عنوان مقاله: معماری تمیز (Clean Architecture) چیست ؟ ۵ مرحله راه اندازی آن

نویسنده مقاله: تیم فنی نیک آموز

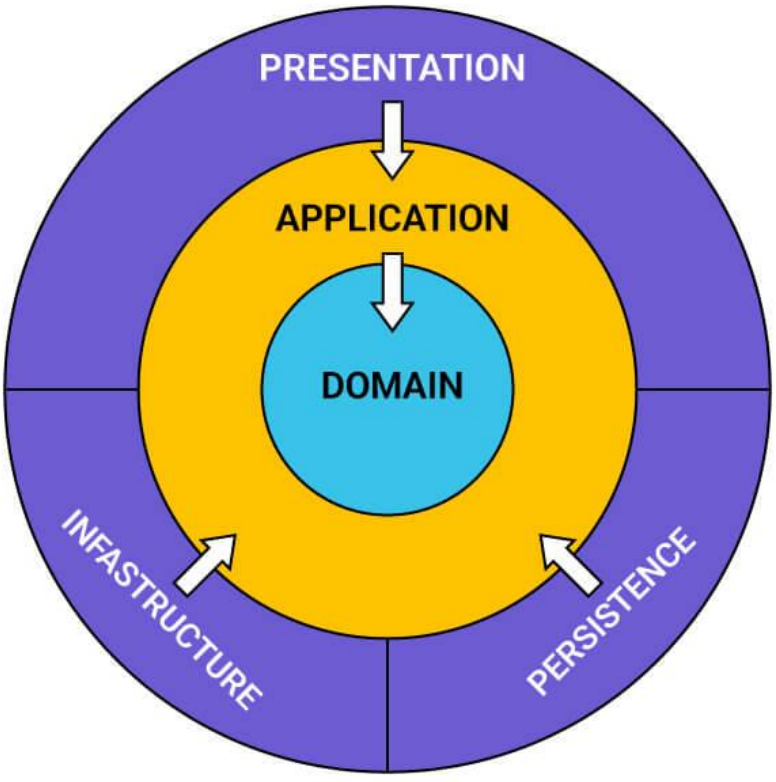
تاریخ انتشار: ۹ آذر ۱۴۰۲

منبع: https://nikamooz.com/what-is-clean-architecture/

راه اندازی Clean Architecture و پیاده سازی DDD ، دو اقدام اساسی هستند که با کمک آن‌ها، یک سیستم ساختارمند و قدرتمند حاصل می‌شود. در بخش اول این مقاله، به بررسی نحوه راه اندازی Clean Architecture و لایه‌های مختلف آن می‌پردازیم و پس از آن، به سراغ پیاده‌سازی Domain Driven Design می‌رویم. درنهایت، به شما روشی را معرفی می‌کنیم که با کمک آن می‌توانید از صحت معماری پیاده‌سازی‌شده، مطمئن شوید.

Clean Architecture چیست ؟

معماری تمیز (Clean Architecture) ، یک معماری محبوب برای سازماندهی اپلیکیشن‌ها محسوب می‌شود. این معماری طرفداران و منتقدان خود را دارا است؛ اما درنهایت، این رویکرد، یکی از معماری‌های مناسب برای پروژه‌های بزرگ و سطح سازمانی تلقی می‌شود. در این مطلب، یک پروژه براساس اصول و لایه‌بندی‌های Clean Architecture از ابتدا تا انتها ایجاد خواهیم کرد تا شما با نحوه راه اندازی Clean Architecture آشنا شوید.

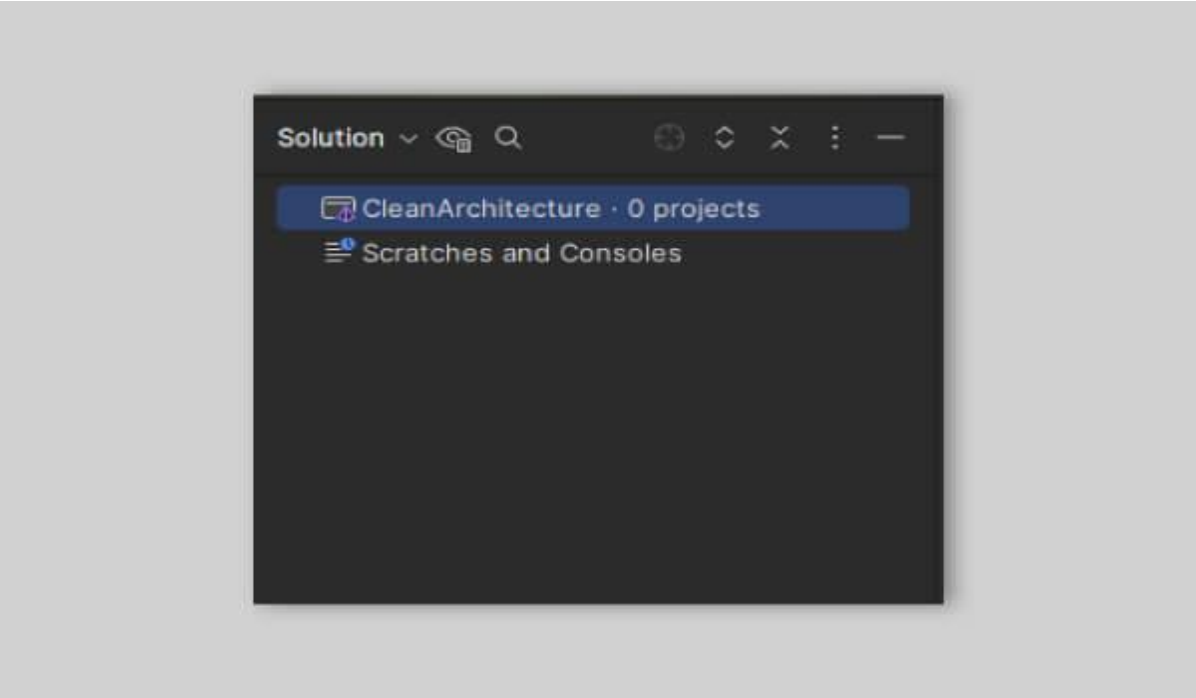


مراحل راه اندازی Clean Architecture

منظور از راه اندازی Clean Architecture به معنای واقعی کلمه این است که یک Solution خالی در Visual Studio شروع کرده و به سمت ساختار کامل Clean Architecture پیش بروید.

