Onion Architecture in Asp.net Core 6 Web API

Sardar Mudassar Ali Khan
Contour Software
29 PUBLICATIONS **0** CITATIONS
SEE PROFILE

Onion Architecture in Asp.net Core 6 Web API

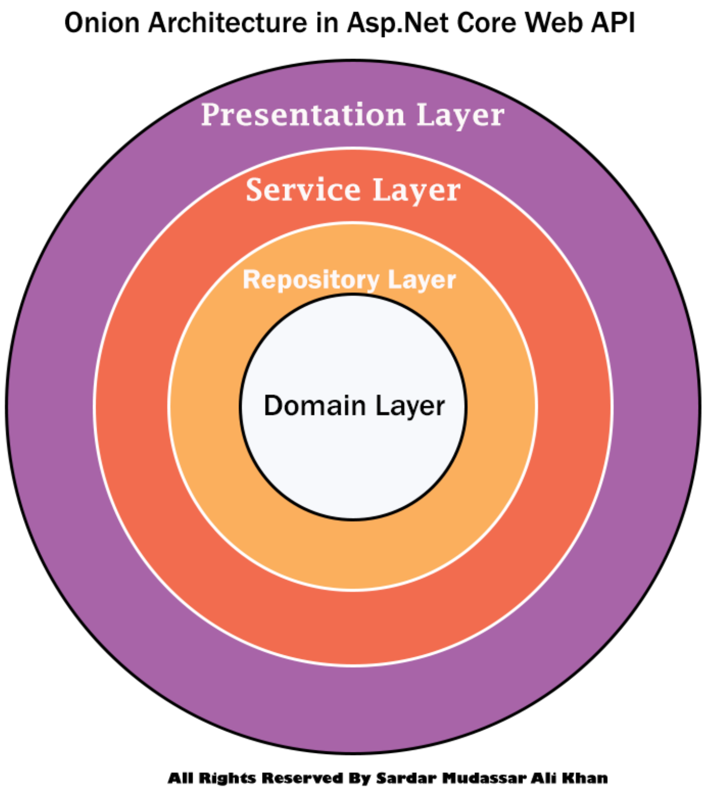
Contents

Introduction

In this article, 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
Presentation Layer
Service Layer
Repository Layer
Domain Layer
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

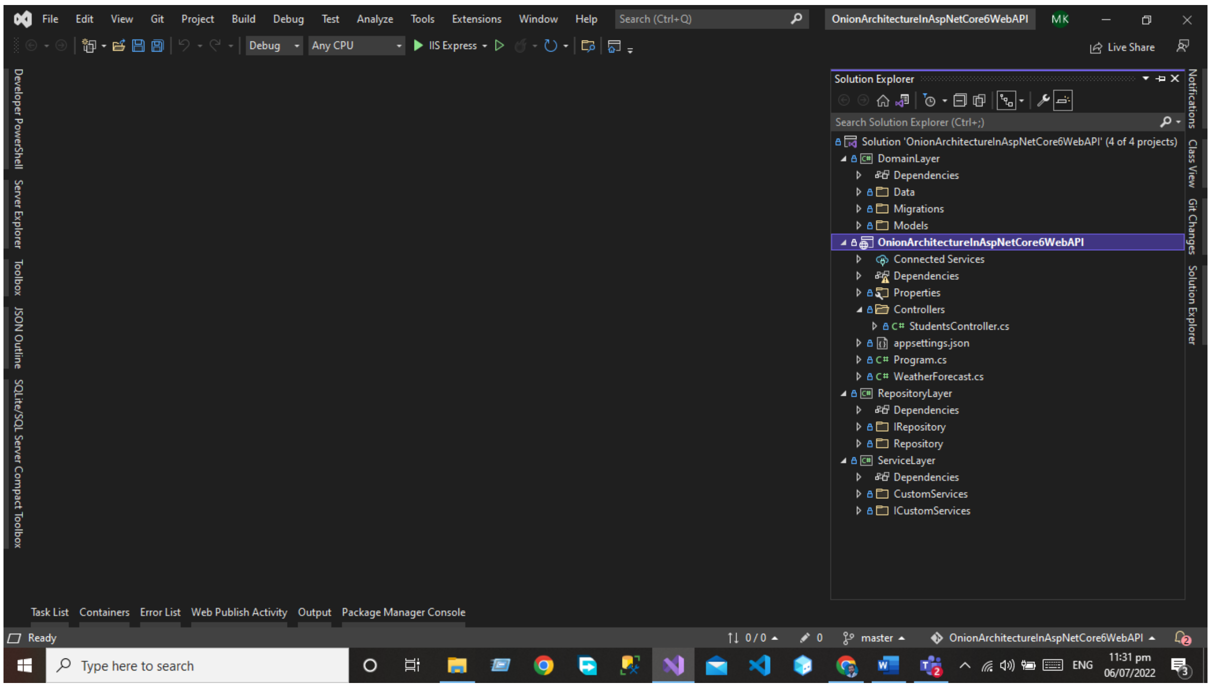
- Onion architecture provides us with the batter maintainability of code because code depends on layers.
- It provides us batter testability for unit tests we can write the separate test cases in layers without affecting the other module in the application.
- Using the onion architecture our application 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.



