Onion Architecture In .Net 5

Introduction

In this article, we are going to cover the Onion architecture in ASP.Net 5.0. As we all know, it's a newly launched framework officially released in the month of November. Here I am sharing the link to install the SDK for [.Net 5](https://dotnet.microsoft.com/download/dotnet/5.0)

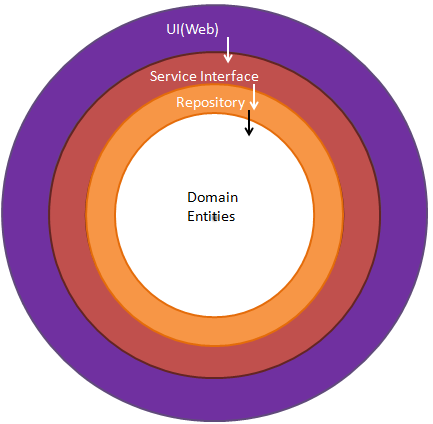
**What we are going to cover in this .NET 5 Onion Architecture?**

* What is Onion Architecture
* Layers of Onion Architecture
* Implementation of Onion Architecture
* Pros and Cons

What is Onion Architecture?

A large portion of the customary design raises basic issues of tight coupling and partition of concerns. Onion Architecture was acquainted by Jeffrey Palermo with giving a superior method to construct applications in the context of better testability, practicality, and constancy.

Onion Architecture tends to the difficulties confronted with 3-tier and n-tier architectures, and gives an answer for normal issues. Onion design layers associate with one another by utilizing the Interfaces.



Layers of Onion Architecture

Basically, it uses the concept of Layers but they are different from 3-tier and N-tier Layers. Let's deep dive into each of these layers.

**Domain Layer**

It exists at the center part of the Onion architecture where it consists of all application domain entities which are nothing but database models created by code first approach. In this project, I have used Fluent API in creating the table schema using Entity Framework

**Repository Layer**

The repository layer acts as a middle layer between the services and Model objects and in this layer, we will maintain all the Database migrations and application Data context object and in this layer, we typically add interfaces that will consist of data access pattern of read and write operations involving a database.

**Services Layer**

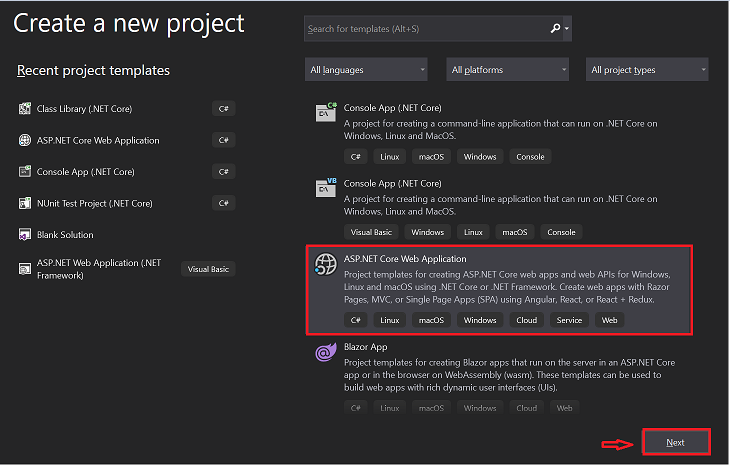
This layer is used to communicate between the Repository layer and Main Project where it consists of exposable API's. The Service layer also could hold business logic for an entity. In this layer, the service interfaces are kept separate from their implementation for loose coupling and also the separation of concerns.