۱- ایجاد پوشه حاوی پروژه‌ها

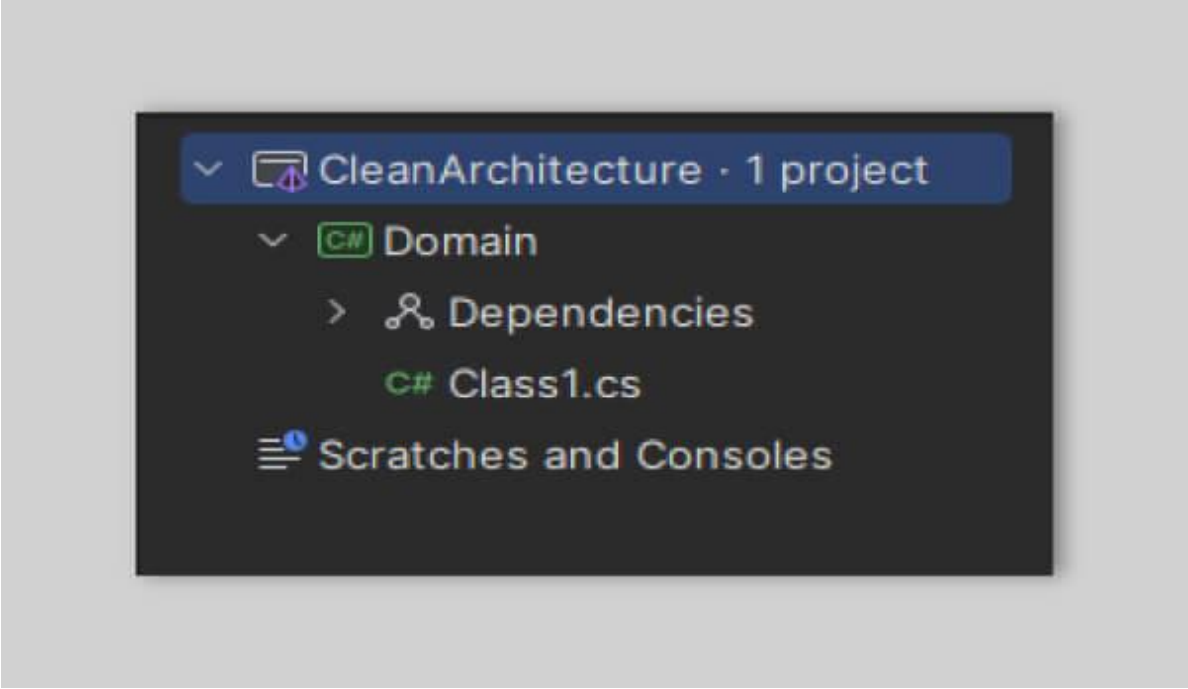
برای آغاز راه اندازی Clean Architecture ، ابتدا باید یک پوشه Solution خالی ایجاد کنید که درنهایت حاوی همه پروژه‌های آتی خواهد بود.



۲- ایجاد لایه Domain

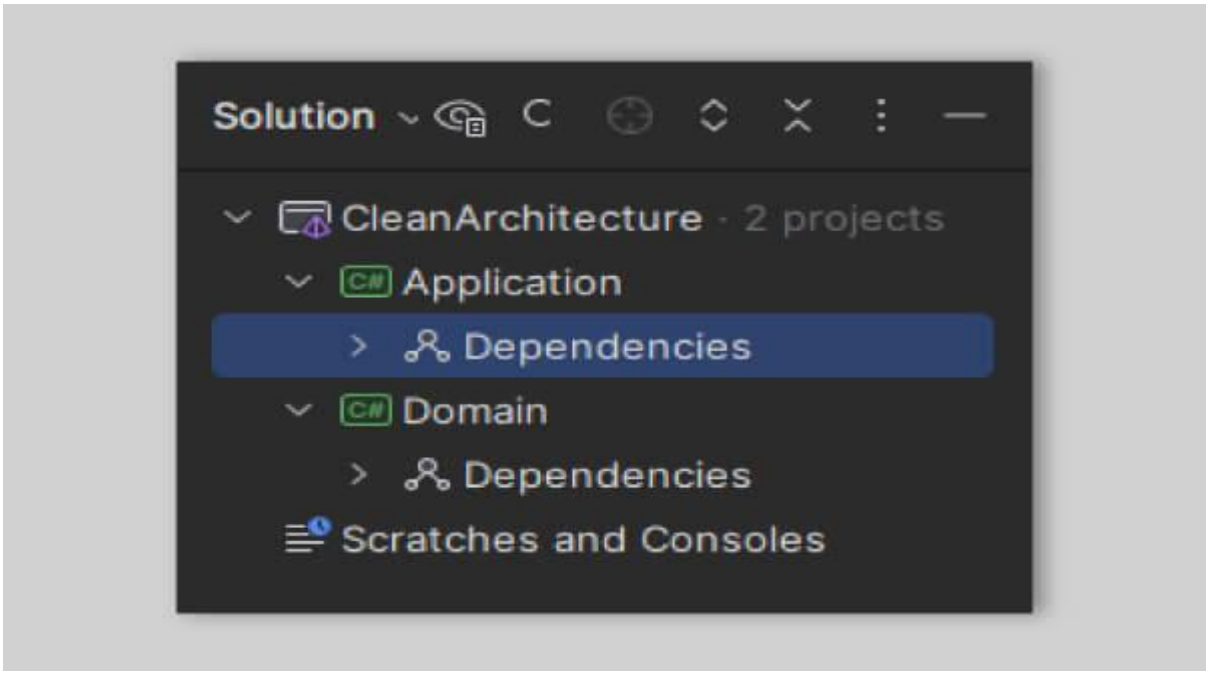
شما از هسته معماری Clean Domain نام دارد، روند راه اندازی Clean Architecture را شروع می‌کنید. نامی که برای این لایه درنظر می‌گیرید، می‌تواند Domain یا ترکیبی از نام پروژه و عبارت Domain باشد. شما باید به گونه‌ای لایه‌ها را نام‌گذاری کنید که با نگاهی سریع متوجه کارکرد آن لایه بشوید. داخل Solution ایجاد شده یک پروژه Dot net 7 از نوع Class Library ایجاد می‌کنیم. این پروژه، حاوی کلاسی به اسم Class1 هست که می‌توانیم آن را پاک کنیم.