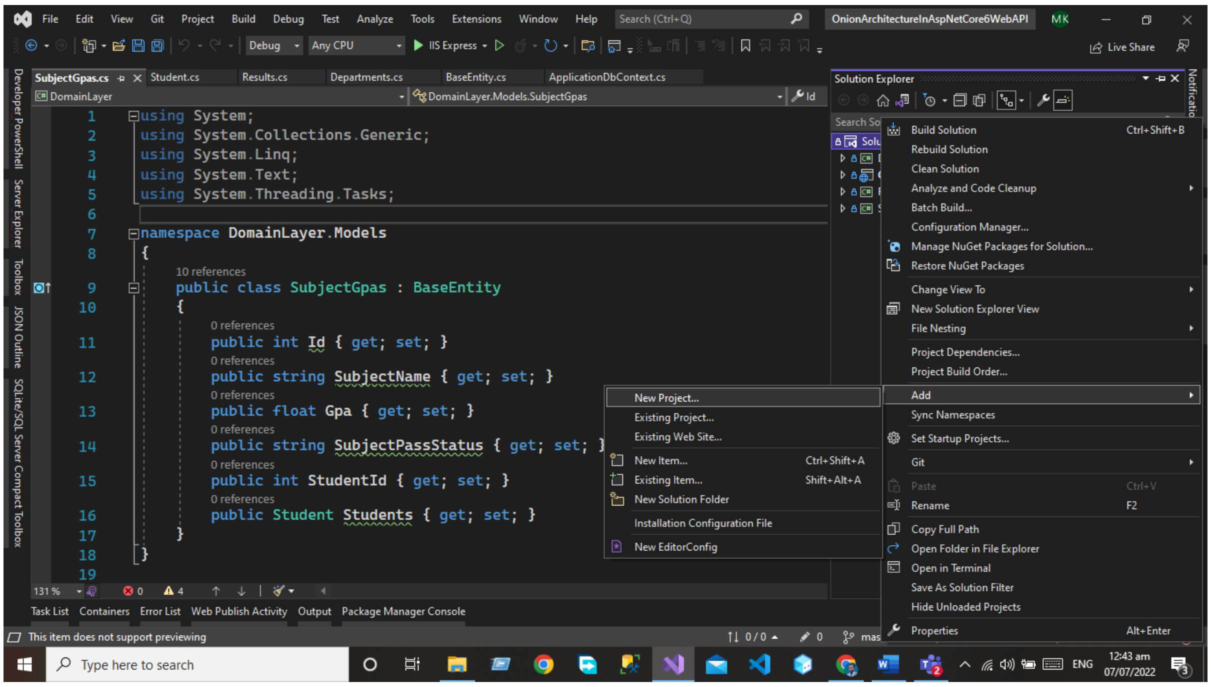
```
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'OnionArchitectureInAspNetCore6WebAPI' (4 of 4 projects)
  DomainLayer
    Dependencies
    Data
    Migrations
    Models
  OnionArchitectureInAspNetCore6WebAPI
    Connected Services
    Dependencies
    Properties
    Controllers
      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

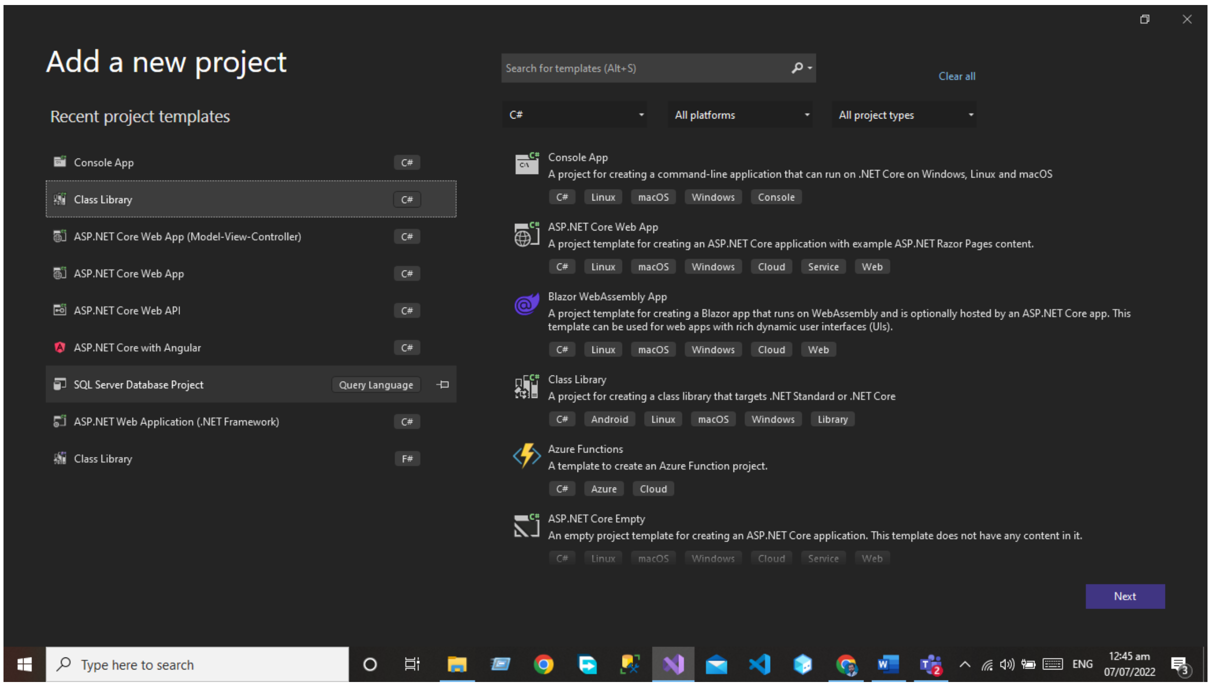
For the Domain layer, we need to add the library project to our application.

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
Next

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

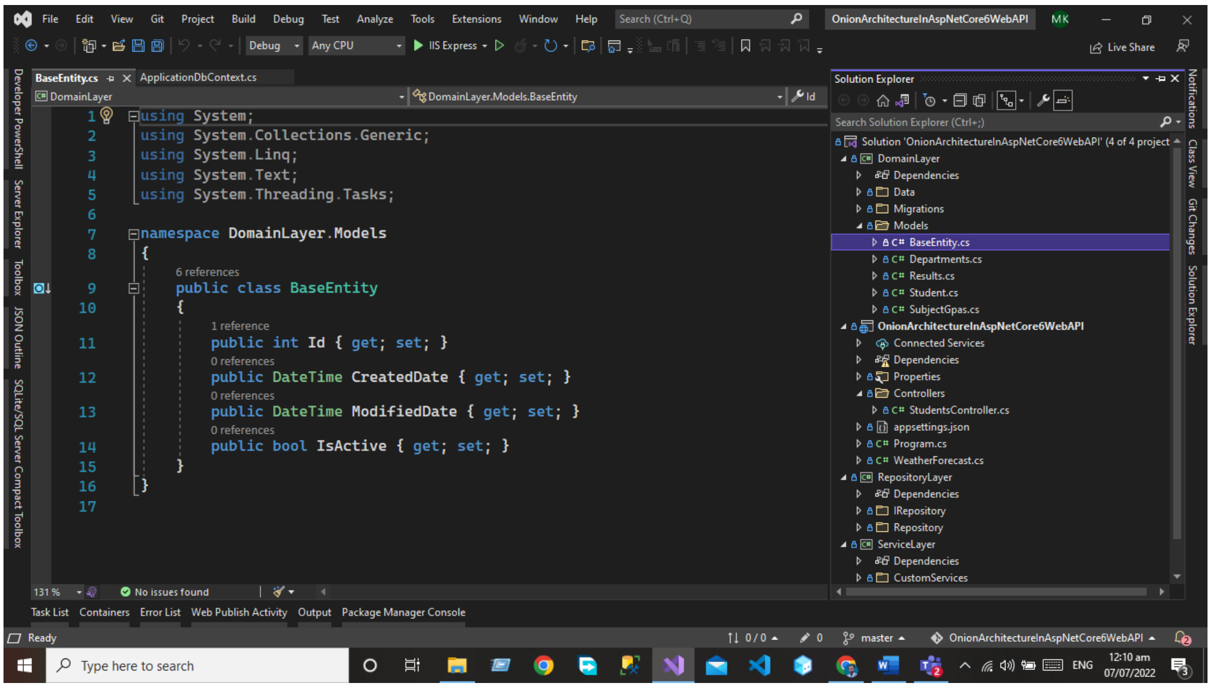
Azure Functions
A template to create an Azure Function project.
C# Azure Cloud

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

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
{
    6 references
    public class BaseEntity
    {
        1 reference
        public int Id { get; set; }
        0 references
        public DateTime CreatedDate { get; set; }
        0 references
        public DateTime ModifiedDate { get; set; }
        0 references
        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.

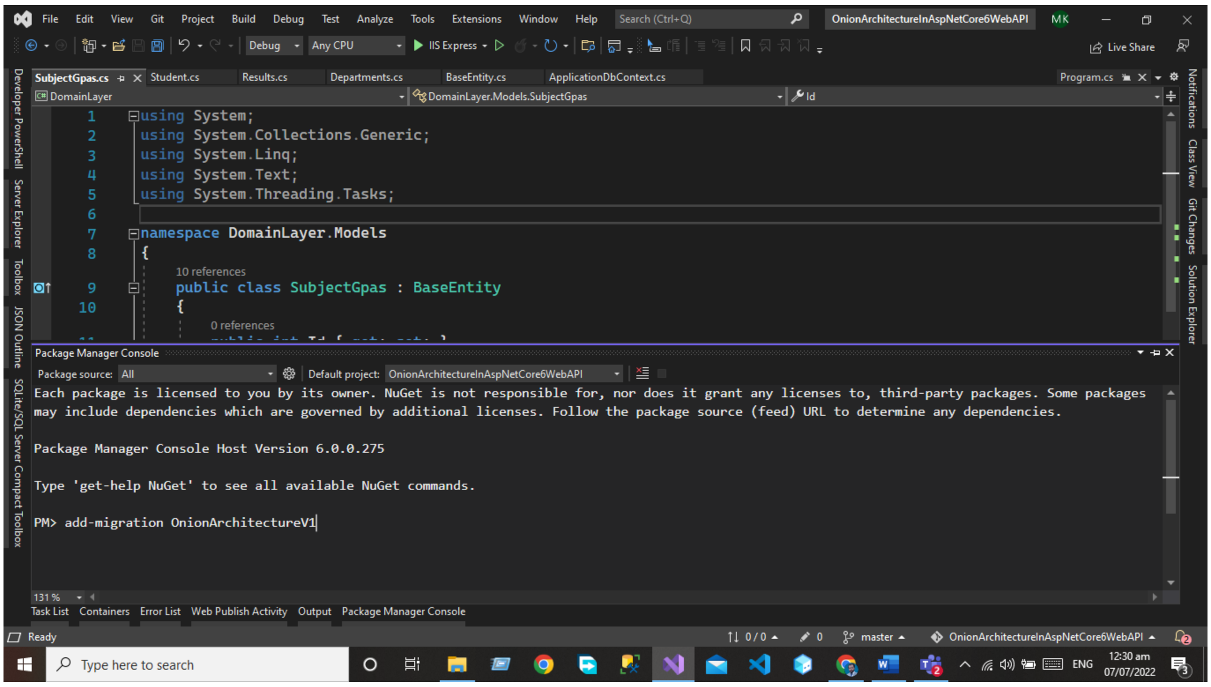
Write click on the application and create the class ApplicationDbContext.cs

