Clean Architecture in Asp.net Core Web API

Article · July 2022

CITATIONS
0

READS
6,356

1 author:

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

SEE PROFILE

Clean Architecture in Asp.net Core Web API

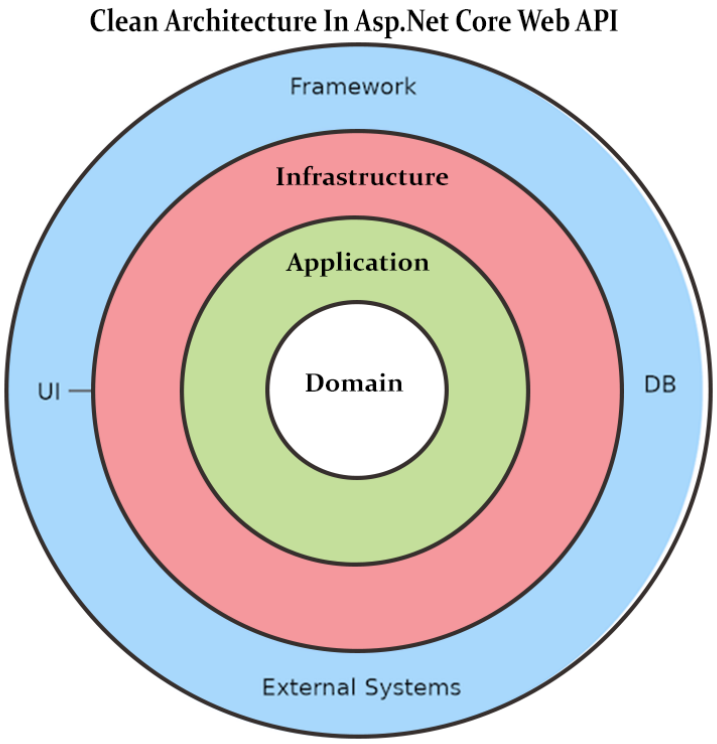
Contents

Introduction

In this article, 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.



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

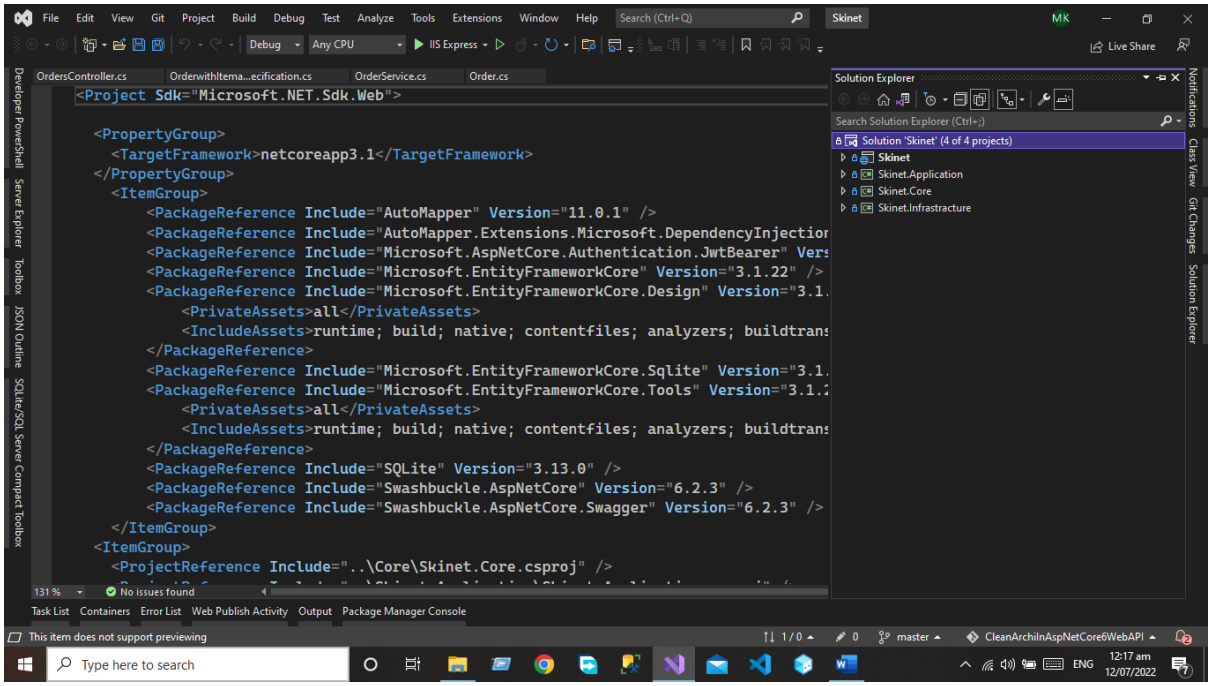
In the case of the API Presentation layer that presents us the object data from the database using the HTTP request in the form of JSON Object. But in the case of front-end applications, we present the data using the UI by consuming the APIS.

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.



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

Let’s Implement the layer practically.

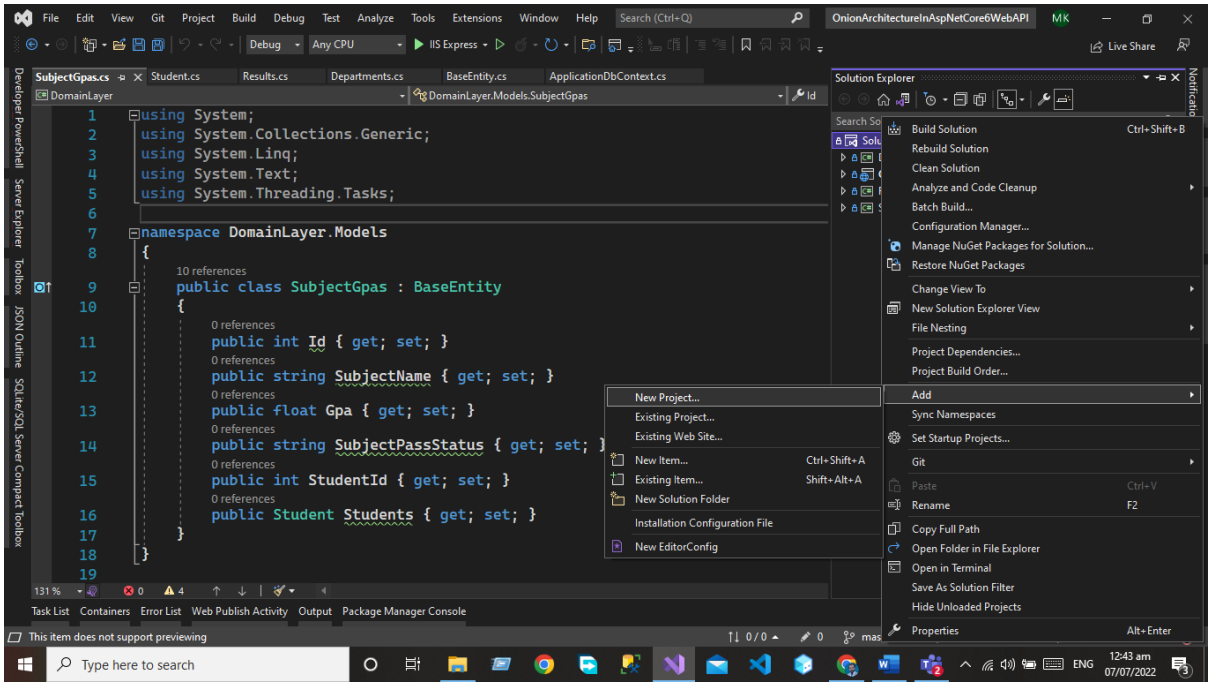
Domain Layer

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.

