Introduction To Clean Architecture And Implementation With ASP.NET Core

 Bishwanath Dey Nayan Updated date Aug 02, 2021

EmployeeManagementCleanArchitecture.rar

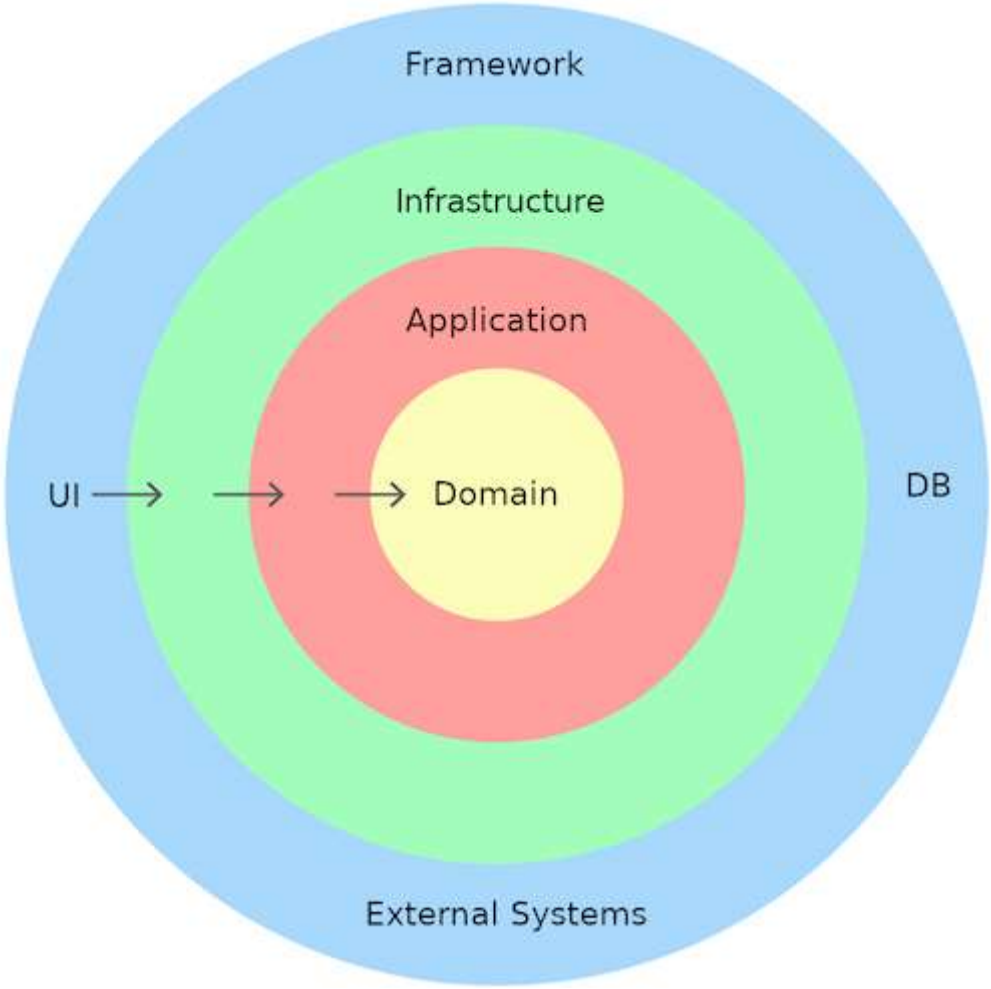
In this article, we will have an overview of Clean architecture and then we will try to implement this architecture with ASP.NET Core.

Let's get started,

Good architecture is a key to building **scalable**, **modular**, **maintainable** applications. Different architectures may vary in their details, but they all have the same objectives which are Separation of concern. And they all try to achieve this Separation by dividing the application into layers.