```
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.

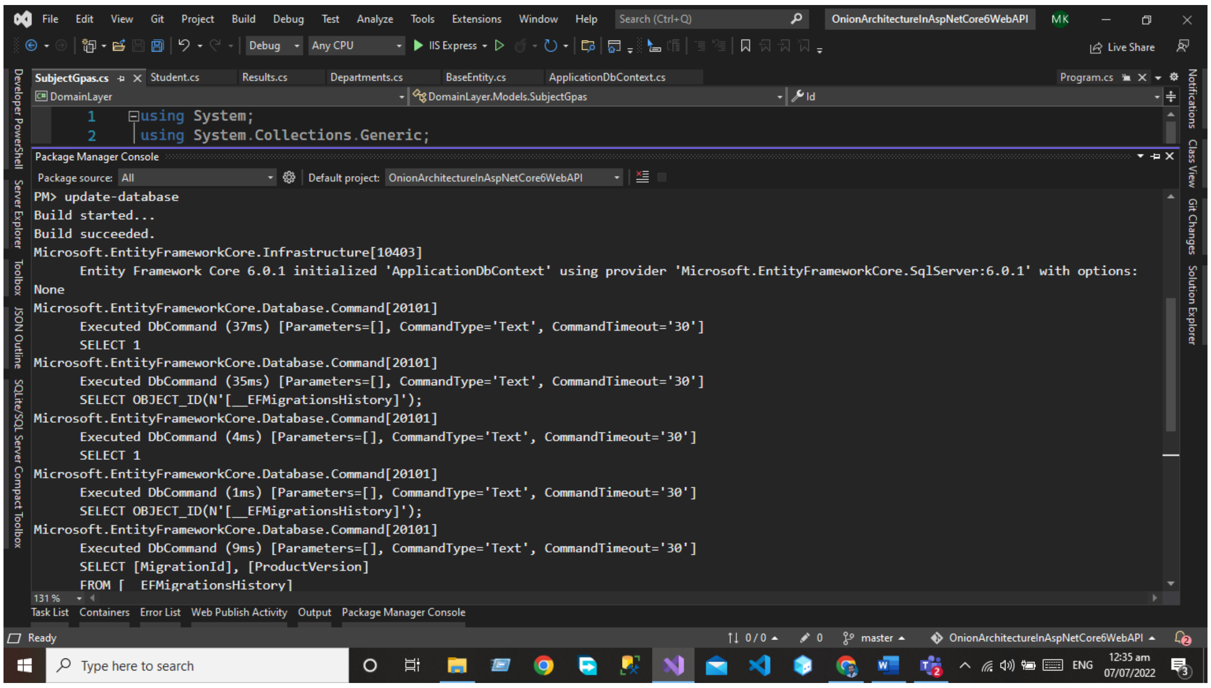
```
add-migration OnionArchitectureV1
```



```
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
```