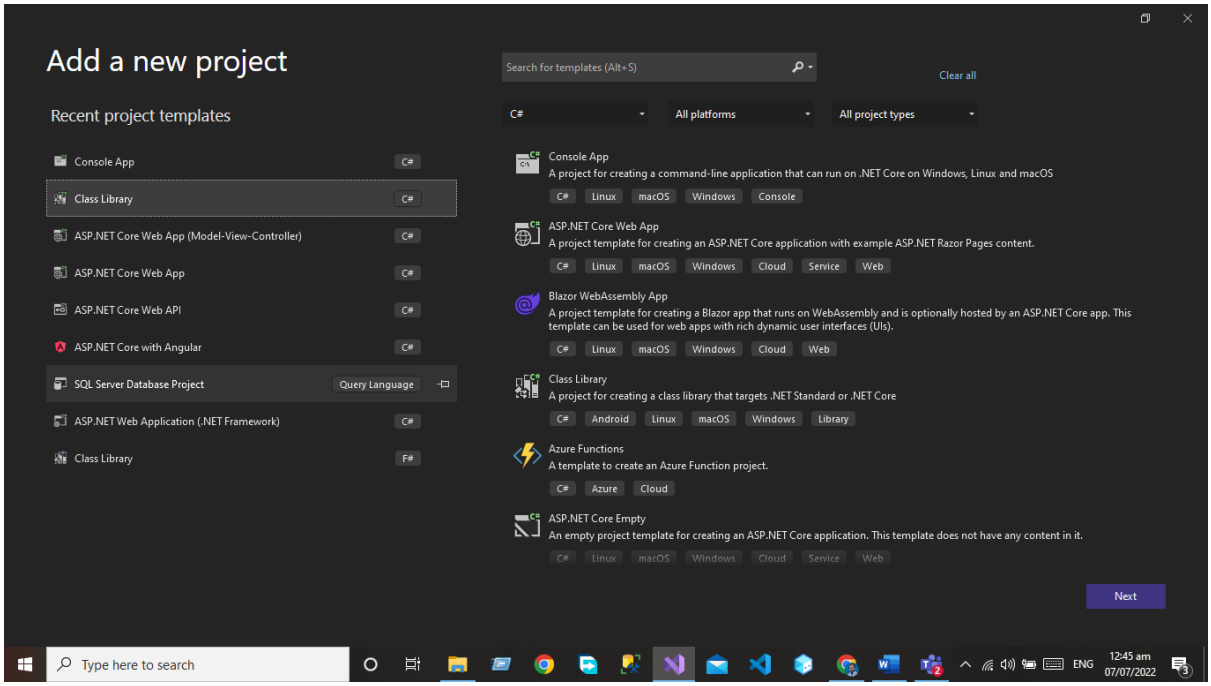
Implementation of Domain layer

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.



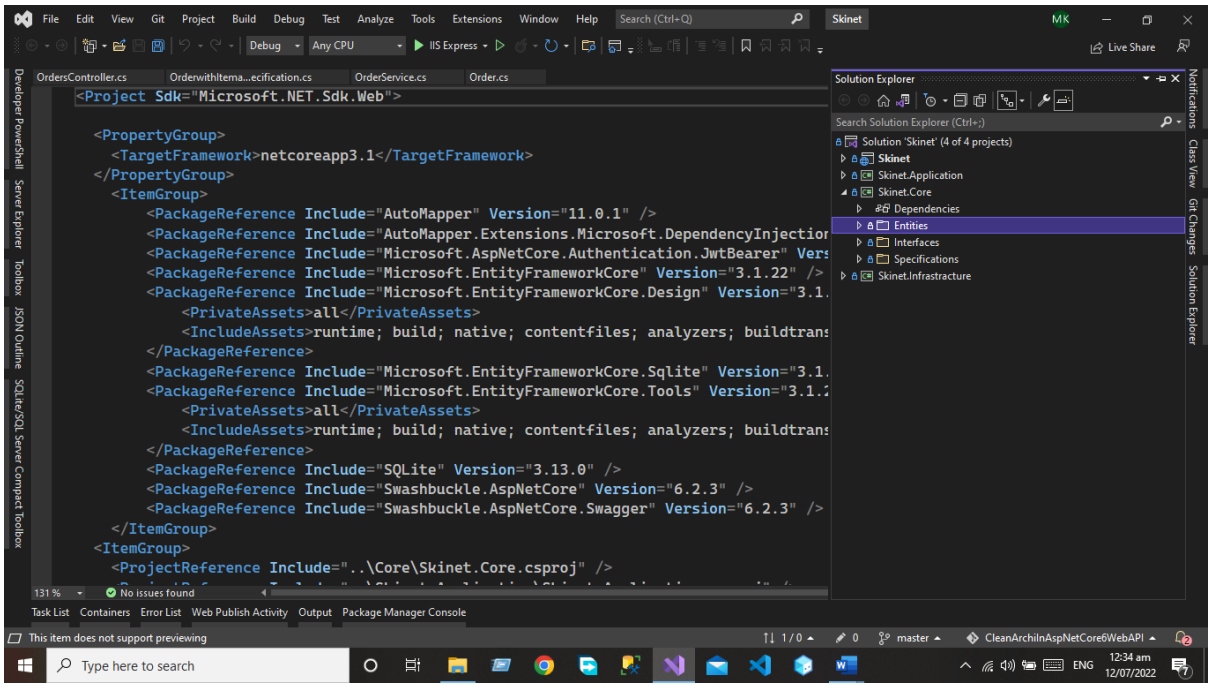
Add the library project to your solution



Name this project as Domain Layer

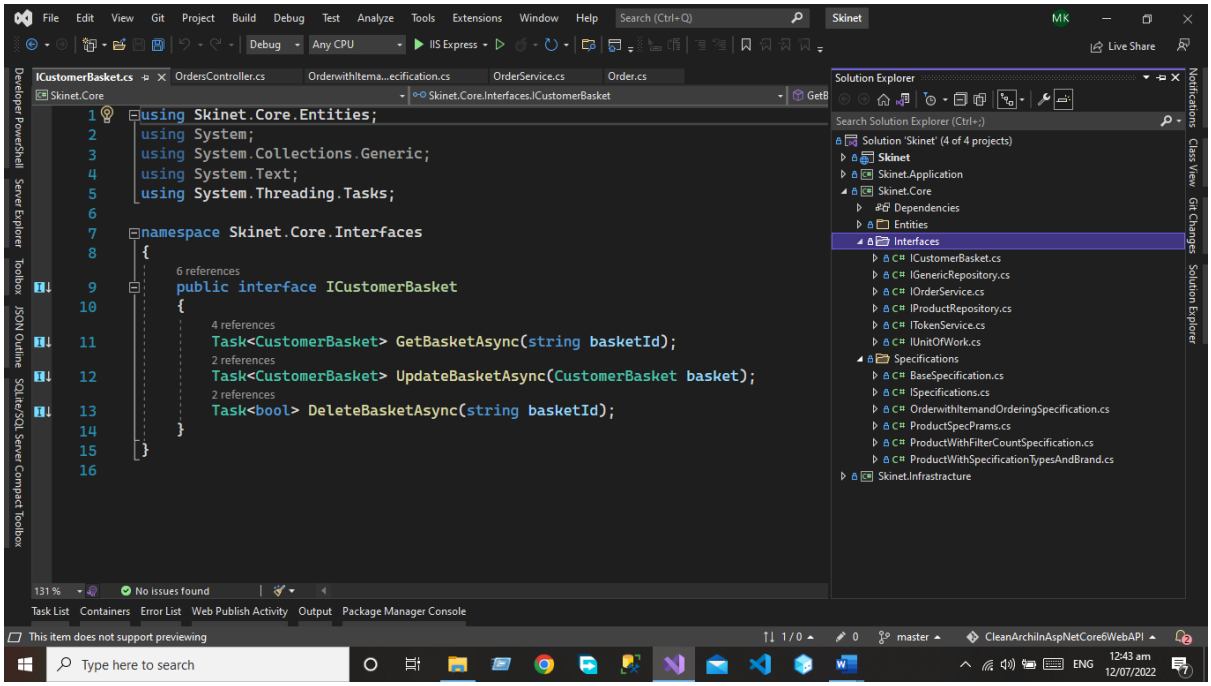
Entities Folder

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.