Clean Architecture is a software architecture intended to **keep the code under control** without all tidiness that spooks anyone from touching a code after the release. The main concept of Clean Architecture is the application code/logic which is very unlikely to change, has to be written without any direct dependencies. So it means that if I change my framework, database, or UI, the core of the system(Business Rules/ Domain) should not be changed. It means **external dependencies** are completely **replaceable**.

Overview



The domain layer contains **enterprise** logic and the **Application** layer contains **business** logic. Enterprise logic can be shared across many systems but the business logic will typically only be used within the system. The core will be independent of data access and other infrastructure concerns. And we can achieve this using interfaces and abstraction within the Core and implement them by other layers outside of the Core.

In Clean Architecture all dependencies flow **inwards** and **Core** has no dependency on any other layer. And Infrastructure and Presentation layer depends on Core.

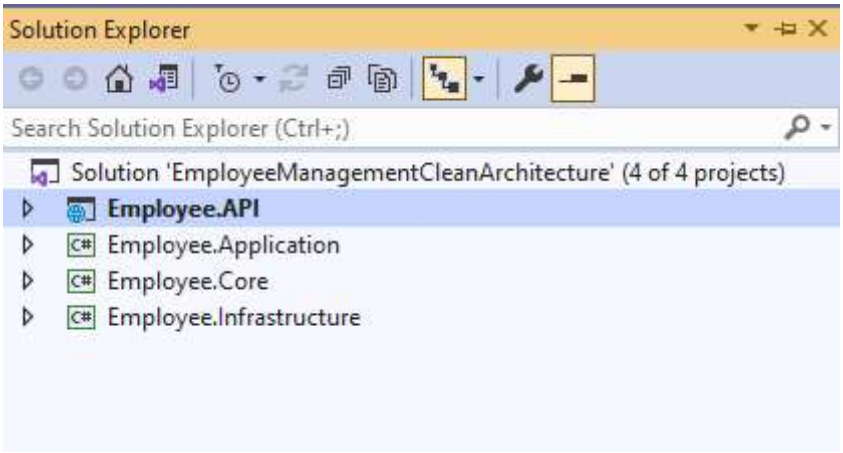
Benefits of Clean Architecture

- Independent of Database and Frameworks.
- Independent of the presentation layer. Anytime we can change the UI without changing the rest of the system and business logic.
- Highly testable, especially the core domain model and its business rules are extremely testable.

Implementing Clean Architecture with ASP.NET Core

Having said that, let's create a blank solution in Visual Studio.

After that let's add a Web API project and three class library projects inside the solution. So the project will look something like this.



Then, we will add project references. The key concept of Clean Architecture is that the inner layer should not know about the outer layer but the outer layer should know about the inner layer. So, Core will have no project reference. But, infrastructure will know about Core. And, API will know about all three layers.

Now let's go to the Core project and create two folders named "**Entities**" and "**Repositories**". And inside the "**Entities**" folder create a class named "**Employee.cs**".

```
1  public class Employee {
2      [Key]
3      [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
4      public Int64 EmployeeId {
5          get;
6          set;
7      }
8      public string FirstName {
9          get;
10         set;
11     }
12     public string LastName {
13         get;
14         set;
```

```
        get;
        set;
    }
    public string PhoneNumber {
        get;
        set;
    }
    public string Email {
        get;
        set;
    }
}
```