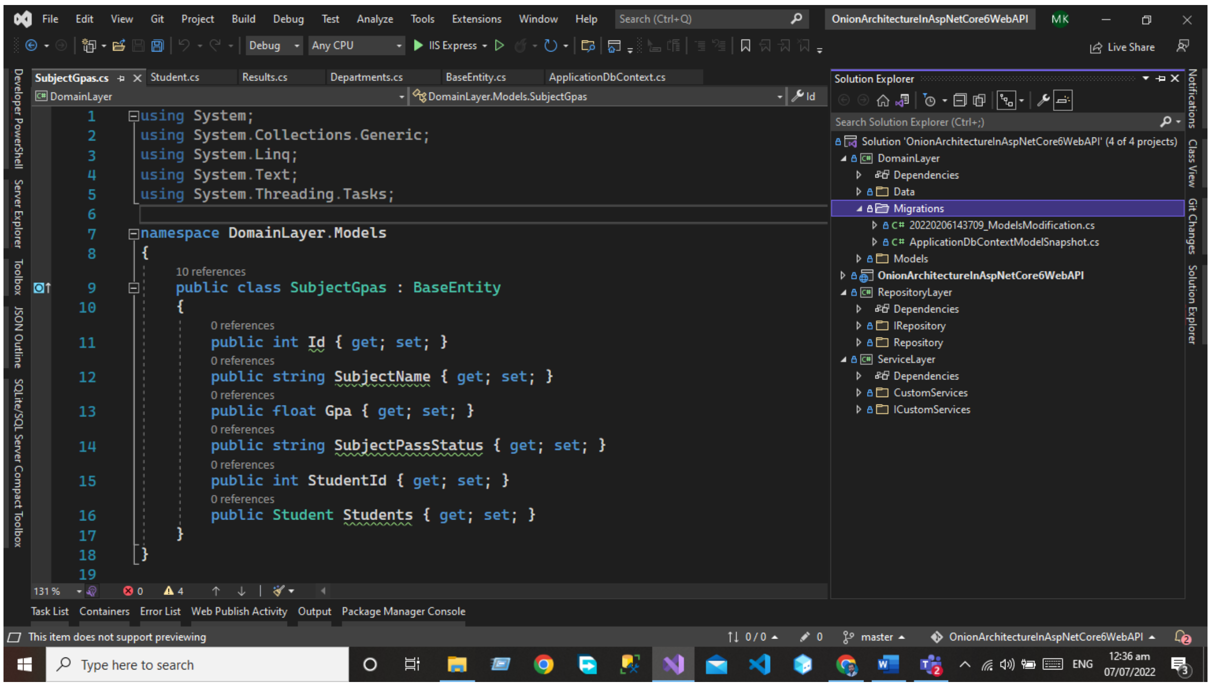
After executing the commands now, you need to update the database by executing the **update-database** Command



```
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.



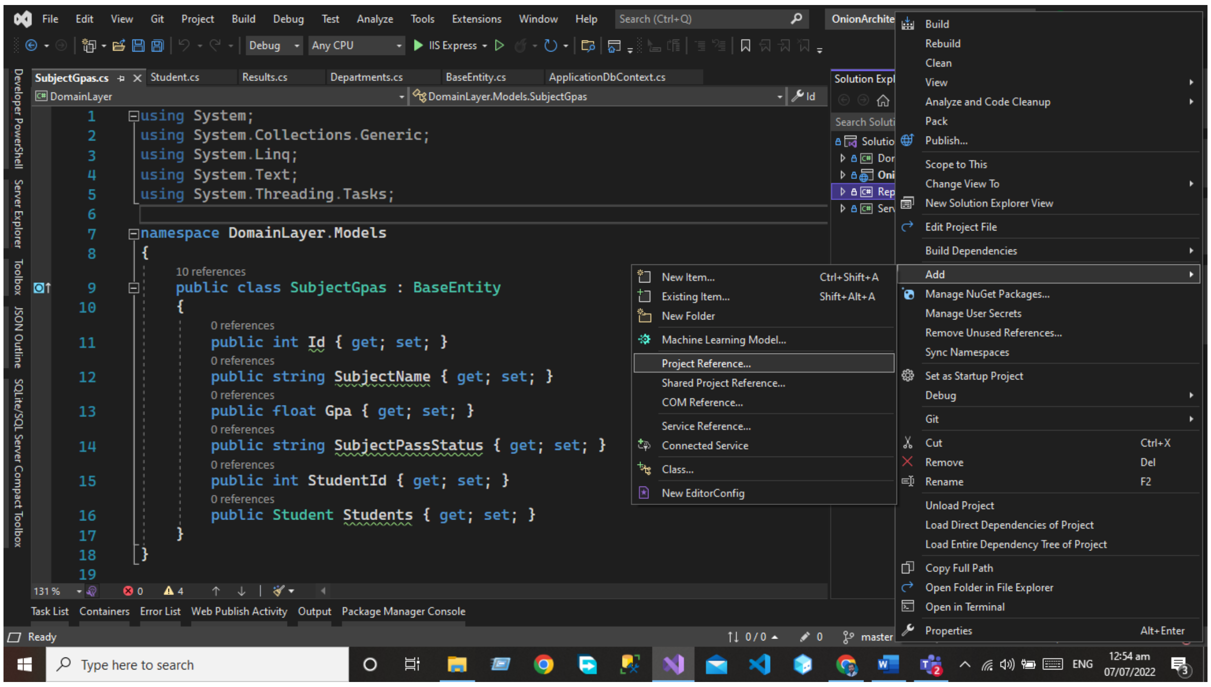
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

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; }
    }
}
```

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.