**UI Layer**

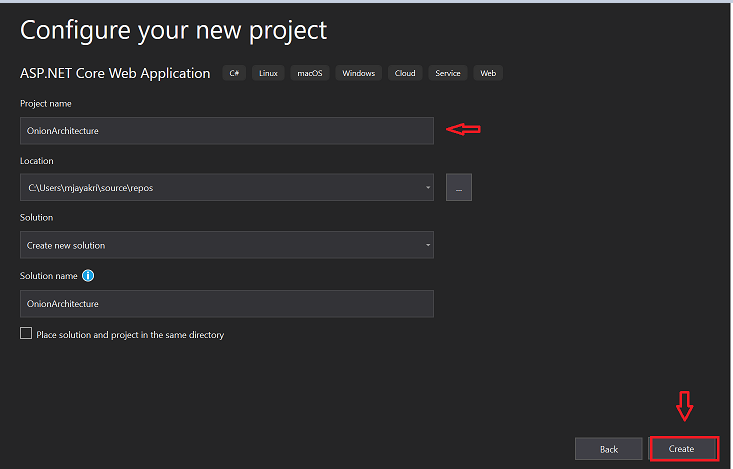
The UI is nothing but a front end application that will communicate with this API.

Implementation of Onion Architecture

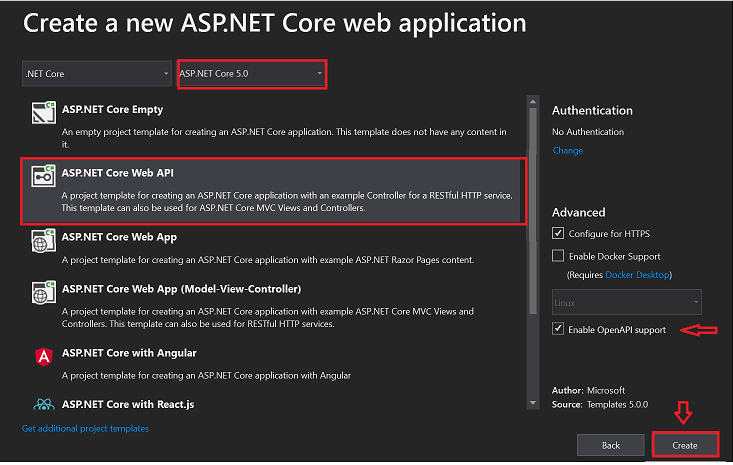
Create a New project



After clicking on the Next button add the project name and solution name and click on create button

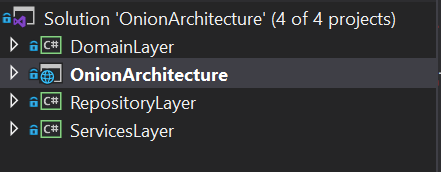


Choose the ASP.Net Core 5.0 template in the drop-down and also make sure to check the **Enable Open API** Support for default Swagger implementation in your project



Default project will be created and now we need 3 empty **Class library(.Net Core)**projects inside this application as

* DomainLayer
* RepositoryLayer
* ServicesLayer



We will start first with Domain Layer

Domain Layer

**Packages used in this Layer**

* Microsoft.EntityFrameworkCore(5.0.3)
* Microsoft.EntityFrameworkCore.Relational(5.0.3)

Create a folder named **Models**and inside that create Customer Class and BaseEntity class where customer class invokes this base entity

**BaseEntity.cs**

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Text;
5. **namespace** DomainLayer.Models
6. {
7. **public** **class** BaseEntity
8. {

**public** **int** Id { **get**; **set**; }

**public** DateTime CreatedDate { **get**; **set**; }

**public** DateTime ModifiedDate { **get**; **set**; }

**public** **bool** IsActive { **get**; **set**; }

1. }
2. }

**Customer.cs**

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Text;
5. **namespace** DomainLayer.Models
6. {
7. **public** **class** Customer : BaseEntity
8. {

**public** **string** CustomerName { **get**; **set**; }

**public** **string** PurchasesProduct { **get**; **set**; }

**public** **string** PaymentType { **get**; **set**; }

1. }
2. }

So, now we will create the actual table creation with this customer & base entity class. For that create a separate folder, EntityMapper, where we will maintain all our table schemas inside this folder

