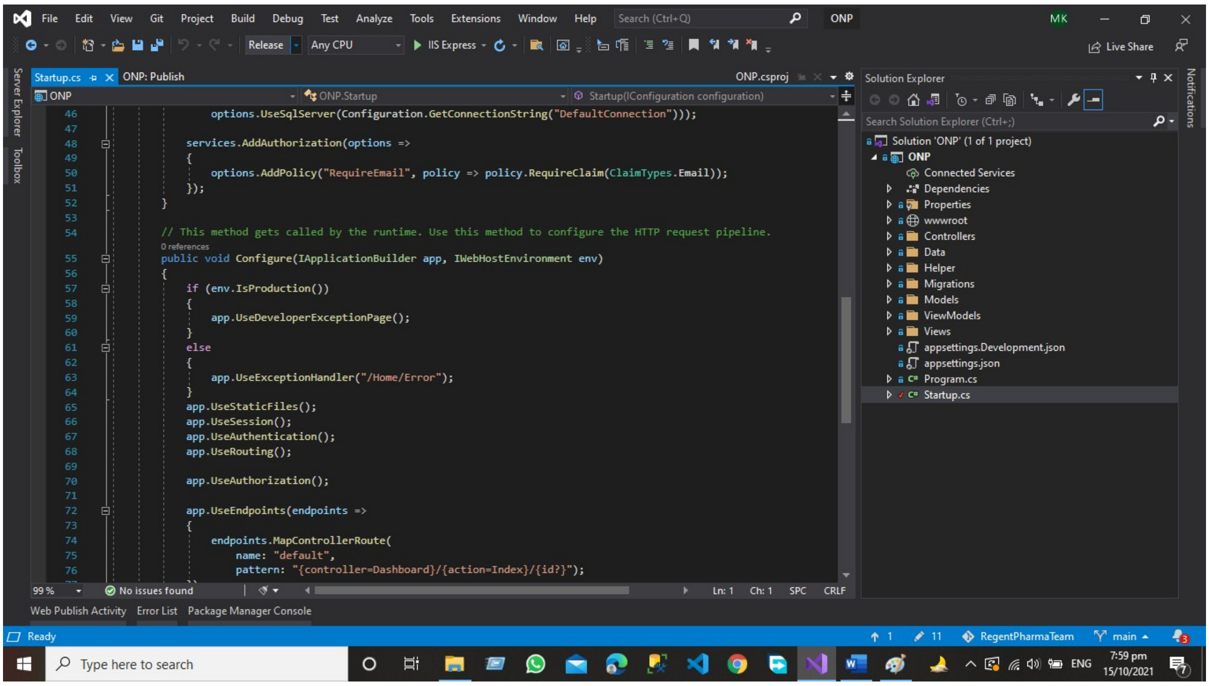
FTP passive mode Unknown

User name [redacted]