آنچه معمولاً در پروژه Domain تعریف می‌کنید، قوانین اصلی مربوط به کسب و کار، Enumerations ، Value Object ها، Custom Exception و چنین مواردی است. توجه کنید که در این آموزش، تمام این موارد انجام نمی‌شوند؛ بلکه تنها روی Setup ساختار پروژه براساس Clean Architecture تمرکز خواهد شد.



۳- ساخت لایه Application

لایه بعدی که باید تعریف کنیم Application نام دارد. برای این کار، مجدداً یک پروژه Dot net 7 و از نوع Class Library لازم است. ضمن اینکه لازم است کلاس پیش‌فرض حذف شود. برای درک بهتر نتیجه، به تصویر زیر توجه کنید.



اساساً، لایه Domain مجوز ارجاع داشتن به هیچ کدام از لایه‌های بیرونی را ندارد و این موضوع، یک قانون مهم در معماری Clean محسوب می‌شود. در حالی که لایه Application ، امکان برقراری ارتباط با لایه Domain را دارد. در انتهای مقاله، روشی بررسی می‌شود که چنین قیدهایی را برای ما در نظر بگیرد.

پروژه Application یک Orchestrator از سایر لایه‌ها و Use Case ها تلقی می‌شود. این یعنی در این لایه، ماژول‌های مختلف فراخوانی و مورد استفاده قرار می‌گیرند و هیچ منطقی مرتبط با کسب و کار تعریف نخواهد شد. همچنین، در این لایه می‌توانید Service های گوناگون را فراخوانی و به‌کار ببرید.

معمولاً برای ارتباط بین لایه Entry Point و Application از mediator استفاده می‌شود. Mediator یک Design Pattern است که به‌واسطه آن، Couple-less بودن لایه‌های مختلف پروژه تضمین خواهد شد. اما چرا باید ماژول‌های مختلف به هم وابستگی نداشته باشند؟

فرض کنید روزی تصمیم گرفته شود تا یکی از لایه‌های پروژه، به سرویس دیگری از یک میکروسرویس بزرگ‌تر منتقل شود. در این شرایط، اگر وابستگی‌های زیادی بین دو لایه وجود داشته باشند، جداسازی آن‌ها دشوار است و دیگر نمی‌توانید یک لایه خاص را به میکروسرویس بزرگ‌تر منتقل کنید.

در .NET ، یک **NuGet Package** به اسم MediatR وجود دارد که در ادامه، آن را نصب و استفاده خواهیم کرد.

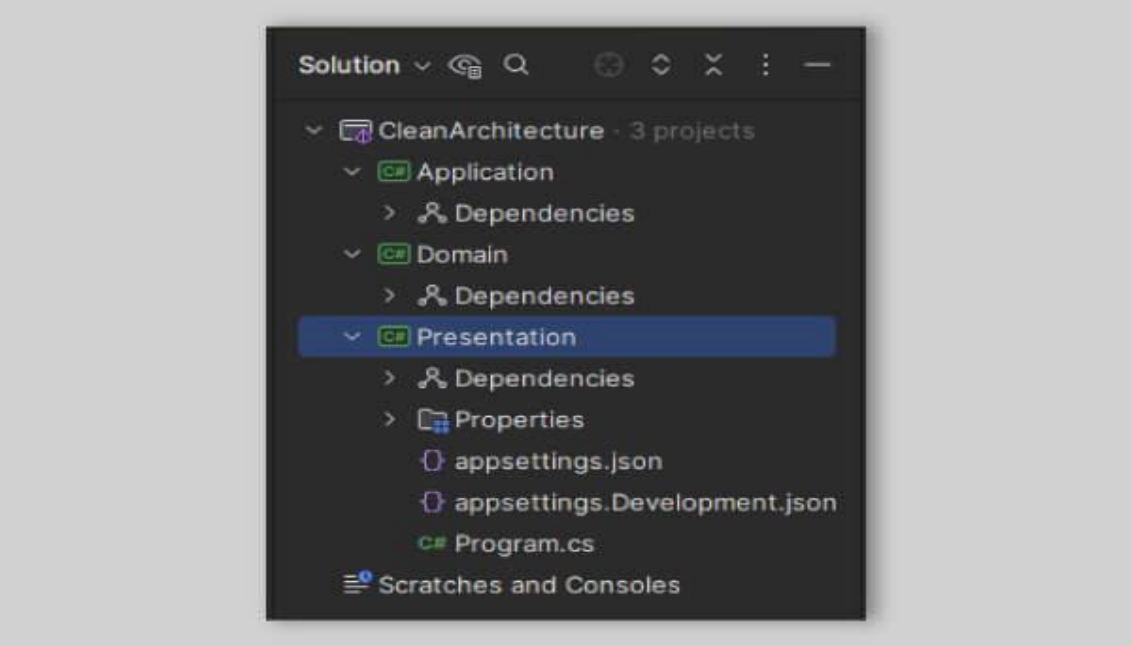
```
dotnet add package MediatR --version 12.1.1
```