**CustomerMap.cs**

**using** DomainLayer.Models;

**using** Microsoft.EntityFrameworkCore;

**using** Microsoft.EntityFrameworkCore.Metadata.Builders;

**namespace** DomainLayer.EntityMapper

{

**public** **class** CustomerMap : IEntityTypeConfiguration<Customer>

    {

**public** **void** Configure(EntityTypeBuilder<Customer> builder)

        {

            builder.HasKey(x => x.Id)

                .HasName("pk\_customerid");

            builder.Property(x => x.Id).ValueGeneratedOnAdd()

                .HasColumnName("id")

                   .HasColumnType("INT");

            builder.Property(x => x.PurchasesProduct)

                .HasColumnName("purchased\_product")

                   .HasColumnType("NVARCHAR(100)")

                   .IsRequired();

            builder.Property(x => x.PaymentType)

              .HasColumnName("payment\_type")

                 .HasColumnType("NVARCHAR(50)")

                 .IsRequired();

            builder.Property(x => x.CreatedDate)

              .HasColumnName("created\_date")

                 .HasColumnType("datetime");

            builder.Property(x => x.ModifiedDate)

              .HasColumnName("modified\_date")

                 .HasColumnType("datetime");

            builder.Property(x => x.IsActive)

              .HasColumnName("is\_active")

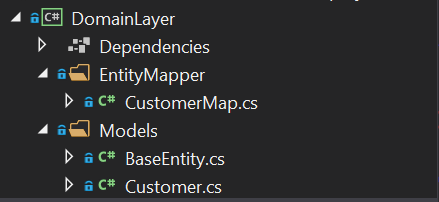
                 .HasColumnType("bit");

        }

    }

}

Domain Layer Structure



Repository Layer

**Packages used in this Layer**

* Microsoft.EntityFrameworkCore(5.0.3)
* Microsoft.EntityFrameworkCore.Design(5.0.3)
* Microsoft.EntityFrameworkCore.SqlServer(5.0.3)
* Microsoft.EntityFrameworkCore.Tools(5.0.3)

So far we designed our table in the Domain layer now will create the same table using migration commands in SQL DB. let's create the connection string in our main project.

**appsettings.json**

1. {
2. "Logging": {
3. "LogLevel": {
4. "Default": "Information",
5. "Microsoft": "Warning",
6. "Microsoft.Hosting.Lifetime": "Information"
7. }
8. },
9. "AllowedHosts": "\*",
10. "ConnectionStrings": {
11. "myconn": "server=**YOUR Server Name**; database=onionarcDb;Trusted\_Connection=True;"
12. }
13. }

 Setup the connection in startup.cs file under the ConfigureMethod

**Startup.cs**

1. **public** **void** ConfigureServices(IServiceCollection services)
2. {
4. services.AddControllers();

            services.AddSwaggerGen(c =>

            {

                c.SwaggerDoc("v1", **new** OpenApiInfo { Title = "OnionArchitecture", Version = "v1" });

            });

2. #region Connection String

            services.AddDbContext<ApplicationDbContext>(item => item.UseSqlServer(Configuration.GetConnectionString("myconn")));

1. #endregion
3. }

 Now switch back to Repository Layer and create a DataContext file where it represents a session with the database and can be used to query and save instances of your entities

**ApplicationDbContext.cs**

**using** DomainLayer.EntityMapper;

**using** DomainLayer.Models;

**using** Microsoft.EntityFrameworkCore;

**using** System;

**namespace** RepositoryLayer

{

**public** partial **class** ApplicationDbContext : DbContext

    {

**public** ApplicationDbContext(DbContextOptions options) : **base**(options)

        {

        }

**protected** **override** **void** OnModelCreating(ModelBuilder modelBuilder)

        {

            modelBuilder.ApplyConfiguration(**new** CustomerMap());