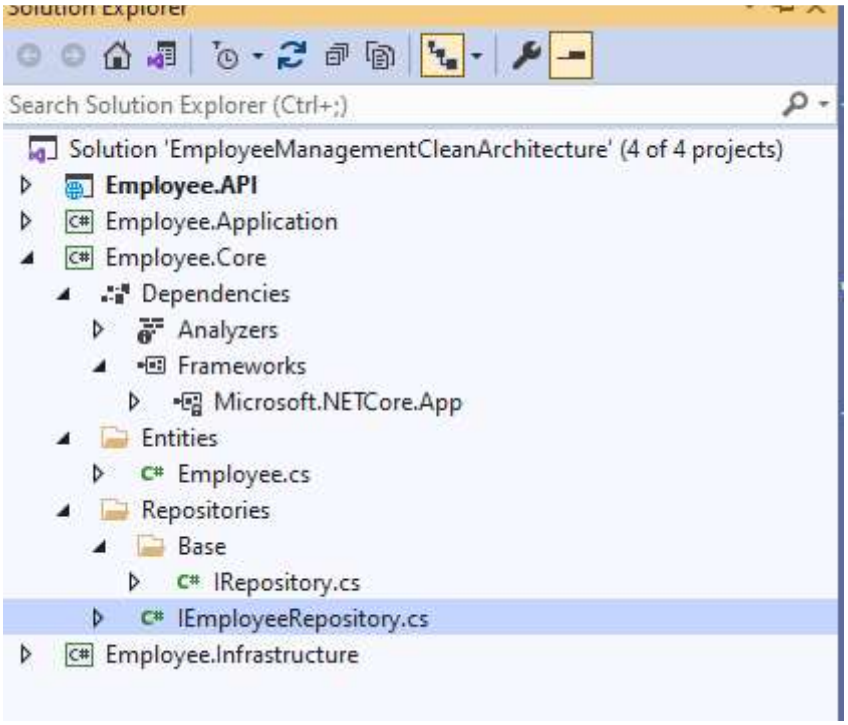
Now inside the "**Repositories**" folder, we will create a folder "**Base**". And inside the "**Base**" we will create an interface "**IRepository.cs**".

```
public interface IRepository < T > where T: class {
    Task < IReadOnlyList < T >> GetAllAsync();
    Task < T > GetByIdAsync(int id);
    Task < T > AddAsync(T entity);
    Task UpdateAsync(T entity);
    Task DeleteAsync(T entity);
}
```

This is a generic CRUD signature. We will create another interface inheriting IRepository which will have some custom method specific to an Entity inside the "**Repositories**" folder and will name it "**IEmployeeRepository.cs**".

```
public interface IEmployeeRepository: IRepository < Employee.Core.Entitie
    //custom operations here
    Task < IEnumerable < Employee.Core.Entities.Employee >> GetEmployeeBy
}
```

So the Core project structure will look like this,



Now we will implement the **infrastructure** layer. In the infrastructure project, we will add two folders "Data" and "Repositories". We will create **EmployeeContext** class in the "Data" folder.

EmployeeContext.cs

```
1  public class EmployeeContext: DbContext {
2      public EmployeeContext(DbContextOptions < EmployeeContext > options):
3      public DbSet < Employee.Core.Entities.Employee > Employees {
4          get;
5          set;
6      }
7  }
```

After that in the "**Repositories**" folder create a folder "**Base**" and inside it, we will create a **Repository** class that implements the interface for generic CRUD.

Repository.cs

```
1  public class Repository < T > : IRepository < T > where T: class {
2      protected readonly EmployeeContext _employeeContext;
3      public Repository(EmployeeContext employeeContext) {
4          _employeeContext = employeeContext;
5      }
```