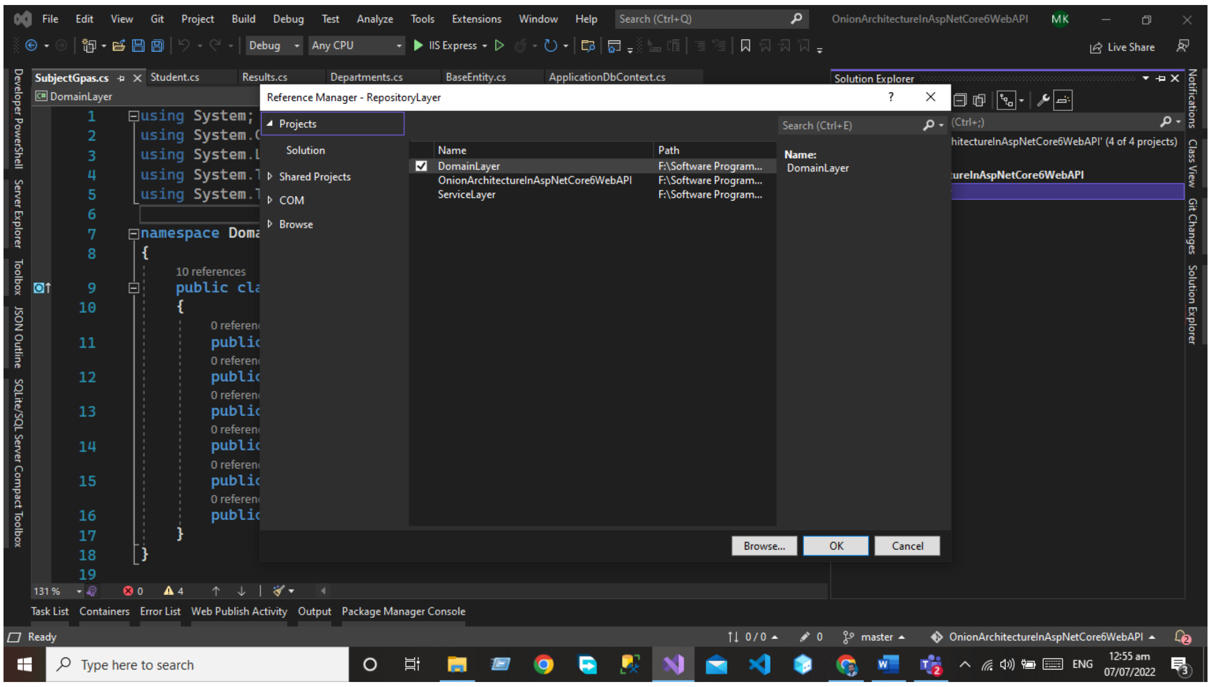
Now we will work on Repository Layer we will follow the same project as we did for the Domain layer add the library project in your applicant and give a name to that project Repository layer.

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

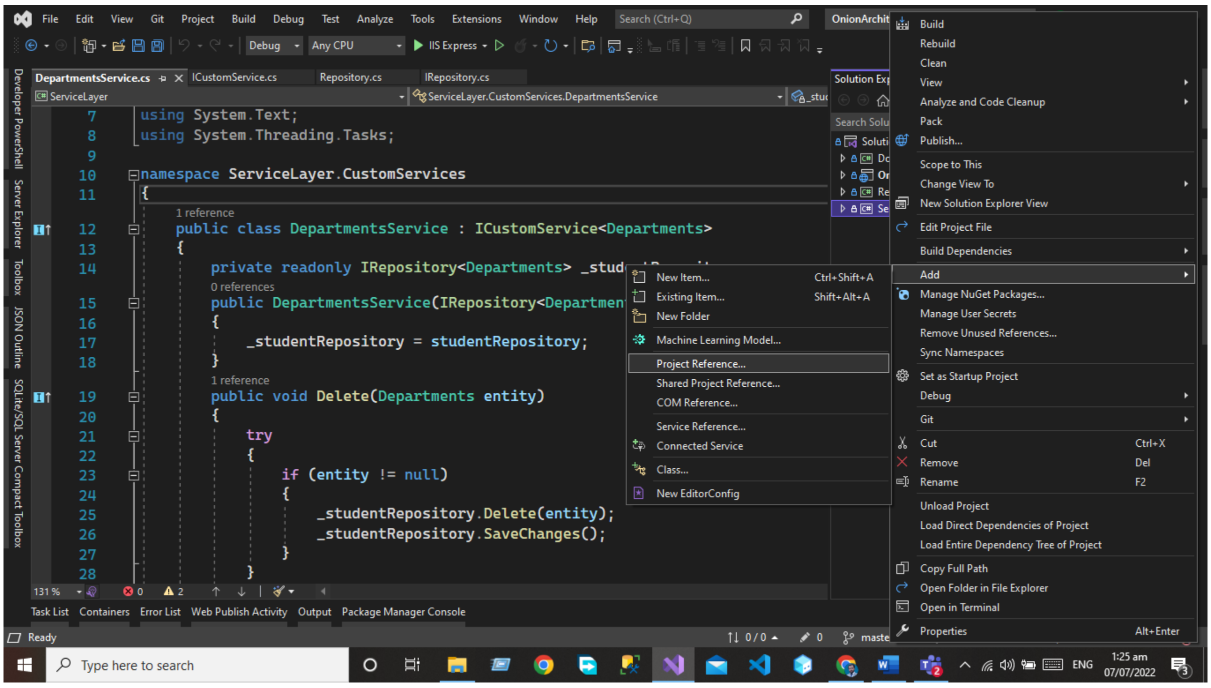
```
using DomainLayer.Data;
using DomainLayer.Models;
using Microsoft.EntityFrameworkCore;
using RepositoryLayer.IRepository;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

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

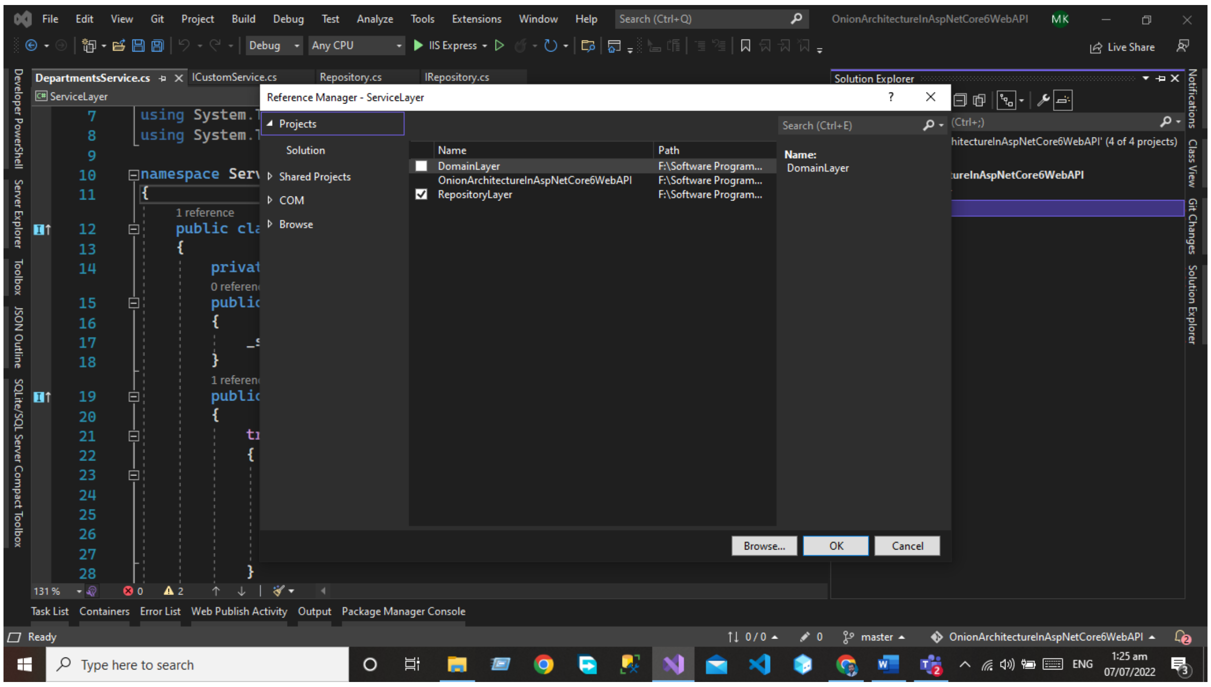
This layer is used to communicate with the presentation and repository layer. The service layer holds all the business logic of the entity. In this layer services interfaces are kept separate from their implementation for loose coupling and separation of concerns.

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.



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 is 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)
```

```csharp
{
    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)
    {
```