Password **********

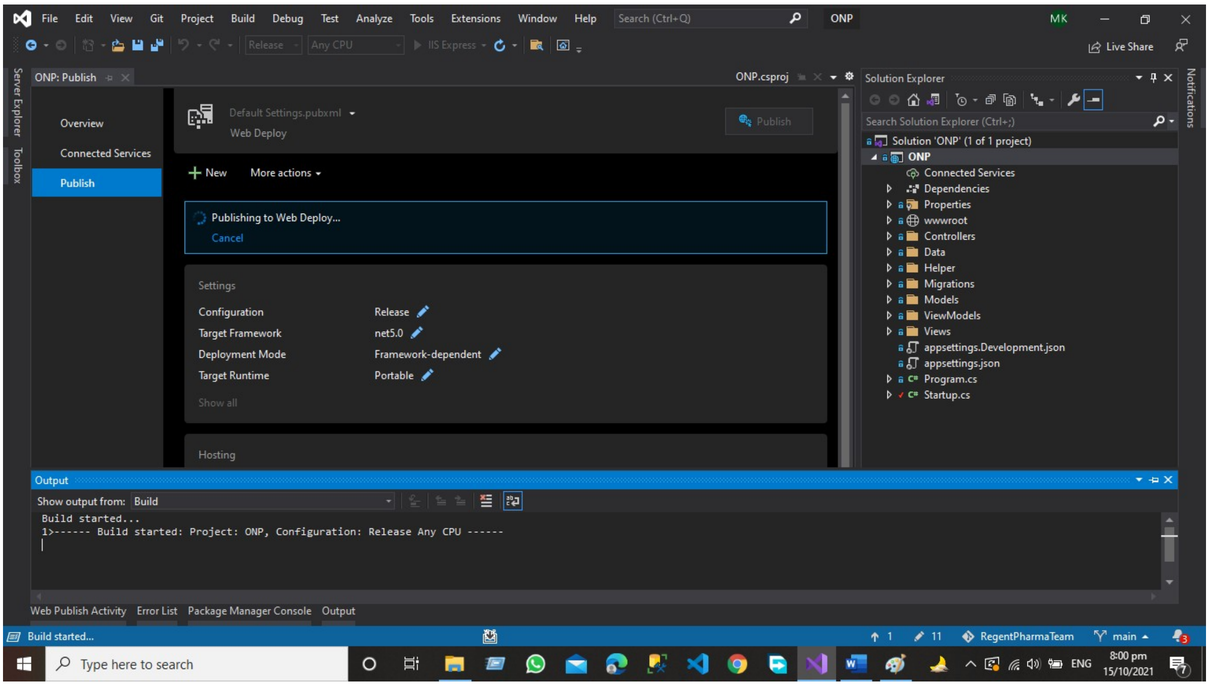
Now Click on Publish

But before the publication just checks your application should be in release mode and the Webhost environment should be in production mode.



```
            options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

            services.AddAuthorization(options =>
            {
                options.AddPolicy("RequireEmail", policy => policy.RequireClaim(ClaimTypes.Email));
            });
        }

        // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsProduction())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Home/Error");
            }
            app.UseStaticFiles();
            app.UseSession();
            app.UseAuthentication();
            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllerRoute(
                    name: "default",
                    pattern: "{controller=Dashboard}/{action=Index}/{id?}");
```

Publication of the website has been started.



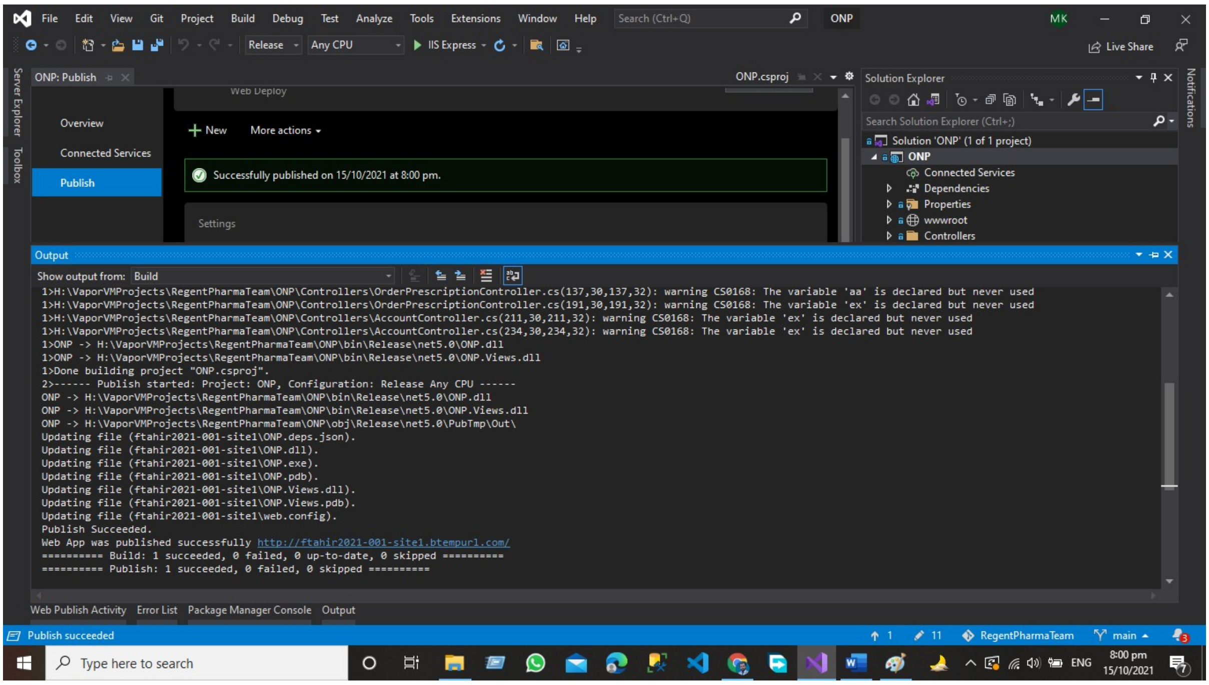
Publishing to Web Deploy...
Cancel

Settings

Configuration	Release
Target Framework	net5.0
Deployment Mode	Framework-dependent
Target Runtime	Portable

```
Build started...
1>------ Build started: Project: ONP, Configuration: Release Any CPU ------
```