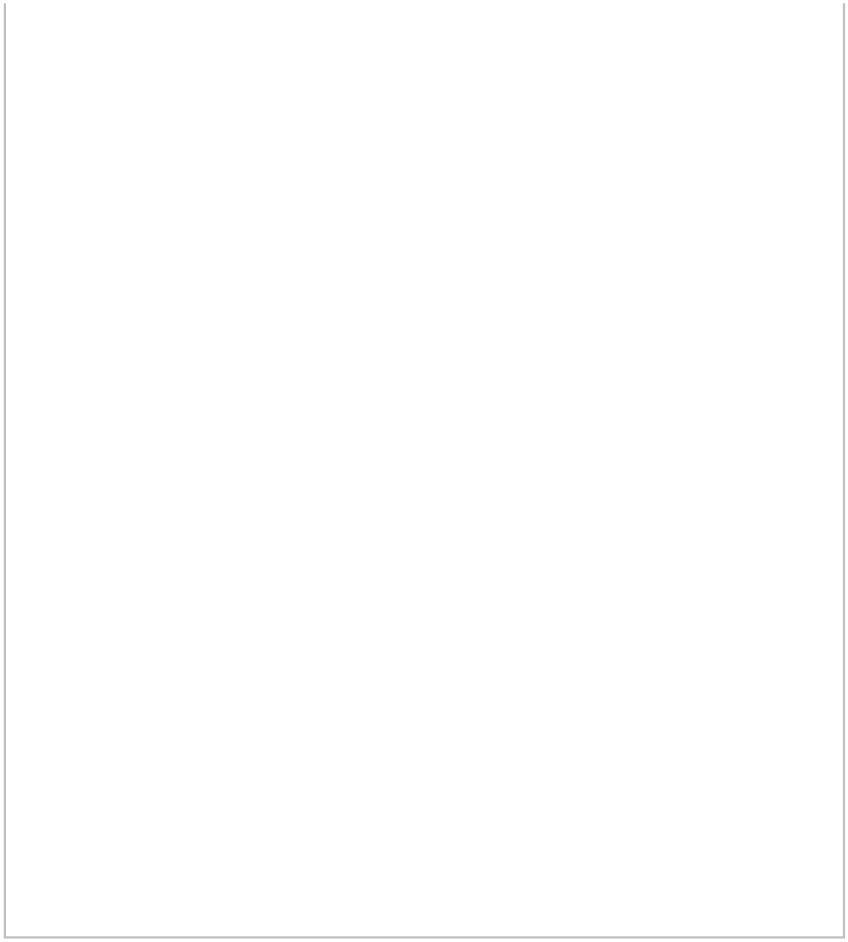
```
            await _employeeContext.SaveChangesAsync();
            return entity;
        }
        public async Task DeleteAsync(T entity) {
            _employeeContext.Set < T > ().Remove(entity);
            await _employeeContext.SaveChangesAsync();
        }
        public async Task < IReadOnlyList < T >> GetAllAsync() {
            return await _employeeContext.Set < T > ().ToListAsync();
        }
        public async Task < T > GetByIdAsync(int id) {
            return await _employeeContext.Set < T > ().FindAsync(id);
        }
        public Task UpdateAsync(T entity) {
            throw new NotImplementedException();
        }
    }
```

For a custom operation that is specific to an entity, we will create another class **EmployeeRepository** inside the **Repositories** folder. We will implement **Repository** and **IEmployeeRepositoy** which will have access to generic and custom code as well.

EmployeeRepository.cs

```
public class EmployeeRepository: Repository < Employee.Core.Entities.Empl
    public EmployeeRepository(EmployeeContext employeeContext): base(empl
    public async Task < IEnumerable < Core.Entities.Employee >> GetEmploy
        return await _employeeContext.Employees.Where(m => m.LastName ==
    }
}
```

Along with these changes, our Infrastructure layer looks like this.



Now we will go to the "**Application layer**". Here in the application layer, we will segregate queries from the command so we will implement CQRS here. Hence we will create some folders inside the **Application** Layer project. And the folders are,

- Commands
- Handlers
- Mappers
- Queries
- Responses

we will add our first command "CreateEmployeeCommand". So inside the **Commands** folder, we will add a new class "**CreateEmployeeCommand.cs**". This will issue a command to the infrastructure to create a new record. So this is going to implement **IRequest** and **MediatR** library

```
1  public class CreateEmployeeCommand: IRequest < EmployeeResponse > {
2      public string FirstName {
3          get;
4          set;
5      }
```

```
        set;
    }
    public DateTime DateOfBirth {
        get;
        set;
    }
    public string PhoneNumber {
        get;
        set;
    }
    public string Email {
        get;
        set;
    }
}
```

Here inside the IRequest<> we define the response type as EmplooyeeResponse. So we will create an **EmployeeResponse.cs** inside the **Responses** Folder.

```
public class EmployeeResponse {
    public int EmployeeId {
        get;
        set;
    }
    public string FirstName {
        get;
        set;
    }
    public string LastName {
        get;
        set;
    }
    public DateTime DateOfBirth {
        get;
        set;
    }
    public string PhoneNumber {
        get;
        set;
```

```
        get;
        set;
    }
}
```