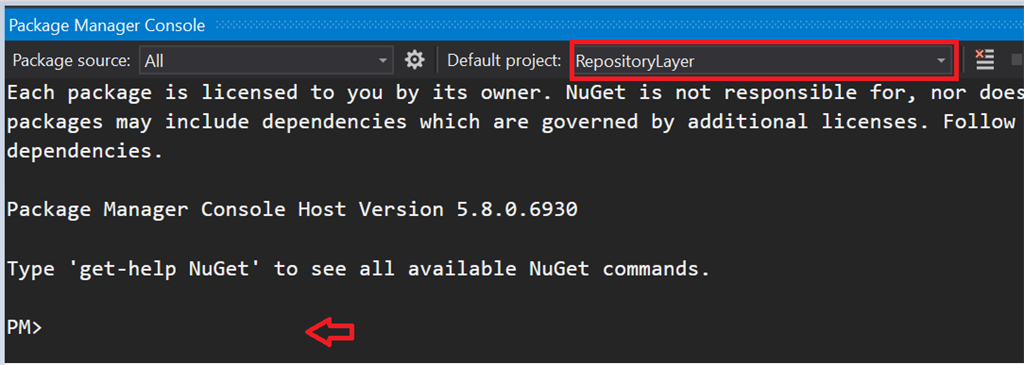
**base**.OnModelCreating(modelBuilder);

        }

    }

}

Let's create the table in SQL using the migration commands. Open the package manager console and switch the default project to Repositorylayer and execute the below commands one after another.



 Commands to execute

* Add-Migration 'CreateCustomerTable'
* Update-database

ASP.Net Core is designed in such a way to support dependency injection. Now we create a generic repository interface for the entity operations so that we can see the loosely coupled application. Below is the code snippet

**IRepository.cs**

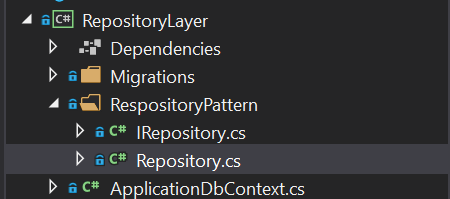
1. **using** DomainLayer.Models;
2. **using** System;
3. **using** System.Collections.Generic;
4. **using** System.Text;
6. **namespace** RepositoryLayer.RespositoryPattern
7. {
8. **public** **interface** IRepository<T> where T : BaseEntity
9. {
10. IEnumerable<T> GetAll();
11. T Get(**int** Id);
12. **void** Insert(T entity);
13. **void** Update(T entity);
14. **void** Delete(T entity);
15. **void** Remove(T entity);
16. **void** SaveChanges();
17. }
18. }

Create the repository class to perform the database operations which inherit the IRepository interface.

**Repository.cs**

1. **using** DomainLayer.Models;
2. **using** Microsoft.EntityFrameworkCore;
3. **using** System;
4. **using** System.Collections.Generic;
5. **using** System.Linq;
6. **using** System.Text;
8. **namespace** RepositoryLayer.RespositoryPattern
9. {
10. **public** **class** Repository<T> : IRepository<T> where T: BaseEntity
11. {
12. #region property
13. **private** **readonly** ApplicationDbContext \_applicationDbContext;
14. **private** DbSet<T> entities;
15. #endregion
17. #region Constructor
18. **public** Repository(ApplicationDbContext applicationDbContext)
19. {
20. \_applicationDbContext = applicationDbContext;
21. entities = \_applicationDbContext.Set<T>();
22. }
23. #endregion
25. **public** **void** Delete(T entity)
26. {
27. **if** (entity == **null**)
28. {
29. **throw** **new** ArgumentNullException("entity");
30. }
31. entities.Remove(entity);
32. \_applicationDbContext.SaveChanges();
33. }
35. **public** T Get(**int** Id)
36. {
37. **return** entities.SingleOrDefault(c => c.Id == Id);
38. }
40. **public** IEnumerable<T> GetAll()
41. {
42. **return** entities.AsEnumerable();
43. }
45. **public** **void** Insert(T entity)
46. {
47. **if** (entity == **null**)
48. {
49. **throw** **new** ArgumentNullException("entity");
50. }
51. entities.Add(entity);
52. \_applicationDbContext.SaveChanges();
53. }
55. **public** **void** Remove(T entity)
56. {
57. **if** (entity == **null**)
58. {
59. **throw** **new** ArgumentNullException("entity");
60. }
61. entities.Remove(entity);
62. }
64. **public** **void** SaveChanges()
65. {
66. \_applicationDbContext.SaveChanges();
67. }
69. **public** **void** Update(T entity)
70. {
71. **if** (entity == **null**)
72. {
73. **throw** **new** ArgumentNullException("entity");
74. }
75. entities.Update(entity);
76. \_applicationDbContext.SaveChanges();
77. }
79. }
80. }