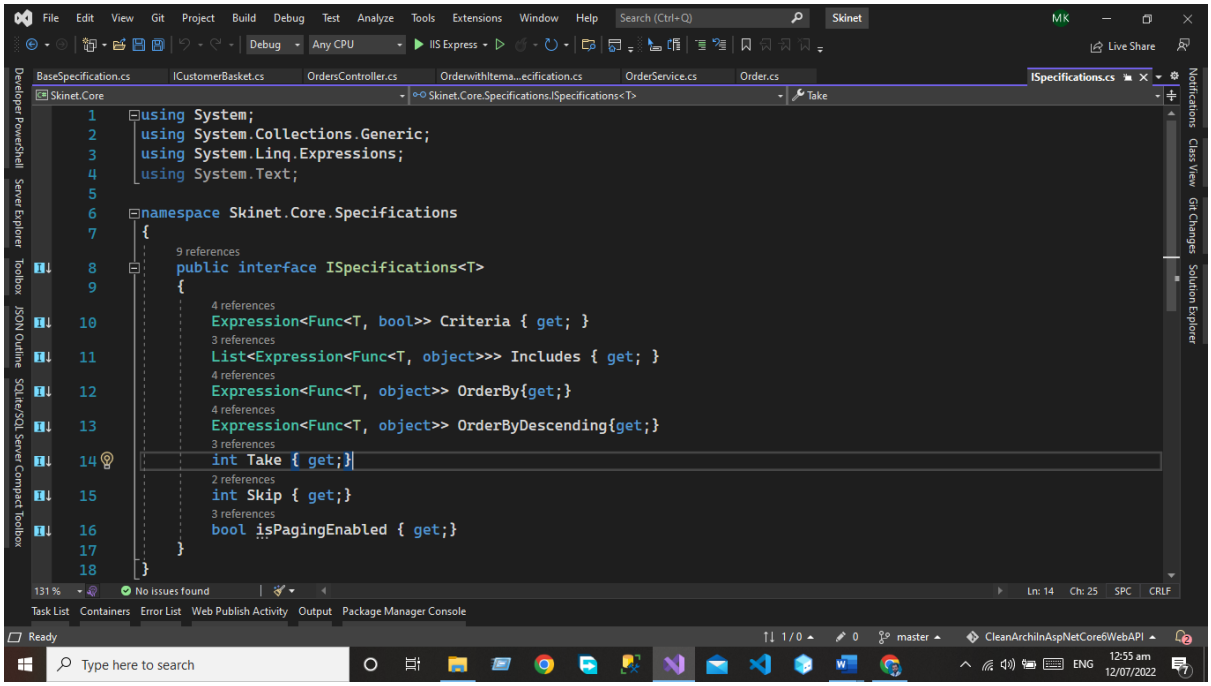
Interface Folder

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



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. Lets us take the example



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;
        }
    }
}
```

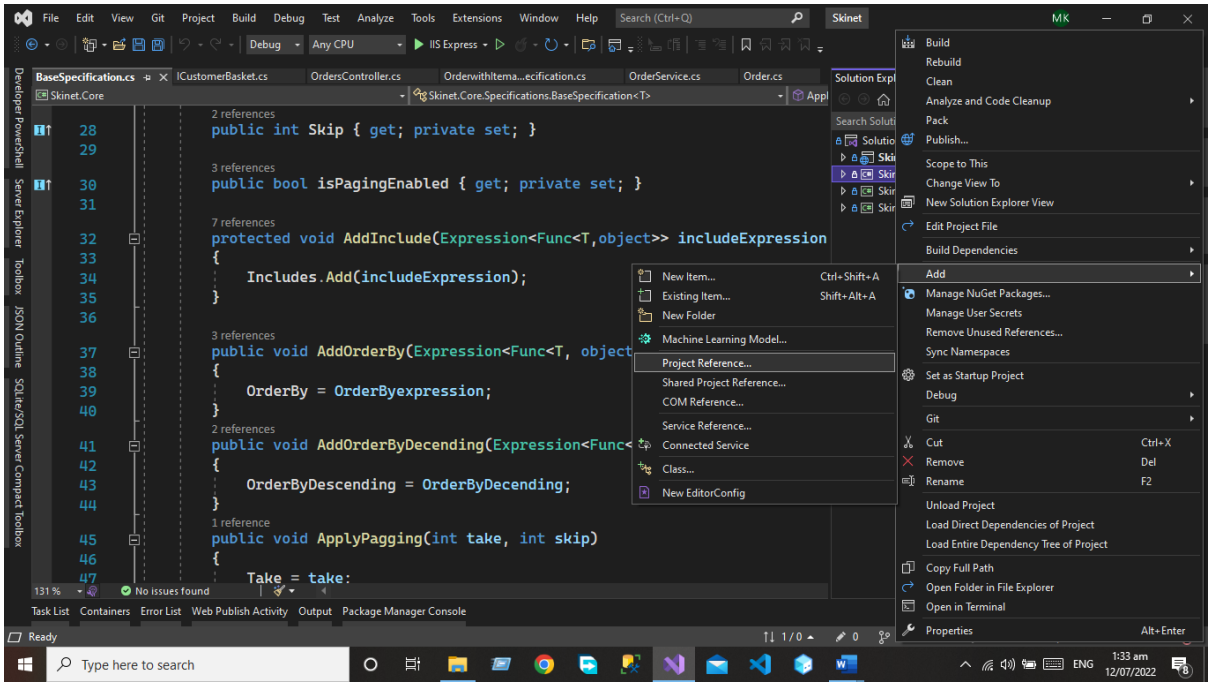
Application Layer

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.

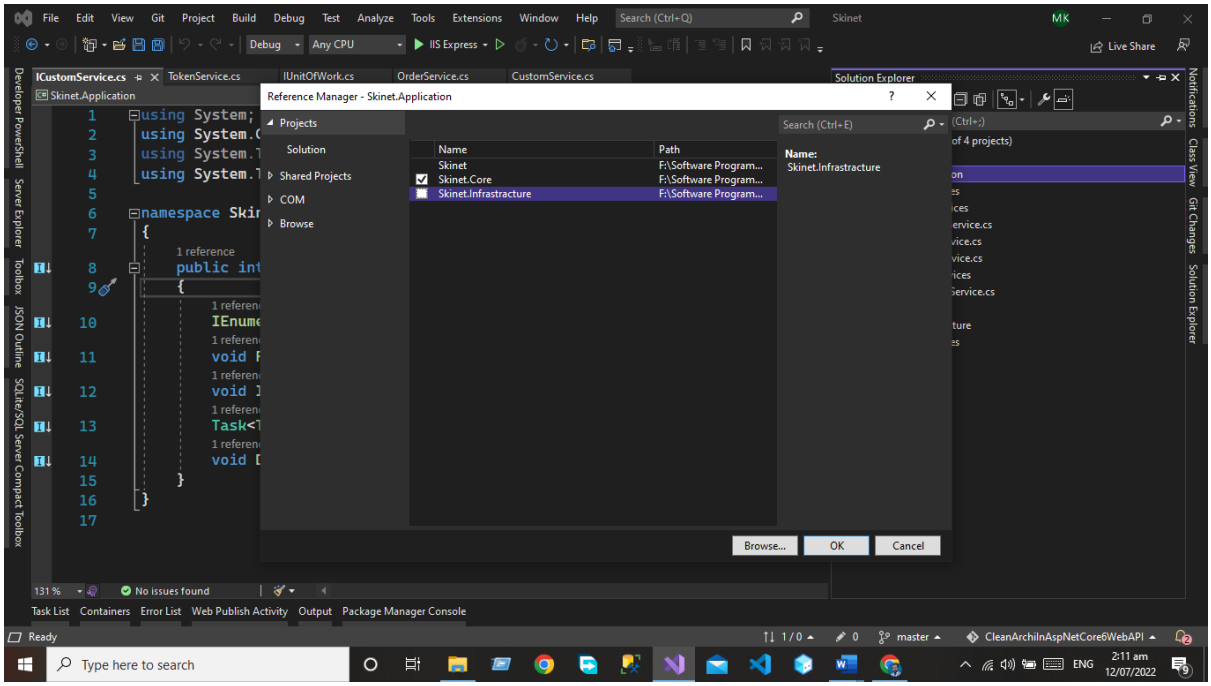
Now we will add the application layer to our application

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.



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



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

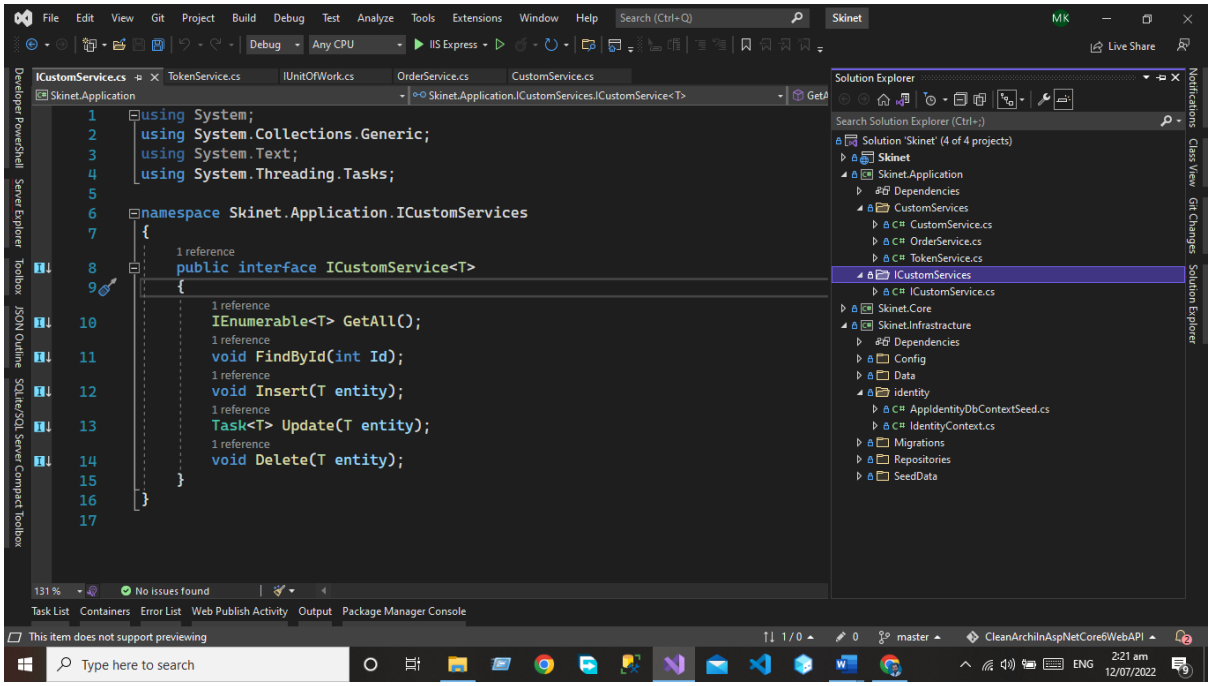
Let's add the I Custom services folder in our application in this folder we will add the ICustom Service Interfaced that will be Inherited by all the services we will add in our Customer Service folder.