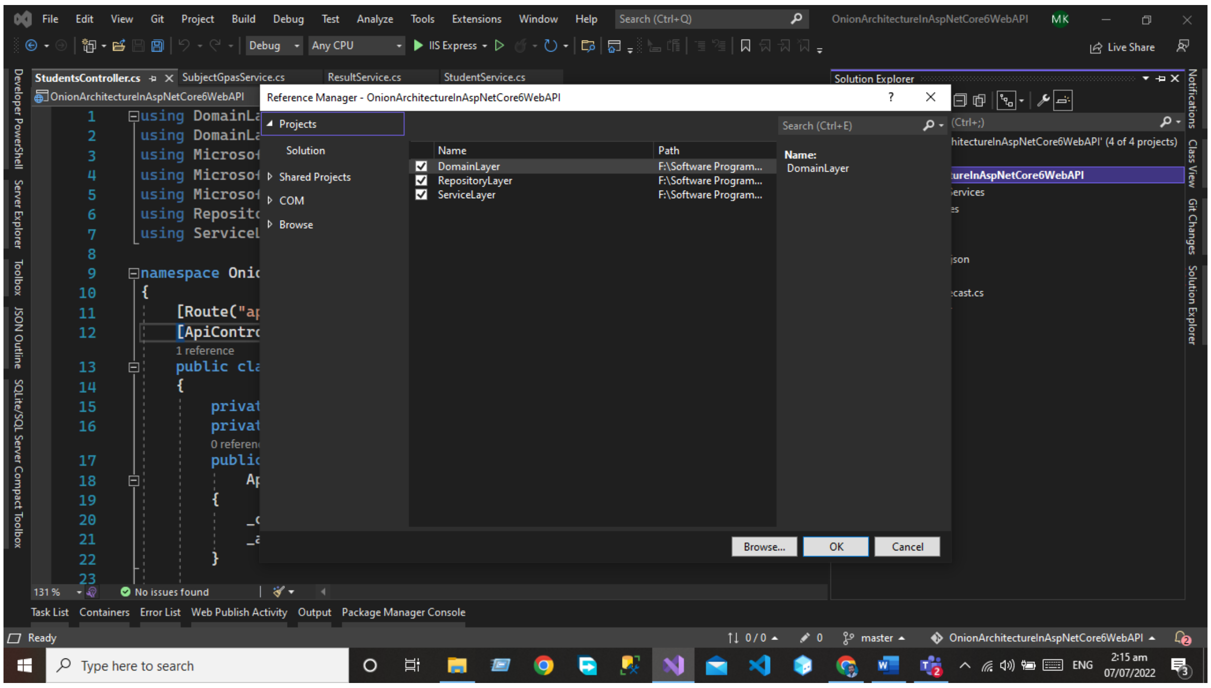
```
                throw;
            }
        }

        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



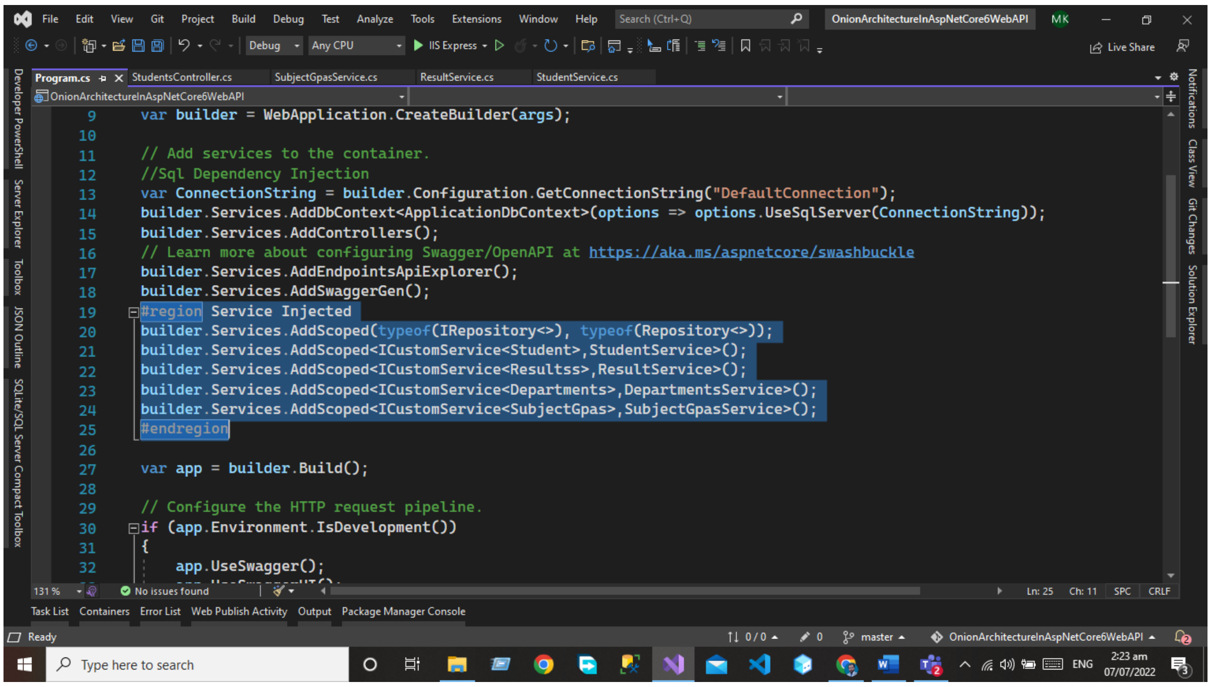
Reference Manager - OnionArchitectureInAspNetCore6WebAPI

	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();
```