**Repository Layer Structure**



Service Layer

This contains the Core Business Logic as part of our project which acts as a layer between the Repositorylayer and Controller.

**ICustomerService.cs**

1. **using** DomainLayer.Models;
2. **using** System.Collections.Generic;
4. **namespace** ServicesLayer.CustomerService
5. {
6. **public** **interface** ICustomerService
7. {
8. IEnumerable<Customer> GetAllCustomers();
9. Customer GetCustomer(**int** id);
10. **void** InsertCustomer(Customer customer);
11. **void** UpdateCustomer(Customer customer);
12. **void** DeleteCustomer(**int** id);
13. }
14. }

**CustomerService.cs**

1. **using** DomainLayer.Models;
2. **using** RepositoryLayer.RespositoryPattern;
3. **using** System;
4. **using** System.Collections.Generic;
5. **using** System.Text;
7. **namespace** ServicesLayer.CustomerService
8. {
9. **public** **class** CustomerService : ICustomerService
10. {
11. #region Property
12. **private** IRepository<Customer> \_repository;
13. #endregion
15. #region Constructor
16. **public** CustomerService(IRepository<Customer> repository)
17. {
18. \_repository = repository;
19. }
20. #endregion
22. **public** IEnumerable<Customer> GetAllCustomers()
23. {
24. **return** \_repository.GetAll();
25. }
27. **public** Customer GetCustomer(**int** id)
28. {
29. **return** \_repository.Get(id);
30. }
32. **public** **void** InsertCustomer(Customer customer)
33. {
34. \_repository.Insert(customer);
35. }
36. **public** **void** UpdateCustomer(Customer customer)
37. {
38. \_repository.Update(customer);
39. }
41. **public** **void** DeleteCustomer(**int** id)
42. {
43. Customer customer = GetCustomer(id);
44. \_repository.Remove(customer);
45. \_repository.SaveChanges();
46. }
47. }
48. }