پس از نصب MediatR، می‌توان Use Case ها را ایجاد کرد. لازم است هر Use Case، به‌عنوان یک کلاس مستقلی پیاده‌سازی شود که از MediatR.RequestHandler<TRequest ,TResponse> ارث‌بری می‌کند. پارامتر TRequest نشان‌دهنده شی درخواستی خاصی است که به Use Case ارسال و پارامتر TResponse نمایان‌گر شی پاسخی است که از Use Case برگردانده می‌شود.

به منظور درک کارایی این پکیج، یک Entry Point یا API ایجاد کرده و طبق آن، یک کاربر را ثبت‌نام خواهیم کرد. منظور از Entry Point یا نقطه ورودی پروژه، جایی است که درخواست‌ها در ابتدا به آن ارسال می‌شوند و سپس، از آن جا به سایر لایه‌ها هدایت خواهند شد. این ورودی می‌تواند یک WebApi یا یک Console Application باشد. در این آموزش، گزینه اول، یعنی WebApi را انتخاب می‌کنیم.

۴- ایجاد لایه Presentation

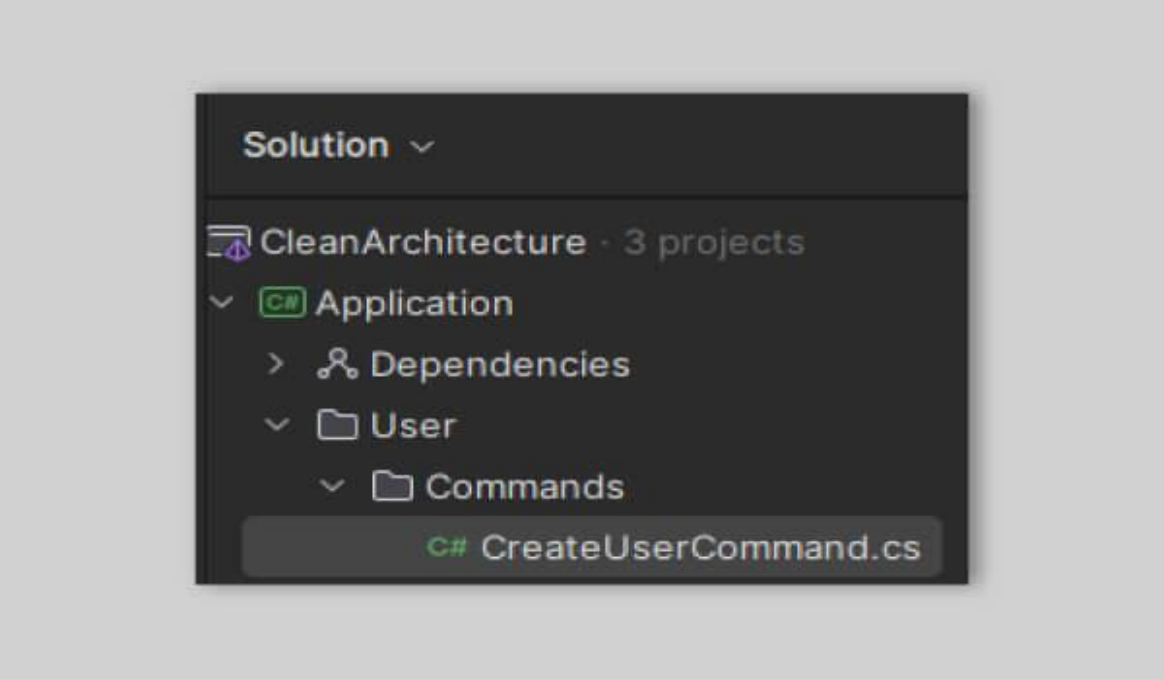
لازم است لایه جدیدی به نام Presentation از Net 7. و از نوع ASP.Net Core Web Application بسازید.



به منظور نوشتن یک API برای ثبت کاربر، کنترلر (Controller) نیاز است. برای انجام این کار، یک پوشه به نام Controllers بسازید و در داخل آن، کلاسی با نام UserController ایجاد کنید. در ادامه، یک Command ایجاد خواهیم کرد که با کمک آن، مشخصات کاربر جدید دریافت بشوند. سپس، یک کلاس Handler می‌سازیم که این درخواست را پردازش کرده و کاربر جدید را ایجاد کند.