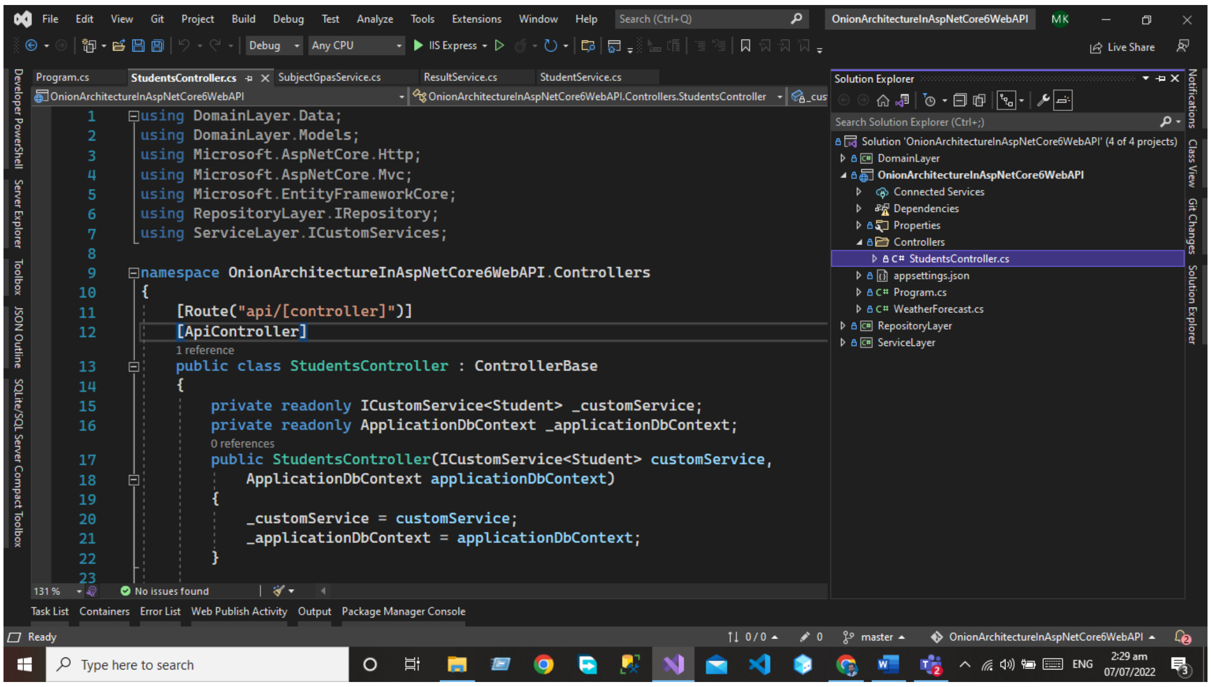
Modify Program.cs File

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 handling 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.



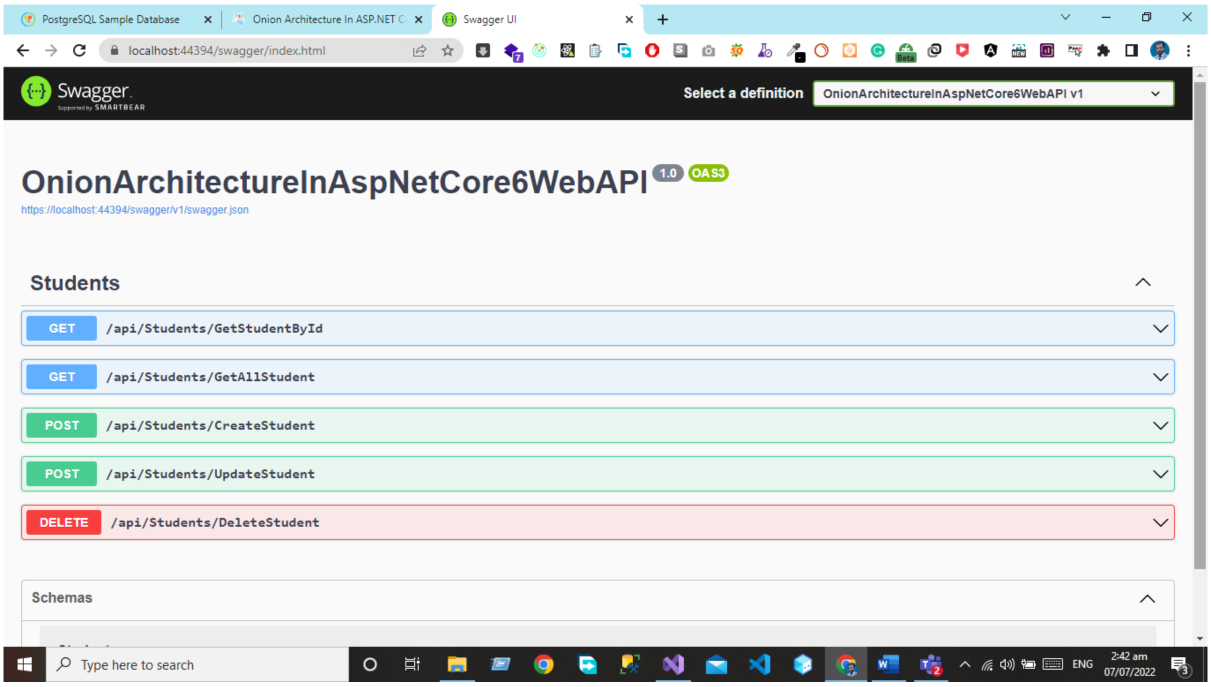
```
using DomainLayer.Data;
using DomainLayer.Models;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using RepositoryLayer.IRepository;
using ServiceLayer.ICustomServices;

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;
        }
```

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.