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



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.

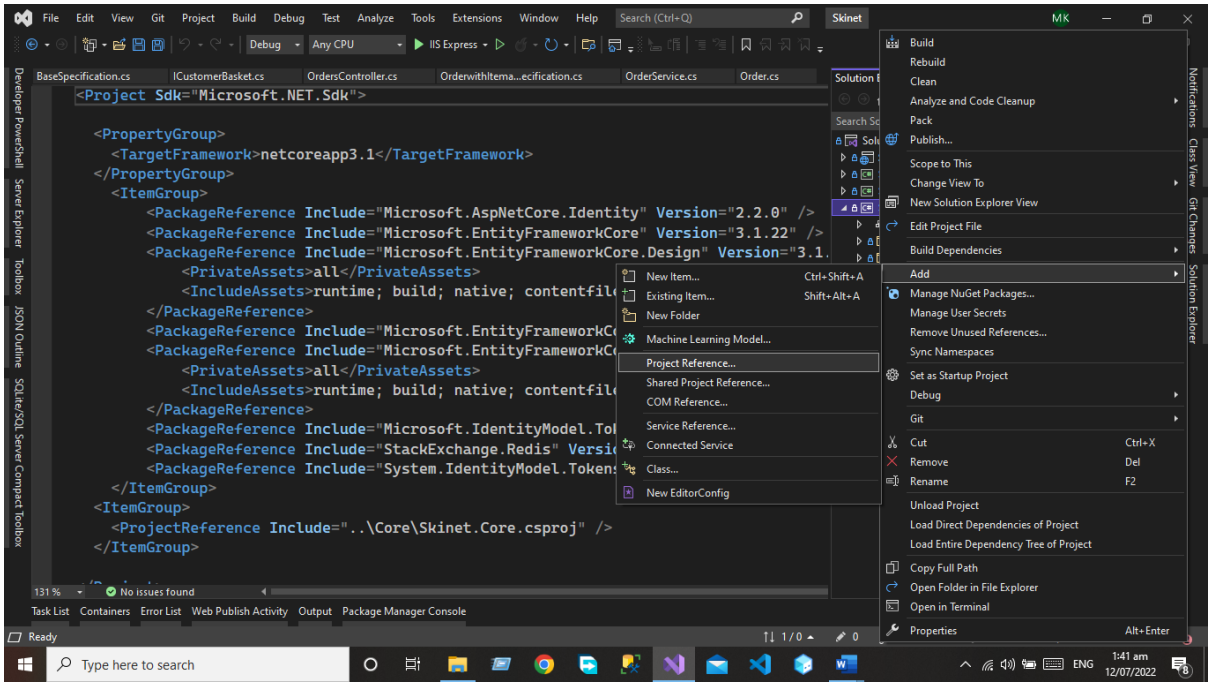
Infrastructure Layer

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.

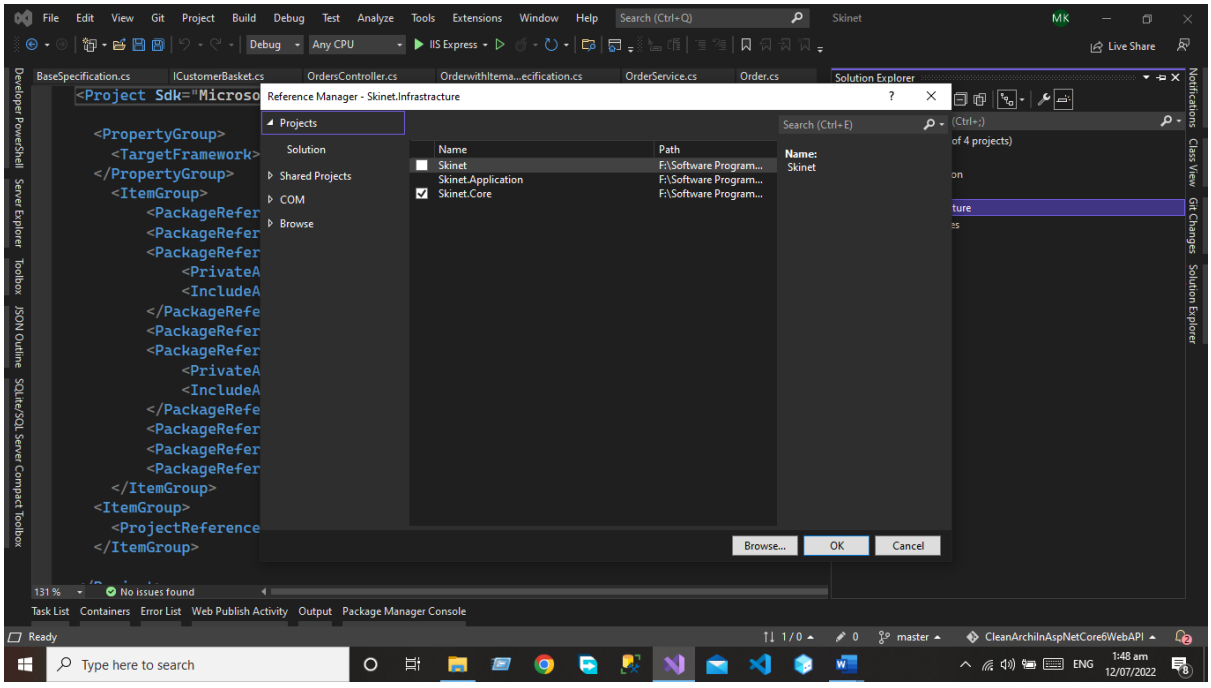
Now we will add the infrastructure layer to our application

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.



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



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 I Basket Interface from the core layer.

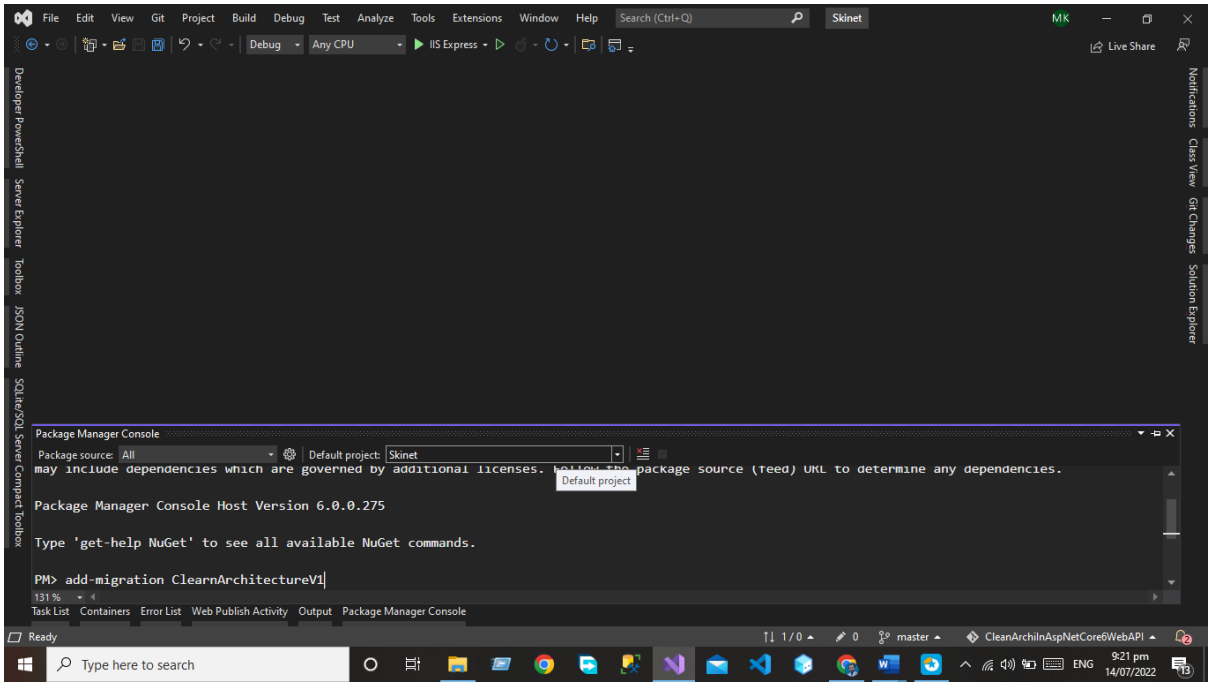
```
using Skinet.Core.Entities;
using Skinet.Core.Interfaces;
using StackExchange.Redis;
using System;
using System.Collections.Generic;
using System.Text;
using System.Text.Json;
using System.Threading.Tasks;