Configure these services in the startup.cs file

**Startup.cs**

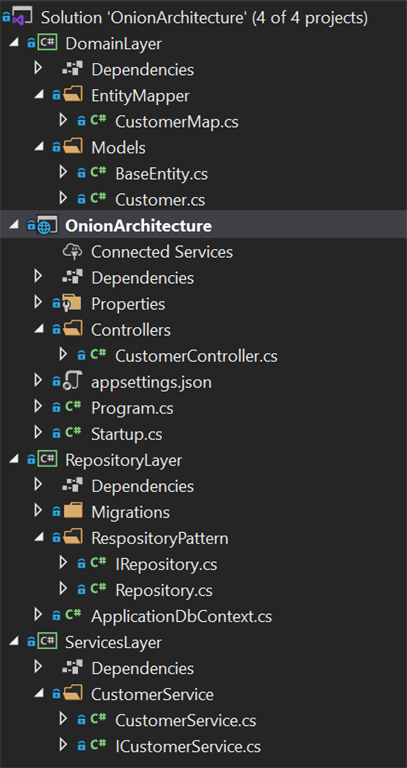
1. **using** Microsoft.AspNetCore.Builder;
2. **using** Microsoft.AspNetCore.Hosting;
3. **using** Microsoft.AspNetCore.HttpsPolicy;
4. **using** Microsoft.AspNetCore.Mvc;
5. **using** Microsoft.EntityFrameworkCore;
6. **using** Microsoft.Extensions.Configuration;
7. **using** Microsoft.Extensions.DependencyInjection;
8. **using** Microsoft.Extensions.Hosting;
9. **using** Microsoft.Extensions.Logging;
10. **using** Microsoft.OpenApi.Models;
11. **using** RepositoryLayer;
12. **using** RepositoryLayer.RespositoryPattern;
13. **using** ServicesLayer.CustomerService;
14. **namespace** OnionArchitecture
15. {
16. **public** **class** Startup
17. {
18. **public** Startup(IConfiguration configuration)
19. {
20. Configuration = configuration;
21. }
23. **public** IConfiguration Configuration { **get**; }
25. // This method gets called by the runtime. Use this method to add services to the container.
26. **public** **void** ConfigureServices(IServiceCollection services)
27. {
29. services.AddControllers();
30. services.AddSwaggerGen(c =>
31. {
32. c.SwaggerDoc("v1", **new** OpenApiInfo { Title = "OnionArchitecture", Version = "v1" });
33. });
35. #region Connection String
36. services.AddDbContext<ApplicationDbContext>(item => item.UseSqlServer(Configuration.GetConnectionString("myconn")));
37. #endregion
39. #region Services Injected
40. services.AddScoped(**typeof**(IRepository<>),**typeof**(Repository<>));
41. services.AddTransient<ICustomerService, CustomerService>();
42. #endregion
43. }
45. // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
46. **public** **void** Configure(IApplicationBuilder app, IWebHostEnvironment env)
47. {
48. **if** (env.IsDevelopment())
49. {
50. app.UseDeveloperExceptionPage();
51. app.UseSwagger();
52. app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json", "OnionArchitecture v1"));
53. }
55. app.UseHttpsRedirection();
57. app.UseRouting();
59. app.UseAuthorization();
61. app.UseEndpoints(endpoints =>
62. {
63. endpoints.MapControllers();
64. });
65. }
66. }
67. }

Create the API Methods in the customer controller which are exposable to UI (Front end app)