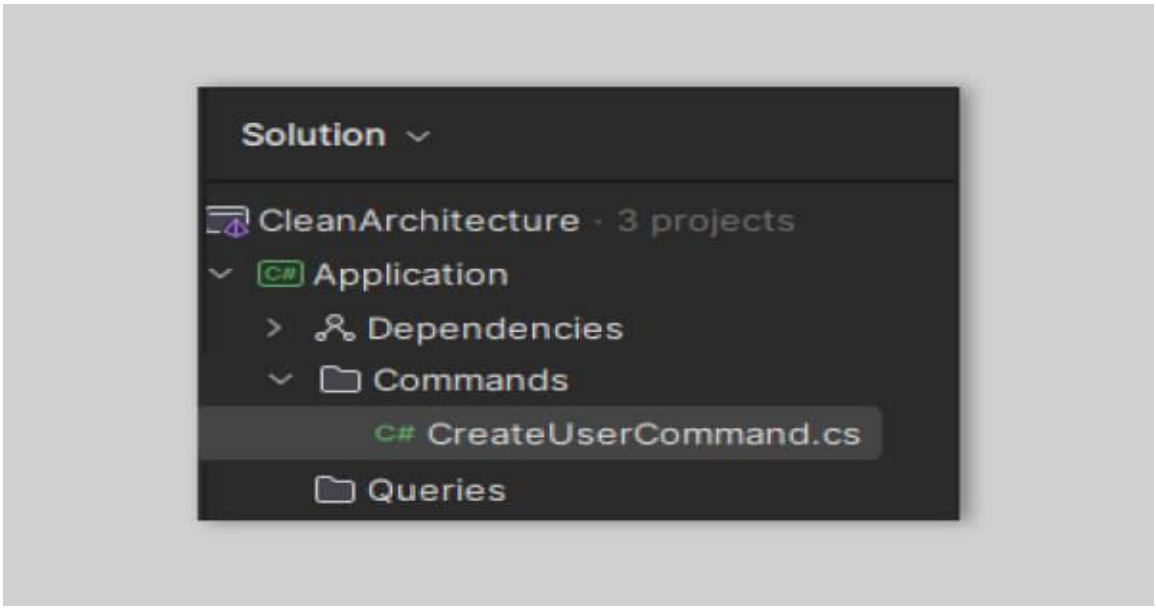
به‌صورت کلی، Command به فرآیندی گفته می‌شود که طی آن، تغییری در وضعیت سیستم به‌وجود می‌آید. در مقابل، اگر بخواهیم از وضعیت یک سرویس یا سیستم مطلع شویم، Query استفاده می‌شود. لایه اپلیکیشن جایی هست که Command ها و کوئری‌ها پیاده‌سازی خواهند شد.

در لایه Application، یک پوشه به اسم User ایجاد کنید و در داخل پوشه User، پوشه دیگری به نام Commands بسازید. سپس، لازم است یک کلاس به نام CreateUserCommand ایجاد شود. در تصویر زیر، نتیجه قابل مشاهده است.



هر Command ای که مربوط به کاربر باشد، درون پوشه Commands قرار می‌گیرد. به‌عنوان مثال، حذف یا ویرایش مشخصات کاربر درون این پوشه هستند.

می‌توان پوشه‌بندی‌های مختلفی مطرح کرد. به‌عنوان مثال، یک نوع مرسوم پوشه‌بندی در پروژه‌ها، دارا بودن دو پوشه به نام‌های Commands و Queries است؛ به‌طوری که تمامی تغییرات در پوشه Commands و تمامی درخواست‌ها در پوشه Queries نگهداری شوند. برای درک بهتر، به تصویر زیر توجه کنید.



مشکل این روش این است که اگر تعداد Command ها یا کوئری‌ها بیش از اندازه باشند، احتمالاً امکان پیدا کردن هرکدام آن‌ها، از میان انبوه کلاس‌ها دشوار خواهد بود. بنابراین، می‌توان این روش را در پروژه‌های کوچک استفاده کرد. در ادامه، از روش اول پوشه‌بندی استفاده خواهد شد؛ زیرا این رویکرد، برای پروژه‌های بزرگ کارایی مناسبی دارد.

- **Convention در نام‌گذاری Command ها و کوئری‌ها**

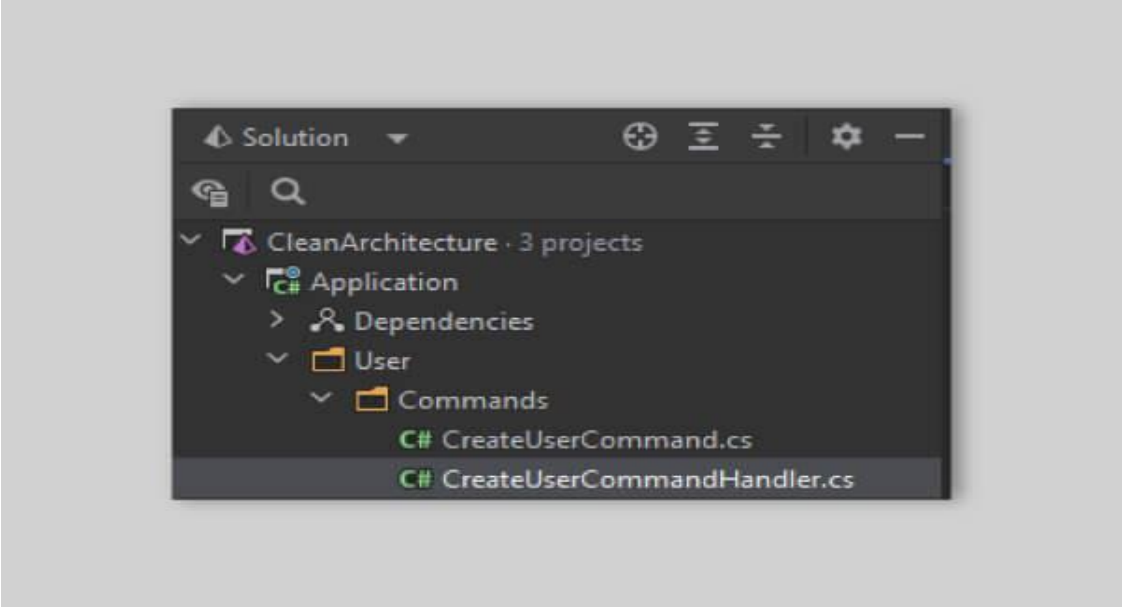
در صورتی که بخواهیم یک Command برای ایجاد کاربر داشته باشیم، ابتدای نام مربوطه، تسکی که قرار است انجام دهد (یعنی Create)، سپس نام فیچر (یعنی User) و درنهایت، عبارت Command آورده می‌شود.