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);
        }
    }
}
```

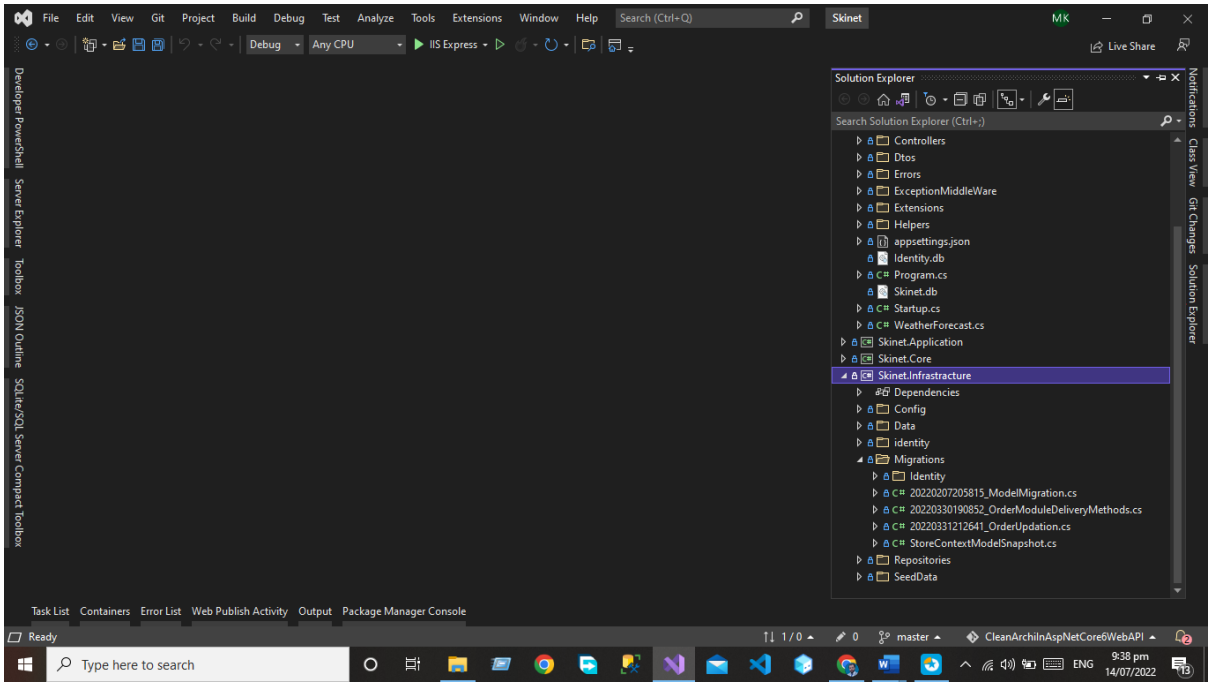
Migrations

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

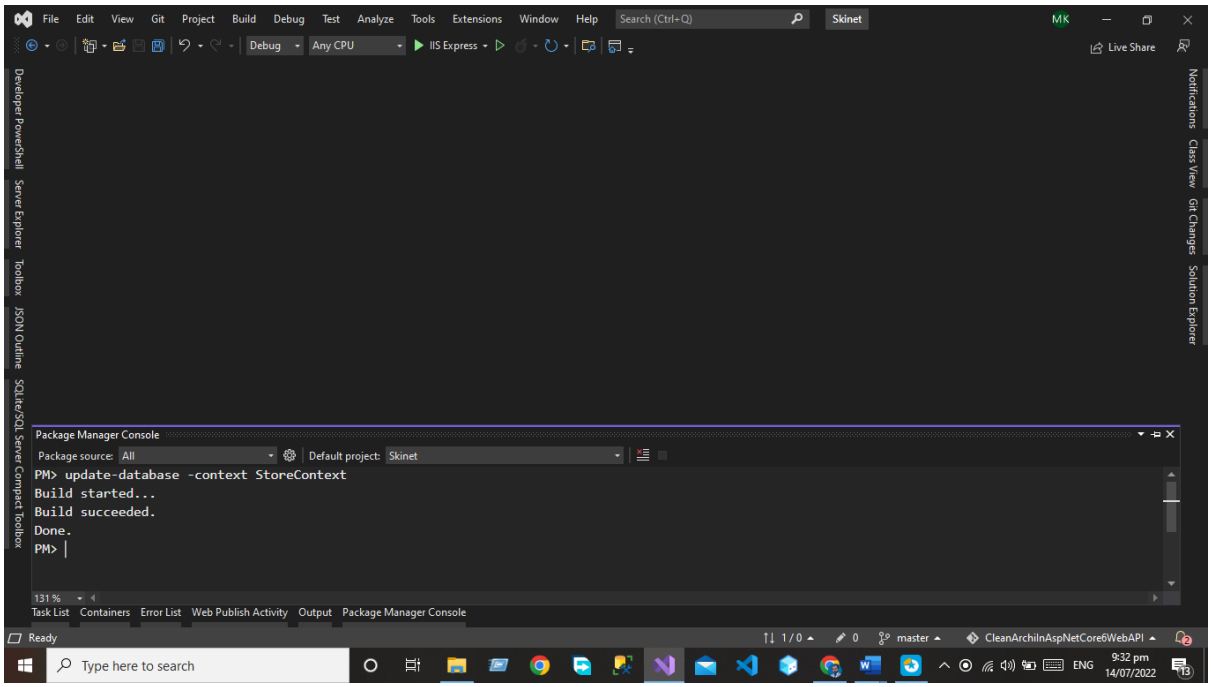
```
add-migration ClearnArchitectureV1
```



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



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



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)
```