Next, we will create a handler for CreateEmployeeCommand. We will be creating this in the **Handlers** folder. **CreateEmployeeHandler** will look like this.

```
public class CreateEmployeeHandler: IRequestHandler < CreateEmployeeComma
    private readonly IEmployeeRepository _employeeRepo;
    public CreateEmployeeHandler(IEmployeeRepository employeeRepository)
        _employeeRepo = employeeRepository;
    }
    public async Task < EmployeeResponse > Handle(CreateEmployeeCommand r
        var employeeEntitiy = EmployeeMapper.Mapper.Map < Employee.Core.E
        if (employeeEntitiy is null) {
            throw new ApplicationException("Issue with mapper");
        }
        var newEmployee = await _employeeRepo.AddAsync(employeeEntitiy);
        var employeeResponse = EmployeeMapper.Mapper.Map < EmployeeRespon
        return employeeResponse;
    }
}
```

Here we supplied IRequestHandler, which takes CreateEmployeeCommand and MovieResponse.

Line 11: We mapped Employee Entity with the request object.

Line 16: Added the employeeEntity to the database.

Line 17: Mapping back to MovieResponse to fetch EmployeeId from the object and after that returning the response.

Here all this Mapping stuffs is implemented with AtuoMapper. Hence, we will create **EmployeeMapper.cs** class inside the **Mapper** folder.

```
public class EmployeeMapper {
    private static readonly Lazy < IMapper > Lazy = new Lazy < IMapper >
        var config = new MapperConfiguration(cfg => {
```

```
6           });
7           var mapper = config.CreateMapper();
8           return mapper;
9       });
10      public static IMapper Mapper => Lazy.Value;
11  }
```

Now we need to create a Mapping profile named "**EmployeeMappingProfile**" inside the "**Mapper**" folder as we provided in **line 8**.

```
1  public class EmployeeMappingProfile: Profile {
2      public EmployeeMappingProfile() {
3          CreateMap < Employee.Core.Entities.Employee, EmployeeResponse > (
4          CreateMap < Employee.Core.Entities.Employee, CreateEmployeeCommar
5      }
6  }
```

So like the command, we can also create some queries in our project. But now I am skipping this. But we can find Query creating example in the attached source code in this article.

So after all these changes, our Application Layer looks like this.

Now we will dive inside API implementation. In the **API Project,** we will add dependencies like Automapper, swagger, MediatR in the **startup.cs** file.

```
public void ConfigureServices(IServiceCollection services) {
    services.AddControllers();
    services.AddDbContext < EmployeeContext > (m => m.UseSqlServer(Config
    services.AddSwaggerGen(c => {
        c.SwaggerDoc("v1", new OpenApiInfo {
            Title = "Employee.API", Version = "v1"
        });
    });
    services.AddAutoMapper(typeof(Startup));
    services.AddMediatR(typeof(CreateEmployeeHandler).GetTypeInfo().Assem
    services.AddScoped(typeof(IRepository < > ), typeof(Repository < > ))
    services.AddTransient < IEmployeeRepository, EmployeeRepository > ();
}
```

In the **appsettings.json** file, we will create connection strings.

```
{
  "ConnectionStrings": {
```