به منظور درک نحوه پیاده‌سازی کلاس CreateUserCommnad، به قطعه کد زیر توجه کنید.

```
public class CreateUserCommand : IRequest<bool>
{
    public string Name { get; set; }
    public string Family { get; set; }
    public string Email { get; set; }
}
```

این کلاس از IRequest<T> ارث بری می‌کند که یکی از اینترفیس‌های پکیج MediatR است. پارامتر T در آن نمایانگر نوع پاسخ، بعد از ایجاد کاربر است. این شی در اینجا، از نوع Boolean خواهد بود؛ یعنی، زمانی که مقدار آن True باشد، کاربر با موفقیت ثبت شده است. البته می‌توان برای پروژه‌های مختلف مدل‌های گوناگونی ایجاد کرد. در این مثال، برای سادگی در بیان، آن را بولین در نظر گرفته‌ایم.

در مرحله بعدی، یک هندلر تحت عنوان CreateUserCommandHandler ، برای این Command ایجاد می‌کنیم. محل نگهداری این کلاس، داخل پوشه Commands (زیرمجموعه پوشه User) خواهد بود. به Convention ای که برای نام‌گذاری کلاس‌های هندلر استفاده می‌کنیم، دقت کنید. در این Convention، نام Command به‌همراه عبارت Handler در انتها قرار داده شده است. این نوع نام‌گذاری‌ها باعث می‌شوند تا سایر برنامه‌نویسان بدون صرف زمان زیادی، بتوانند کلاس‌های مدنظرشان را پیدا کنند.



در ادامه، قطعه پیاده‌سازی هندلر مربوط به ایجاد کاربر قرار داده شده است.

```
public class CreateUserCommandHandler : IRequestHandler<CreateUserCommand,
bool>
{
    public Task<bool> Handle(CreateUserCommand request, CancellationToken
cancellationToken)
    {
        return Task.FromResult(true);
    }
}
```

کلاس CreateUserCommandHandler از IRequestHandler<TRequest, TResponse> ارث بری می‌کند.

توجه کنید که TRequest نمایانگر نوع درخواست، TResponse نمایانگر نوع پاسخ و IRequestHandler یکی از اینترفیس‌های پکیج MediatR محسوب می‌شوند.

در متد Handle، می‌توان کاربر جدید را ایجاد و در دیتابیس مربوط به آن ذخیره کرد. اما در این مرحله، هنوز لایه مربوط به دیتابیس را ایجاد نکرده‌ایم و فقط به بازگرداندن مقدار True اکتفا می‌کنیم. این امکان وجود دارد که به جای مقدار Boolean، یک GUID برگردانیم. GUID نشان‌دهنده شناسه کاربر در دیتابیس است.

اکنون در این مرحله، می‌توانیم به منظور ایجاد کاربر، این Handler را در یک کنترلر استفاده کنیم.

مشابه قطعه‌کد زیر، در لایه Presentation و کنترلر User قرار گرفته و اقدامات لازم برای ایجاد یک کاربر با استفاده از MediatR را لحاظ کنید:

```
[ApiController]
[Route("api/[controller]")]
public class UserController : ControllerBase
{
    private readonly ISender _sender;

    public UserController(ISender sender)
    {
        _sender = sender;
    }

    [HttpPost]
    public async Task<IActionResult> Create(string name, string family,
string email)
    {
        var command = new CreateUserCommand()
        {
            Name = name,
            Family = family,
            Email = email
        };
        var response = await _sender.Send(command);
        return Ok(response);
    }
}
```

اینترفیس ISender برای پکیج MediatR است و به منظور ارسال درخواست توسط این پکیج به‌کار می‌رود. ضمن اینکه درخواست‌های ارسال‌شده توسط هندلرهای مرتبط با آن، دریافت و پردازش می‌شوند.

در این متد، ما یک درخواست CreateUserCommand را از HTTP دریافت می‌کنیم و آن را به MediatR می‌فرستیم. سپس، MediatR به جستجو برای Handler مناسب می‌پردازد و پس از یافتن آن، متد Handle را فراخوانی می‌کند. پس از ایجاد شدن کاربر جدید توسط هندلر، مقدار True در پاسخ HTTP برگردانده می‌شود.

این فقط یک مثال ابتدایی از نحوه پیاده‌سازی Use Case ها در Clean Architecture است. شما می‌توانید سایر موارد را با الگوبرداری از این روش پیاده‌سازی کنید.

- **نصب و استفاده از پکیج Fluent Validation**

فرض کنید باید خالی بودن مقادیر ورودی بررسی شود. محل مناسب برای پیاده‌سازی این منطق کجاست؟ بررسی خالی بودن مقدار یک فیلد، از موضوعات مرتبط با Business نیست. موارد مرتبط با کسب و کار، مانند تکراری بودن نام یا ایمیل، در لایه Domain پیاده‌سازی می‌شوند.

به منظور اعتبارسنجی مقدار یک فیلد، یک پکیج شناخته‌شده به نام Fluent Validation وجود دارد.

برای نصب Fluent Validation در لایه Application، قطعه کد زیر را تایپ کنید:

```
dotnet add package FluentValidation --version 11.8.0
```

حال برای بررسی مقادیر ورودی‌ها، در داخل پوشه Commands، یک کلاس به اسم CreateUserValidator ایجاد می‌کنیم که از AbstractValidator<T> ارث بری می‌کند. T نمایانگر نوع شی است که اعتبارسنجی خواهد شد و AbstractValidator (یکی از کلاس‌های پکیج Fluent Validation) برای پیاده‌سازی Rule های مختلف به‌کار می‌بریم.