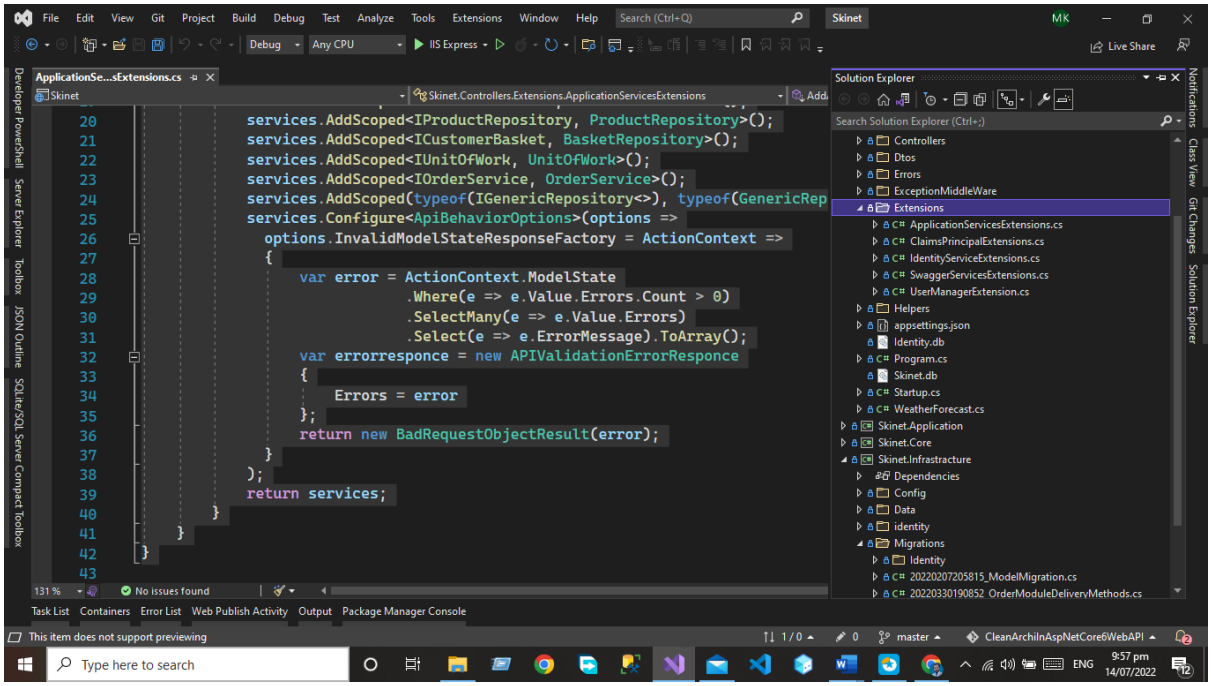
```
                                .SelectMany(e => e.Value.Errors)
                                .Select(e => e.ErrorMessage).ToArray();
                    var errorresponce = new APIValidationErrorResponce
                    {
                        Errors = error
                    };
                    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.Infrastructure.Data;
using Skinet.Infrastructure.Repositories;
using Microsoft.EntityFrameworkCore;
using Skinet.Infrastructure.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.Infrastructure.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();
            });
        }
    }
}
```

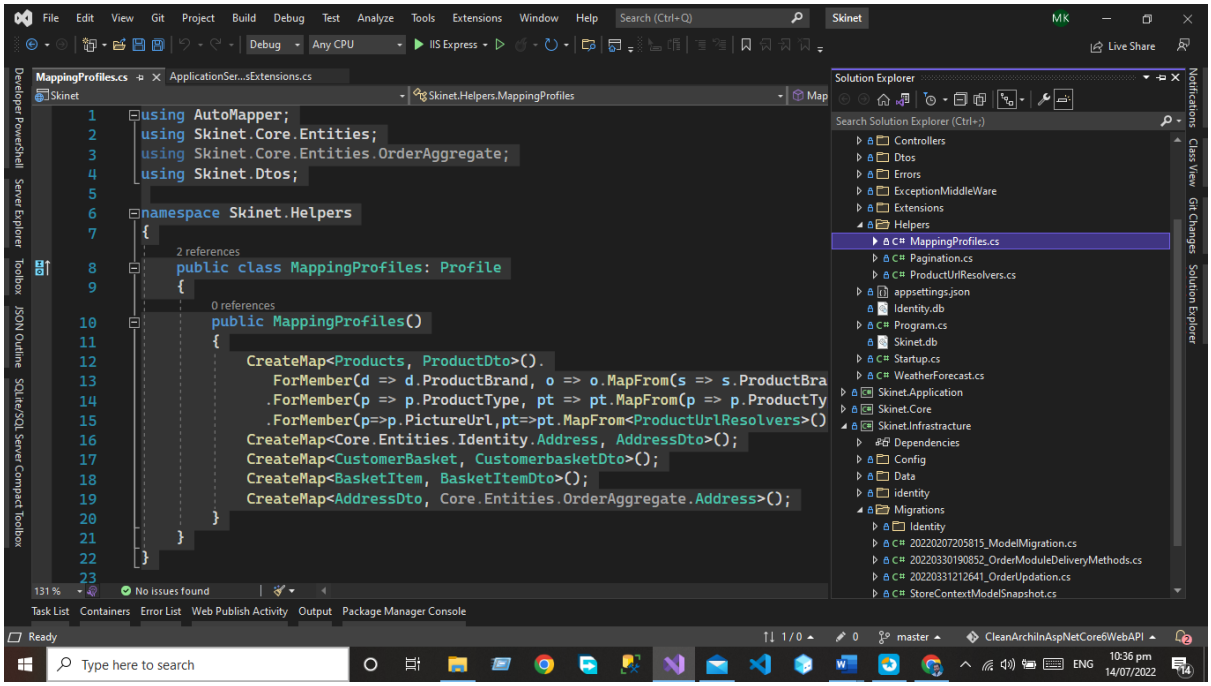


Helpers

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>();
        }
    }
}
```



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

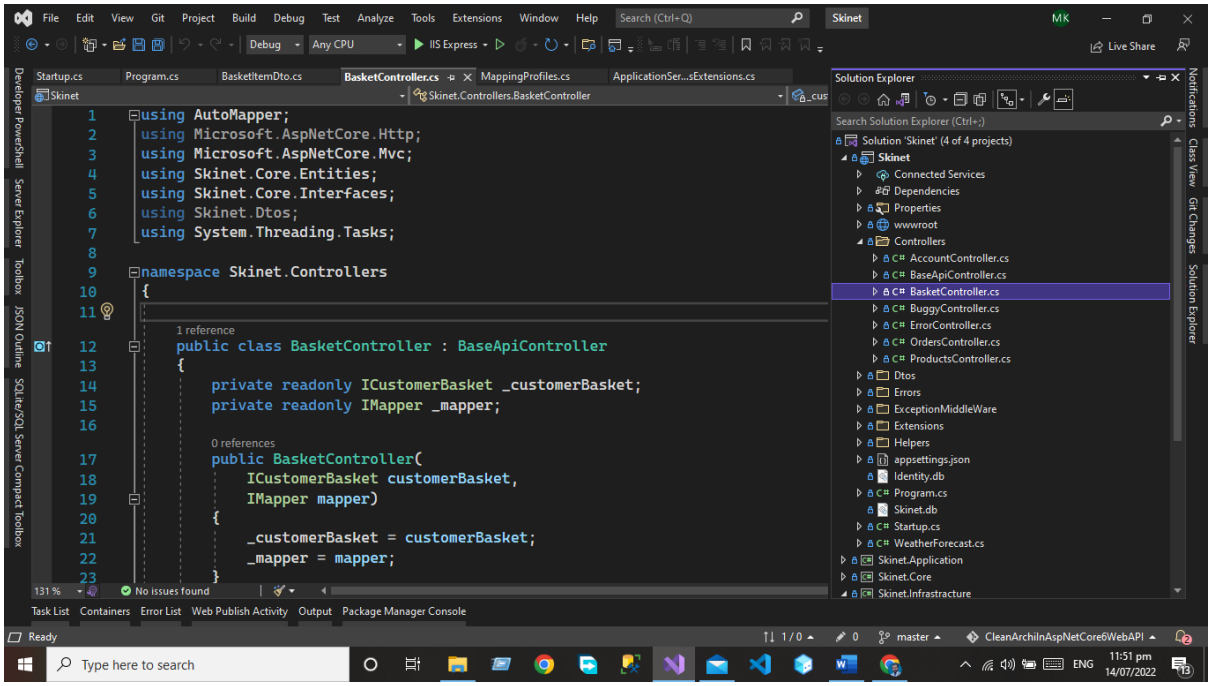
Lets us Create the Basket DTOs for clearing the concept about

```
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 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.