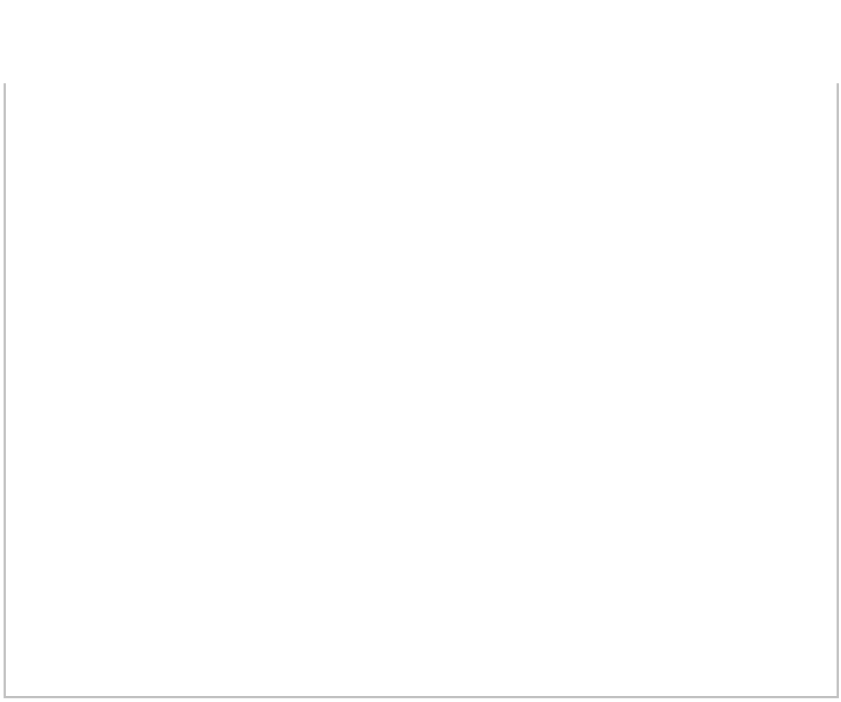
```
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

Later we will create a new controller **EmployeeController** as shown below,

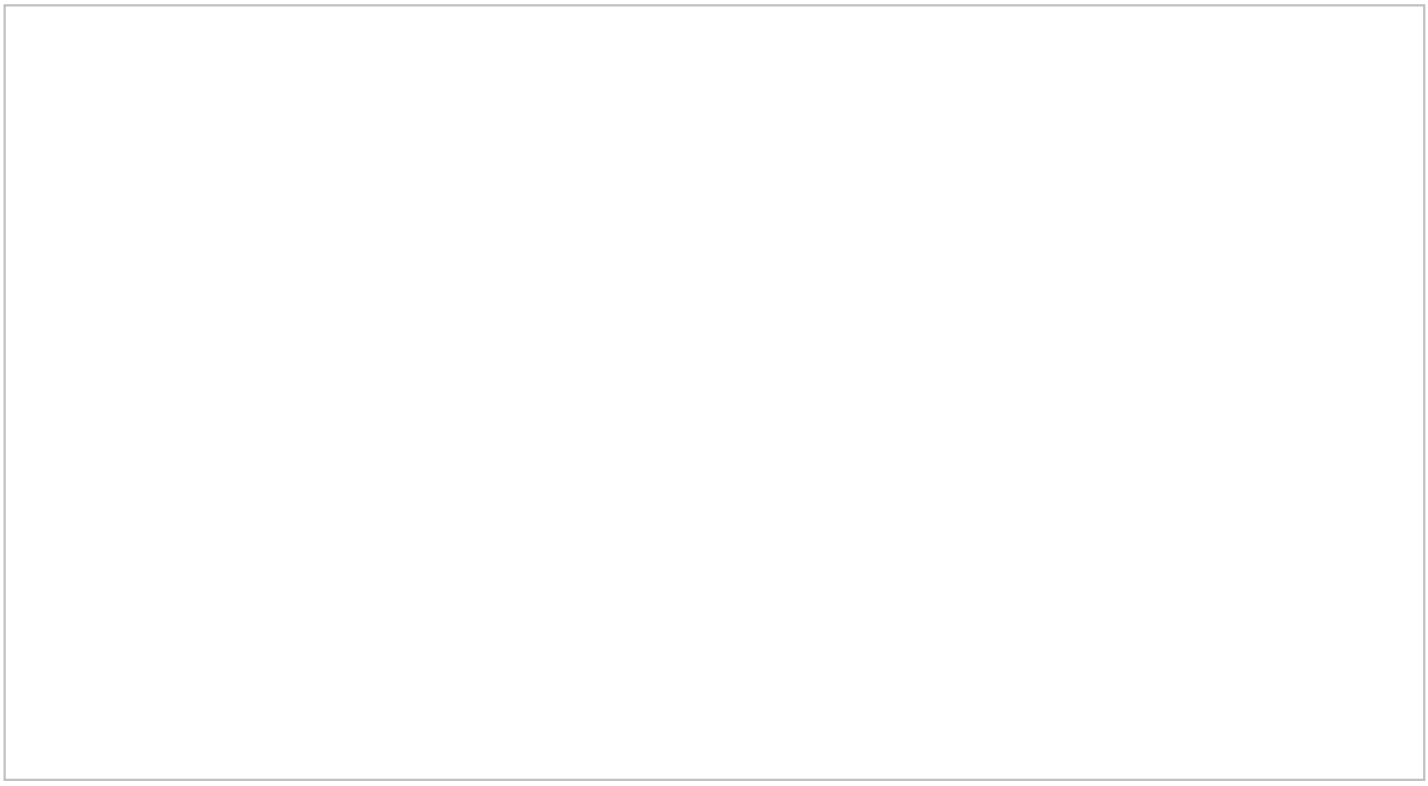
```
[Route("api/[controller]")]
    [ApiController]
    public class EmployeeController : ControllerBase
    {
        private readonly IMediator _mediator;
        public EmployeeController(IMediator mediator)
        {
            _mediator = mediator;
        }