به مثال زیر توجه کنید:

```
public class CreateUserValidator : AbstractValidator<CreateUserCommand>
{
    public CreateUserValidator()
    {
        RuleFor(x => x.Name)
            .NotNull()
            .NotEmpty();

        RuleFor(x => x.Family)
            .NotNull()
            .NotEmpty();

        RuleFor(x => x.Email)
            .NotNull()
            .NotEmpty();
    }
}
```

در قطعه کد بالا، این قاعده را تعیین کرده‌ایم که مقدار Name نمی‌تواند Null و Empty باشد. همین قاعده درخصوص سایر پارامترها نیز بررسی شده‌اند.

به ازای همه پکیج‌ها و کلاس‌های استفاده‌شده در این پروژه، باید Instance های مختلفی از آن‌ها تعریف شود و در بخش‌های گوناگون پروژه به‌کار برده شوند. Net Core. از یک IoC درونی پشتیبانی می‌کند و استفاده از آن به برنامه‌نویس کمک‌کننده است.

- **استفاده از Dependency Injection**

در ادامه این مقاله راه اندازی Clean Architecture ، موضوع Dependency Injection در پروژه‌های Clean بررسی می‌شوند. به‌طور کلی، لازم است یک کلاس به نام DependencyInjection در تمام لایه‌ها ایجاد شود. هرکدام از کلاس‌ها یک متد دارند و نام متد، ترکیبی از عبارت Add و نام لایه است.

به‌عنوان مثال، کلاس DependencyInjection لایه Application، به‌صورت زیر تعریف می‌شود:

```
public static class DependencyInjection
{
    public static IServiceCollection AddApplication(this
IServiceCollection services)
    {

    }
}
```

کلاس مذکور یک کلاس Static است و فقط یک متد به نام AddApplication دارد. این تابع، یک Extension Method برای IServiceCollection محسوب می‌شود. به این ترتیب، می‌توانیم تمامی Dependency Injection های مربوط به لایه Application را در این متد قرار دهیم. همین موضوع می‌تواند برای سایر لایه‌ها صادق باشد. درنهایت، می‌توان در کلاس Program.cs ، تمام Dependency Injection های پروژه را اضافه کرد:

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

builder.Services.AddApplication();
builder.Services.AddPresentation();
builder.Services.AddDomain();
```

به این ترتیب، هر یک از Configuration های مربوط به هر لایه از هم تفکیک شدند.

برای نمونه، متد مربوط به AddApplication تکمیل شد. این متد باید امکان رجیستر کردن دو پکیج نصب‌شده (MediatR و FluentValidation) را داشته باشد.

پکیج FluentValidation، یک متد Extension برای Injection دارد که با کمک این اکستنشن، register متد تسهیل می‌یابد.

دستور زیر را برای نصب اکستنشن Dependency Injection به‌کار ببرید:

```
dotnet add package FluentValidation.DependencyInjectionExtensions --version
11.8.0
```

پیاده‌سازی متد AddApplication به‌صورت زیر است:

```
public static class DependencyInjection
{
    public static IServiceCollection AddApplication(this
IServiceCollection services)
    {
        var assembly = typeof(DependencyInjection).Assembly;
        services.AddMediatR(configuration =>
            configuration.RegisterServicesFromAssemblies(assembly));

        services.AddValidatorsFromAssembly(assembly);

        return services;
    }
}
```

ورژن‌های جدید پکیج‌های MediatR و FluentValidation، به Assembly پروژه برای رجیستر کردن interface ها و پیاده‌سازی آن‌ها نیاز دارند. بنابراین، ابتدا یک متغیر به نام assembly تعریف کرده و آن را مقداردهی می‌کنیم. حال باید این متغیر در اختیار متدهای RegisterServicesFromAssemblies و AddValidatorsFromAssembly قرار گیرد. این متدها، اینترفیس‌های مشخص و از پیش تعریف شده خود را در داخل assembly داده شده جستجو می‌کنند. اگر کلاسی وجود داشته باشد که از این اینترفیس‌ها ارث‌بری کرده باشد، آن کلاس به‌عنوان پیاده‌سازی اینترفیس مذکور Register می‌شود.

مقدار بازگشتی متدهایی که برای Register کردن Dependency ها تعریف کردیم، از نوع IServiceCollection بودند.

به این ترتیب، می‌توان فراخوانی آن‌ها در کلاس Program.cs را به‌صورت زیر تغییر داد:

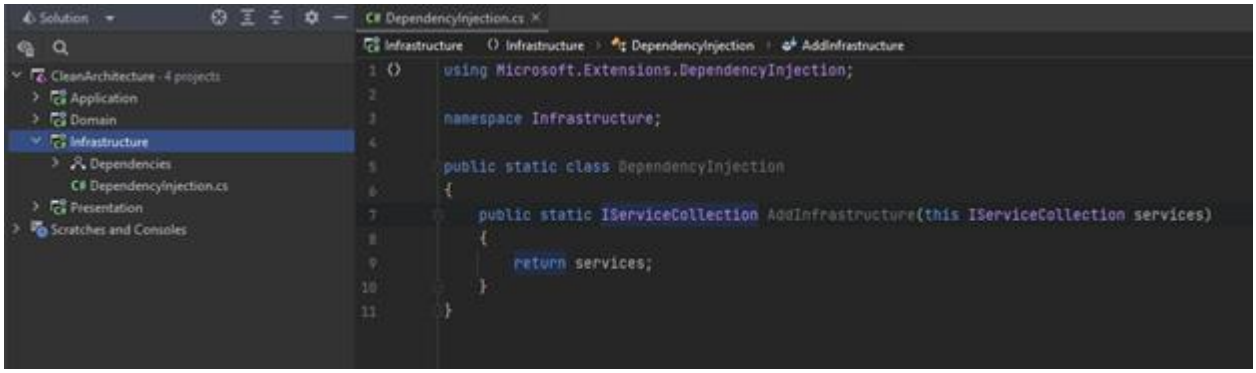
```
var builder = WebApplication.CreateBuilder(args);