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



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

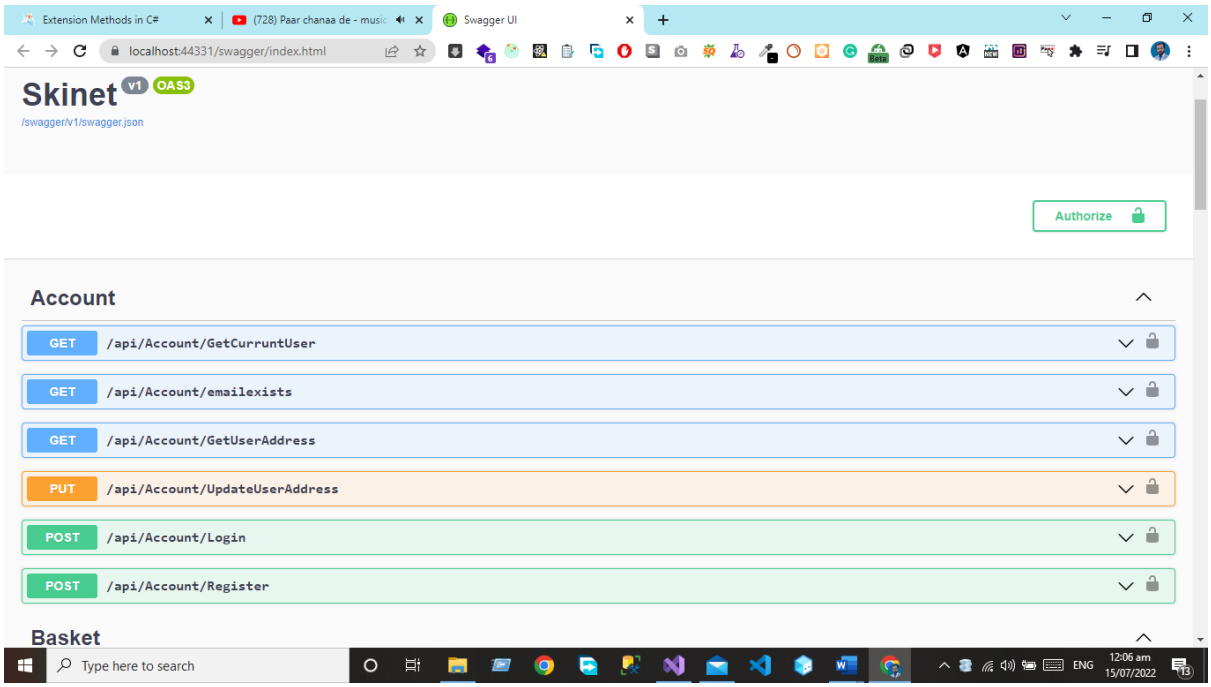
Code of the Product Controller

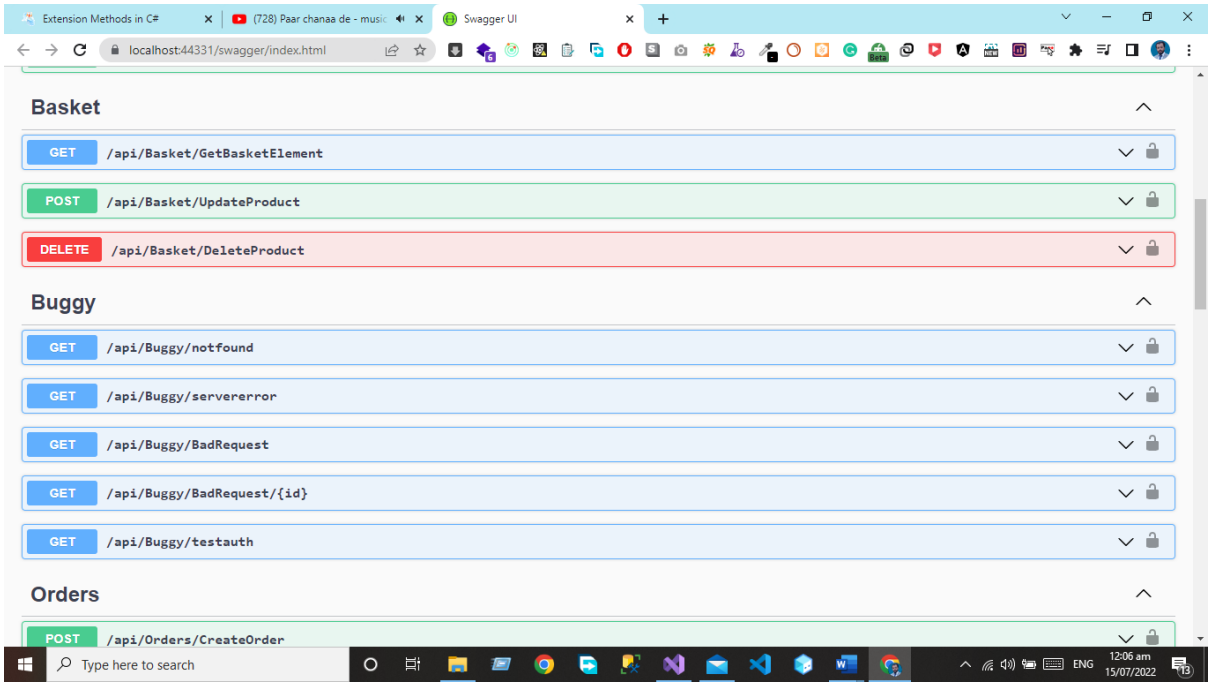
```
using AutoMapper;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Skinet.Core.Entities;
using Skinet.Core.Interfaces;
using Skinet.Core.Specifications;
using Skinet.Dtos;
using Skinet.Errors;
using Skinet.Helpers;
using System.Collections.Generic;
using System.Threading.Tasks;

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);
```