        [HttpGet]
        [ProducesResponseType(StatusCodes.Status200OK)]
        public async Task<List<Employee.Core.Entities.Employee>> Get()
        {
            return await _mediator.Send(new GetAllEmployeeQuery());
        }
        [HttpPost]
        [ProducesResponseType(StatusCodes.Status200OK)]
        public async Task<ActionResult<EmployeeResponse>> CreateEmployee(
        {
            var result = await _mediator.Send(command);
            return Ok(result);
        }
    }
```

After all these changes our API project will look like this,



Now if we run our application it will redirect us to **localhost:port/swagger/index.html** and from there we can test our endpoint.



With this, we can conclude our Clean Architecture tour with ASP.NET Core.

Happy coding!!!

Clean Architecture | Clean Architecture with .NET Core | Clean Architecture with ASP.NET Core

Next Recommended Reading

jQuery PJAX Implementation In ASP.NET Core

Bishwanath Dey Nayan
Programmer by heart writing about his daily software adventures, wherever they may lead.
C# | .NET Core | React | Angular.

1218 **42.4k**

6 **3**

Type your comment here and press Enter Key (Minimum 10 characters)

Follow Comments

This is a good example and I appreciate the effort, however only caveat is API layer is having reference of Data Access Layer(Employee.Infrastructure).API/web layer should be totally decoupled with DAL. It should have zero reference of Entity Framework etc

Kashif Reza 3h 33m ago

1889 **170** **0** 0 0 Reply

Very nice articles!!!

Saravanakumar Sekaran 4h 12

Thank you Saravanakumar Sekaran.

Bishwanath Dey Nayan
1218 917 42.4k

3h 34m ago
1

FEATURED ARTICLES

What Is NFT? Everything You Need To Know About NFTs

Building Secure REST API

Unit Testing With xUnit And Moq In ASP.NET Core

Using Certificates For API Authentication In .NET 5

List Of Commonly Used GitHub Commands

View All

TRENDING UP

01 What's New In Java 16?

02 Azure Function - An Serverless Architecture

03 Data Science - A Skill Needed For Future

04 Creating And Training Custom ML Model to Read Sales Receipts Using AI-Powered Azure Form Recognizer

05 Introduction To Clean Architecture And Implementation With ASP.NET Core

06 Create And Validate JWT Token In .NET 5.0

07 RabbitMQ Designs

08 .NET Core NUGET packages

09 Distributed Caching (Redis)

10 A Diagnosis Of Parallel.Foreach

View All