builder.Services
    .AddPresentation()
    .AddApplication()
    .AddPresentation()
    .AddDomain();
```

در این بخش از راه اندازی Clean Architecture ، Dependency Injection بررسی شد. در ادامه، لایه آخر یعنی Infrastructure و اهمیت آن در راه اندازی Clean Architecture شرح داده می‌شود.

۵- ساخت لایه Infrastructure

در لایه آخر، تمرکز روی پیاده‌سازی‌های مربوط به دیتابیس و ارتباط با سرویس‌های خارجی است. البته می‌توان پیاده‌سازی‌های مرتبط با اتصال به دیتابیس را در لایه درونی‌تر، یعنی Persistence، قرار داد.

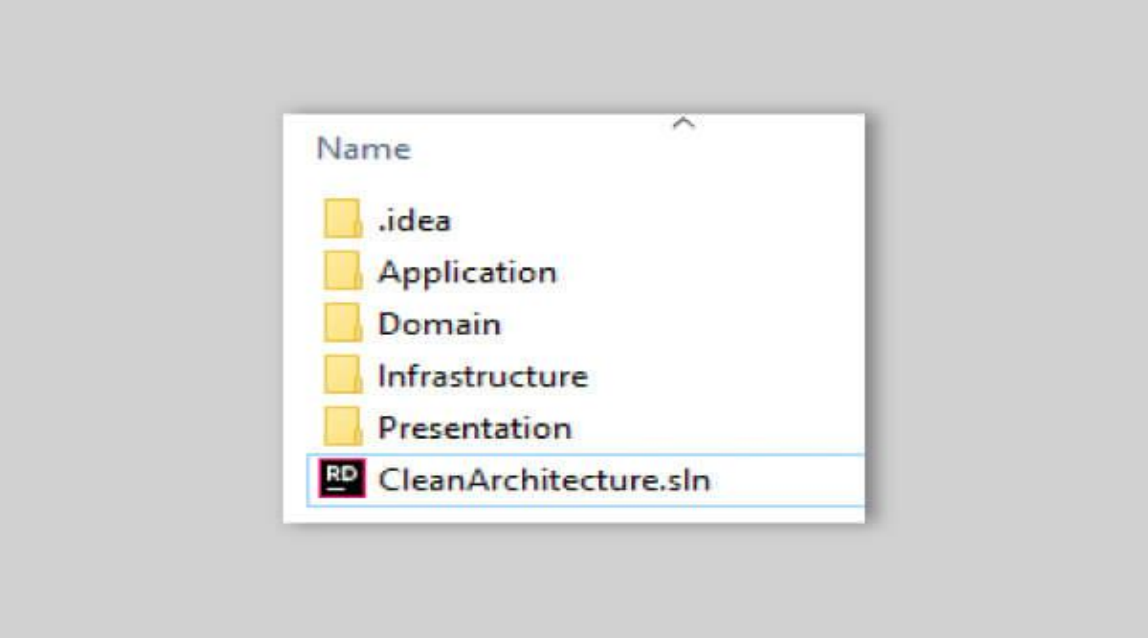
مجدداً یک Class Library با Net 7. ایجاد کرده و کلاس پیش‌فرض آن را حذف کنید. توجه کنید که یک کلاس به نام DependecyInjection نیز باید به آن اضافه شود. نتیجه در شکل زیر قابل مشاهده است:



درنهایت، کلاس Program.cs به‌صورت زیر، به‌روزرسانی می‌شود:

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

builder.Services
    .AddPresentation()
    .AddApplication()
    .AddPresentation()
    .AddDomain()
    .AddInfrastructure();
```

این یک نمونه از راه اندازی Clean Architecture در Net 7. است. در این مطلب، لایه‌های مختلف معماری Clean ، ازجمله لایه‌های Presentation ، Infrastructure، Domain و Application قرار دارند که همگی در زمان اجرا به یکدیگر متصل و اجرا می‌شوند.



راه اندازی Clean Architecture مشابه مثال فوق، شروع مناسب و ساختارمندی برای معماری محسوب می‌شود و شما می‌توانید آن را با گذر زمان و تغییر نیازمندی‌های معماری تکمیل کنید. البته راه اندازی Clean Architecture الزاماً پاسخگوی تمام مشکلات سیستم شما نیست و در کنار آن، لازم است سایر فاکتورهای مؤثر نیز بررسی شوند. در سال‌های اخیر، روش‌هایی مانند Layered Architecture و Slice Architecture نیز محبوبیت یافته‌اند.

مروری بر معماری Clean Architecture

در ابتدا ممکن است راه اندازی Clean Architecture در دات نت برای شما با چالش همراه باشد، اما این عمل پایه مستحکمی برای ساخت اپلیکیشن‌های قابل آزمایش، مقیاس‌پذیر و قابل نگهداری ایجاد خواهد کرد. درواقع با به‌کارگیری قوانین معماری تمیز ، اپلیکیشن شما به لایه‌های متمایزی تقسیم می‌شود که هر کدام کارایی مشخصی دارا هستند. در این مطلب، مراحل لازم برای راه اندازی معماری تمیز (Clean Architecture) را مورد بررسی قرار دادیم و در مقاله آتی، قصد داریم پروژه را مبتنی بر Domain Driven Design ادامه دهیم.