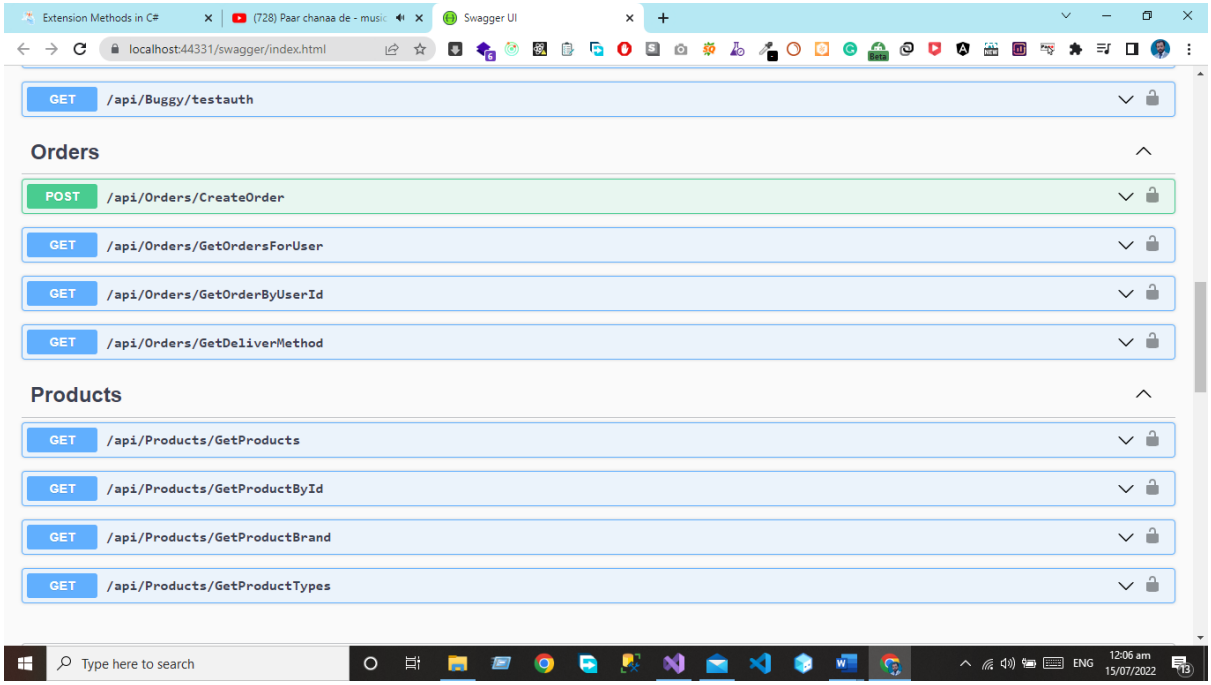
```
            return Ok(new Pagination<ProductDto>(productSpecPrams.pageIndex,
productSpecPrams.pageSize, total, data));
        }

        [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

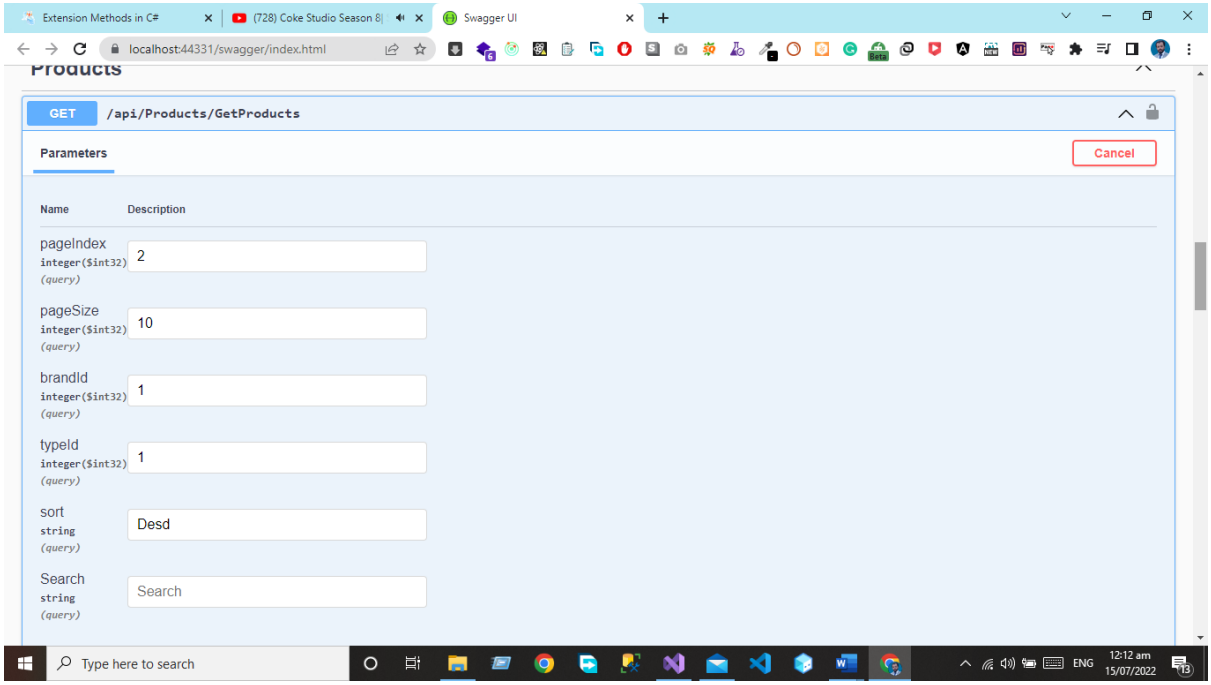
Now we will run the project and will see the output using the swagger.



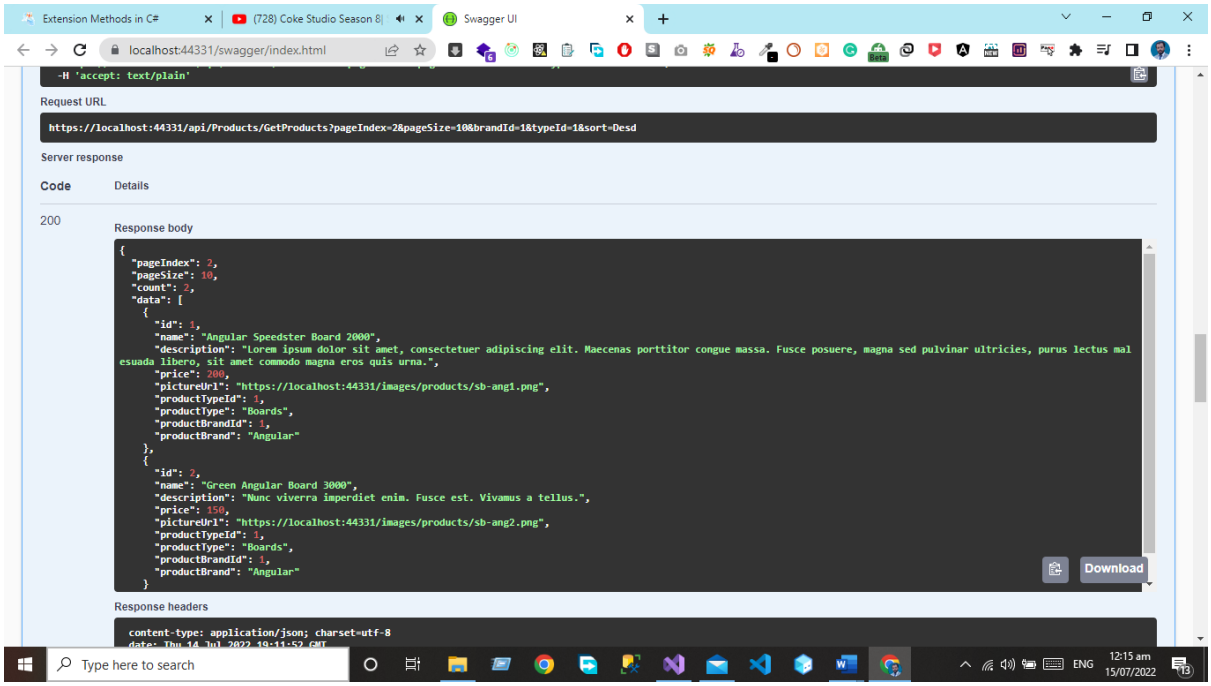

Extension Methods in C#
(728) Paar chanaa de - music
Swagger UI
localhost:44331/swagger/index.html
Basket
GET /api/Basket/GetBasketElement
POST /api/Basket/UpdateProduct
DELETE /api/Basket/DeleteProduct
Buggy
GET /api/Buggy/notfound
GET /api/Buggy/servererror
GET /api/Buggy/BadRequest
GET /api/Buggy/BadRequest/{id}
GET /api/Buggy/testauth
Orders
POST /api/Orders/CreateOrder
Type here to search
ENG
12:06 am
15/07/2022


Extension Methods in C#
(728) Paar chanaa de - music
Swagger UI
localhost:44331/swagger/index.html
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
Type here to search
ENG
12:06 am
15/07/2022

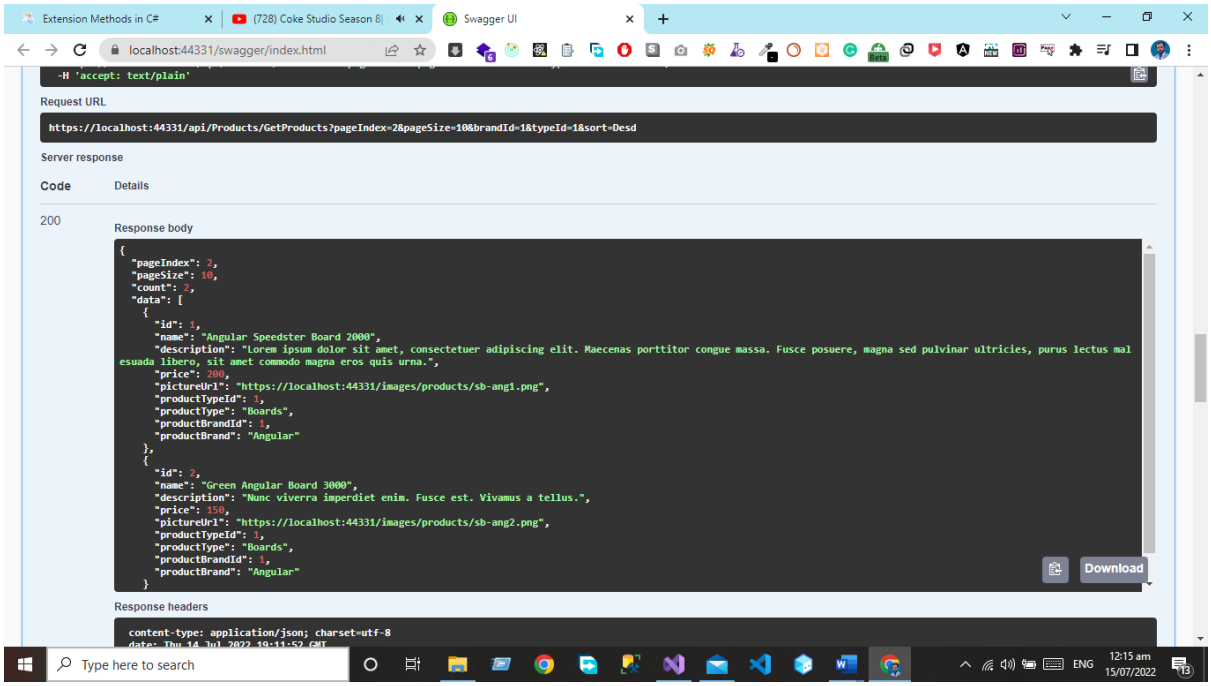
Let’s Get the Product using the Product Controller



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



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.



Conclusion

In this article, 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