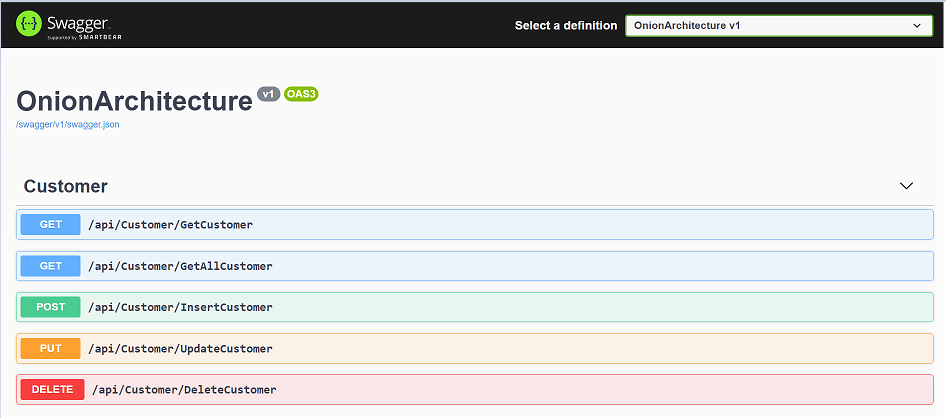
**CustomerController.cs**

1. **using** DomainLayer.Models;
2. **using** Microsoft.AspNetCore.Http;
3. **using** Microsoft.AspNetCore.Mvc;
4. **using** ServicesLayer;
5. **using** ServicesLayer.CustomerService;
6. **using** System;
7. **using** System.Collections.Generic;
8. **using** System.Linq;
9. **using** System.Threading.Tasks;
11. **namespace** OnionArchitecture.Controllers
12. {
13. [Route("api/[controller]")]
14. [ApiController]
15. **public** **class** CustomerController : ControllerBase
16. {
17. #region Property
18. **private** **readonly** ICustomerService \_customerService;
19. #endregion
21. #region Constructor
22. **public** CustomerController(ICustomerService customerService)
23. {
24. \_customerService = customerService;
25. }
26. #endregion
28. [HttpGet(nameof(GetCustomer))]
29. **public** IActionResult GetCustomer(**int** id)
30. {
31. var result = \_customerService.GetCustomer(id);
32. **if**(result **is** not **null**)
33. {
34. **return** Ok(result);
35. }
36. **return** BadRequest("No records found");
38. }
39. [HttpGet(nameof(GetAllCustomer))]
40. **public** IActionResult GetAllCustomer()
41. {
42. var result = \_customerService.GetAllCustomers();
43. **if** (result **is** not **null**)
44. {
45. **return** Ok(result);
46. }
47. **return** BadRequest("No records found");
49. }
50. [HttpPost(nameof(InsertCustomer))]
51. **public** IActionResult InsertCustomer(Customer customer)
52. {
53. \_customerService.InsertCustomer(customer);
54. **return** Ok("Data inserted");
56. }
57. [HttpPut(nameof(UpdateCustomer))]
58. **public** IActionResult UpdateCustomer(Customer customer)
59. {
60. \_customerService.UpdateCustomer(customer);
61. **return** Ok("Updation done");
63. }
64. [HttpDelete(nameof(DeleteCustomer))]
65. **public** IActionResult DeleteCustomer(**int** Id)
66. {
67. \_customerService.DeleteCustomer(Id);
68. **return** Ok("Data Deleted");
70. }
71. }
72. }

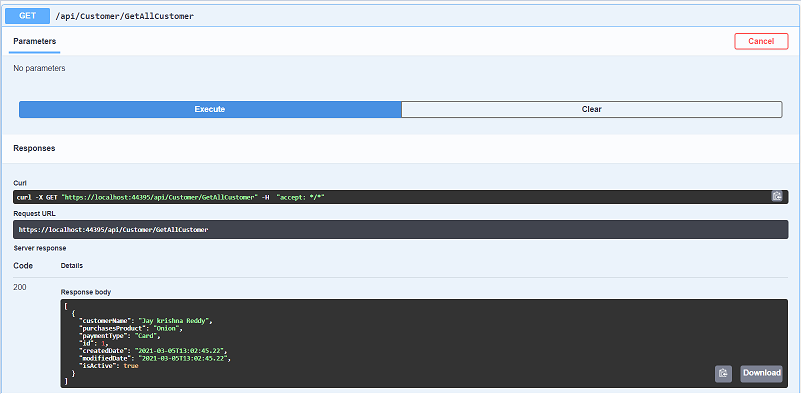
Onion Architecture Project Structure



Let's run and test this application to check the output in swagger or postman.



As I have already inserted one record in the database by using InserCustomer API wevwill see the data by executing Get ALL Customers API.



Pros and Cons in Onion Architecture

Following are the advantages of actualizing Onion Architecture:

* Onion Architecture layers are associated through interfaces. Implantations are given during run time.
* Application engineering is based on top of an area model.
* All outer reliance, similar to data set admittance and administration calls, are addressed in outside layers.
* No conditions of the Internal layer with outer layers.
* Couplings are towards the middle.
* Adaptable and feasible and convenient design.
* No compelling reason to make normal and shared activities.
* Can be immediately tried in light of the fact that the application center doesn't rely upon anything.

A couple of disadvantages of Onion Architecture as follows:

* Difficult to comprehend for amateurs, expectation to absorb information included. Modelers generally jumble up parting obligations between layers.
* Intensely utilized interfaces