OnionArchitectureInAspNetCore6WebAPI 1.0 OAS3

https://localhost:44394/swagger/v1/swagger.json

Select a definition: OnionArchitectureInAspNetCore6WebAPI v1

Students

GET /api/Students/GetStudentById

GET /api/Students/GetAllStudent

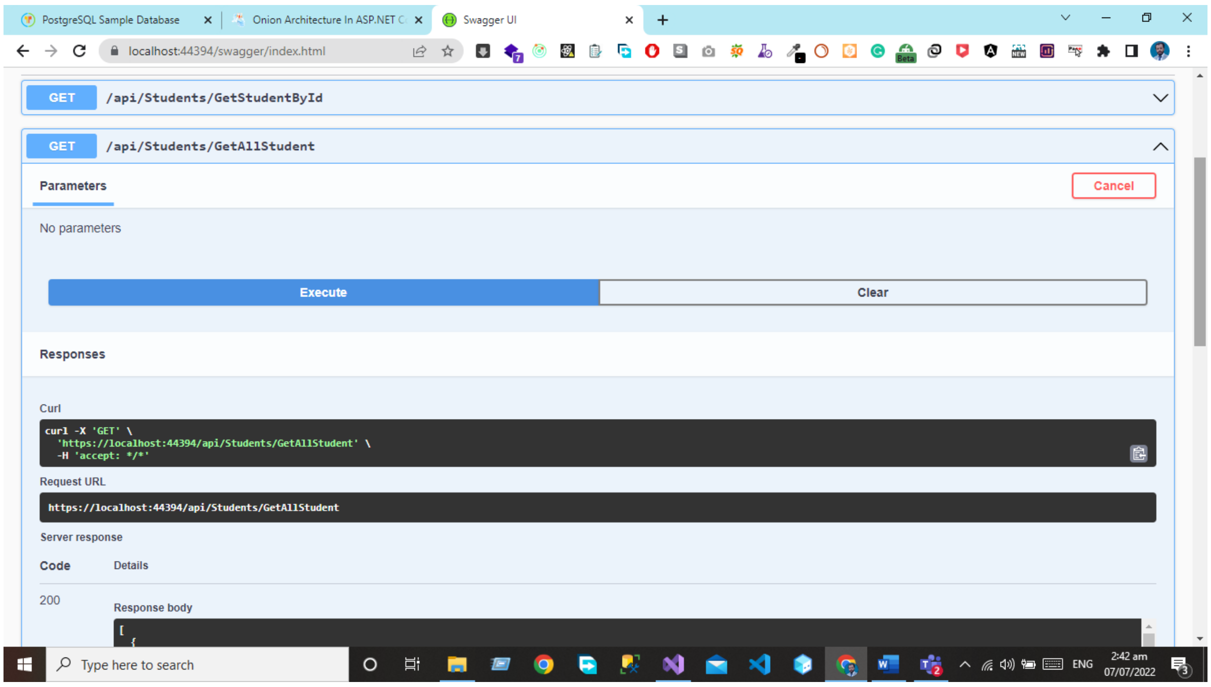
POST /api/Students/CreateStudent

POST /api/Students/UpdateStudent

DELETE /api/Students/DeleteStudent

Schemas

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.



GET /api/Students/GetStudentById

GET /api/Students/GetAllStudent

Parameters

No parameters

Responses

Curl

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

Request URL

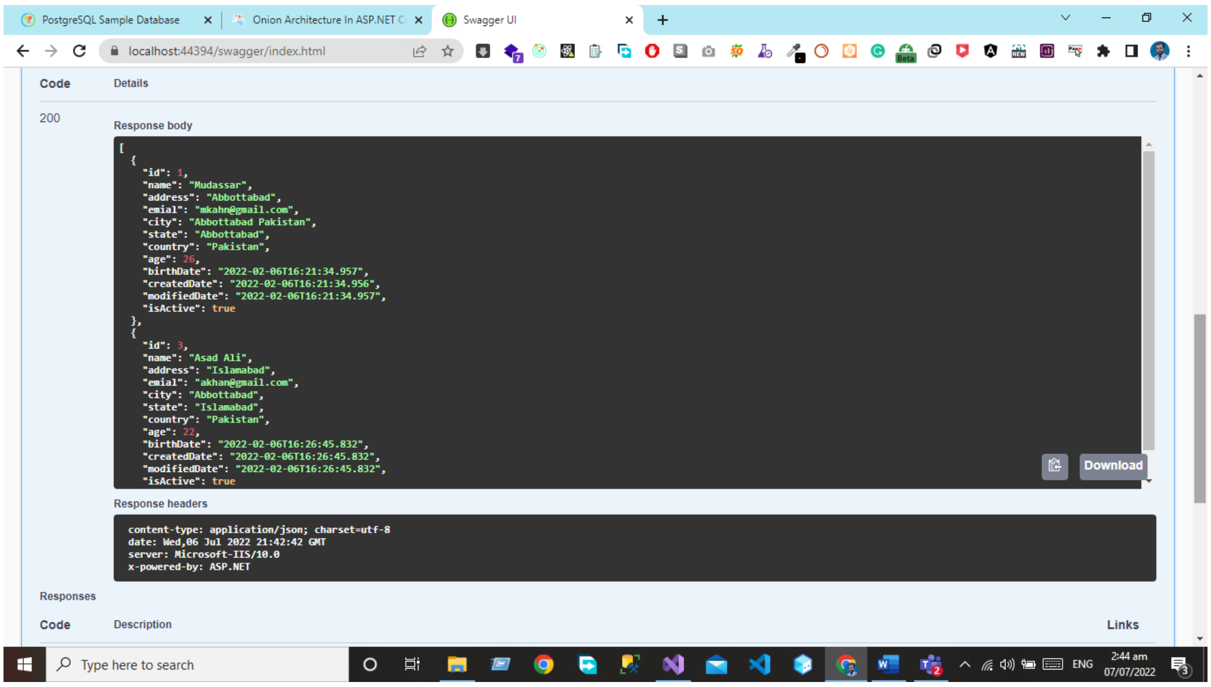
```
https://localhost:44394/api/Students/GetAllStudent
```

Server response

Code: 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
```

```
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
```

Conclusion

In this article, 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.