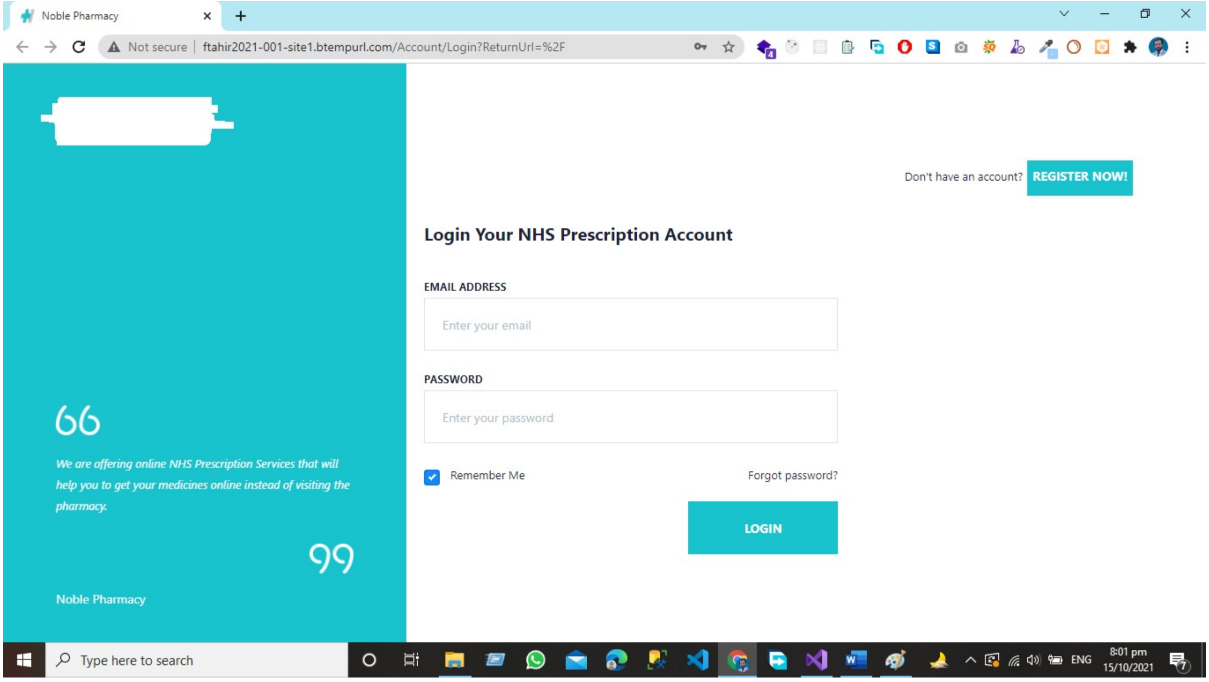
Our Deployment of the website on the server has been successfully done.



Successfully published on 15/10/2021 at 8:00 pm.

```
1>H:\VaporVMProjects\RegentPharmaTeam\ONP\Controllers\OrderPrescriptionController.cs(137,30,137,32): warning CS0168: The variable 'aa' is declared but never used
1>H:\VaporVMProjects\RegentPharmaTeam\ONP\Controllers\OrderPrescriptionController.cs(191,30,191,32): warning CS0168: The variable 'ex' is declared but never used
1>H:\VaporVMProjects\RegentPharmaTeam\ONP\Controllers\AccountController.cs(211,30,211,32): warning CS0168: The variable 'ex' is declared but never used
1>H:\VaporVMProjects\RegentPharmaTeam\ONP\Controllers\AccountController.cs(234,30,234,32): warning CS0168: The variable 'ex' is declared but never used
1>ONP -> H:\VaporVMProjects\RegentPharmaTeam\ONP\bin\Release\net5.0\ONP.dll
1>ONP -> H:\VaporVMProjects\RegentPharmaTeam\ONP\bin\Release\net5.0\ONP.Views.dll
1>Done building project "ONP.csproj".
2>------ Publish started: Project: ONP, Configuration: Release Any CPU ------
ONP -> H:\VaporVMProjects\RegentPharmaTeam\ONP\bin\Release\net5.0\ONP.dll
ONP -> H:\VaporVMProjects\RegentPharmaTeam\ONP\bin\Release\net5.0\ONP.Views.dll
ONP -> H:\VaporVMProjects\RegentPharmaTeam\ONP\obj\Release\net5.0\PubTmp\Out\
Updating file (ftahir2021-001-site1\ONP.deps.json).
Updating file (ftahir2021-001-site1\ONP.dll).
Updating file (ftahir2021-001-site1\ONP.exe).
Updating file (ftahir2021-001-site1\ONP.pdb).
Updating file (ftahir2021-001-site1\ONP.Views.dll).
Updating file (ftahir2021-001-site1\ONP.Views.pdb).
Updating file (ftahir2021-001-site1\web.config).
Publish Succeeded.
Web App was published successfully http://ftahir2021-001-site1.btempurl.com/
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
========== Publish: 1 succeeded, 0 failed, 0 skipped ==========
```

Now we are live on the website.



Noble Pharmacy

Not secure | ftahir2021-001-site1.btempurl.com/Account/Login?ReturnUrl=%2F

Don't have an account? REGISTER NOW!

Login Your NHS Prescription Account

EMAIL ADDRESS

Enter your email

PASSWORD

Enter your password

☑ Remember Me

Forgot password?

LOGIN

We are offering online NHS Prescription Services that will help you to get your medicines online instead of visiting the pharmacy.

Noble Pharmacy

Conclusion

We learned how to publish our asp.net Core web application using Visual Studio in this chapter, and in the next post, we'll learn how to deploy our application using FileZilla Software FTP.

“Happy Coding”

References

1. *https://www.c-sharpcorner.com/*
2. *https://docs.microsoft.com/en-us/*
3. *https://google.com*
4. *https://www.castsoftware.com/*

Feedback

If you want to submit your feedback regarding the book theme book content or any other suggestion for improvement, then scan the QR Code and submit your feedback.