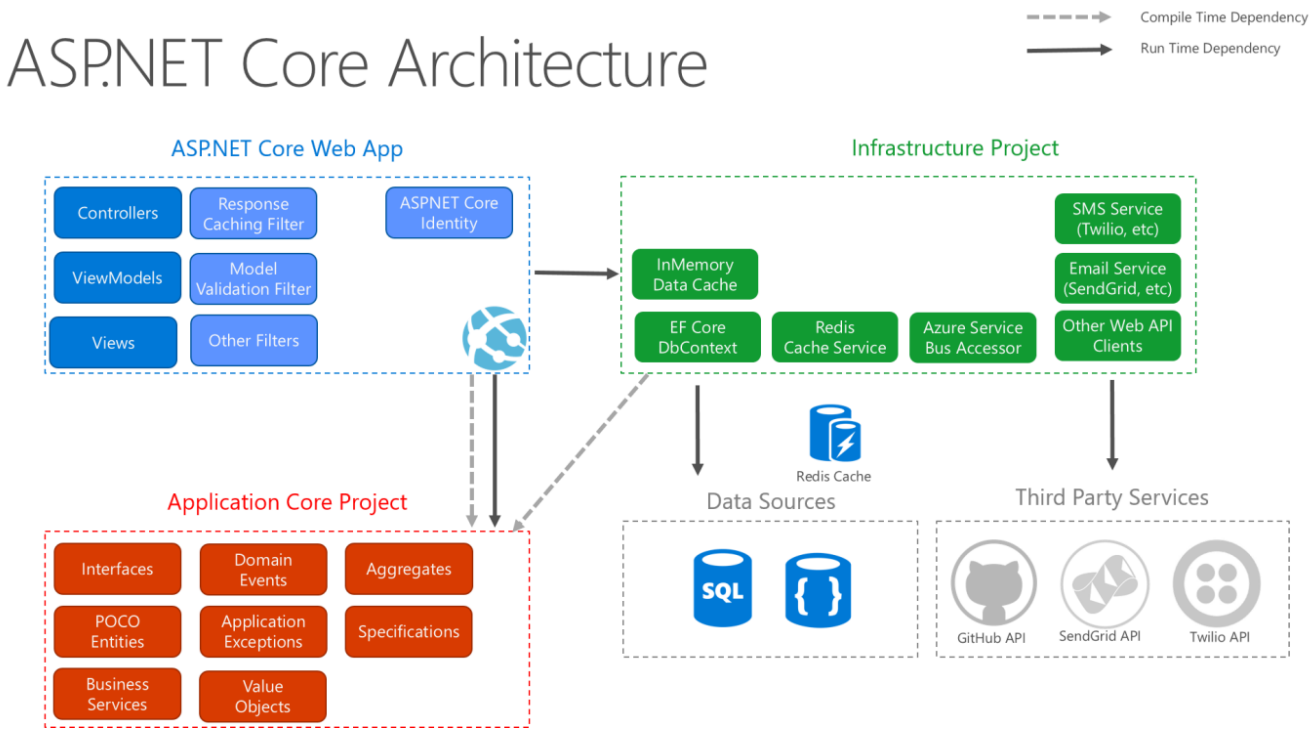
Clean Architecture .NET Core (Part 2: Implementation)

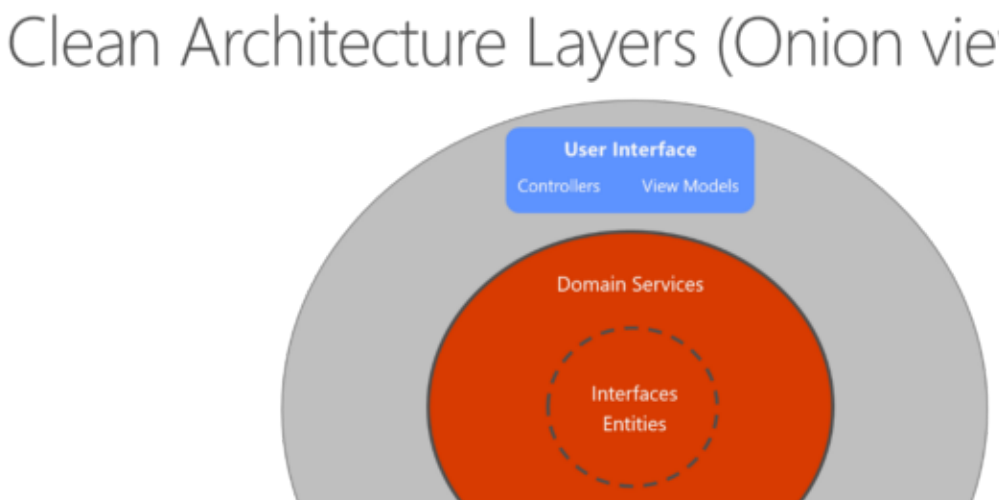
Nishan Chathuranga Wickramarathna Mar 20, 2020 · 11 min read



More detailed view of an ASP.NET Core application's architecture when built following clean architecture recommendations. Source — Common web application architectures

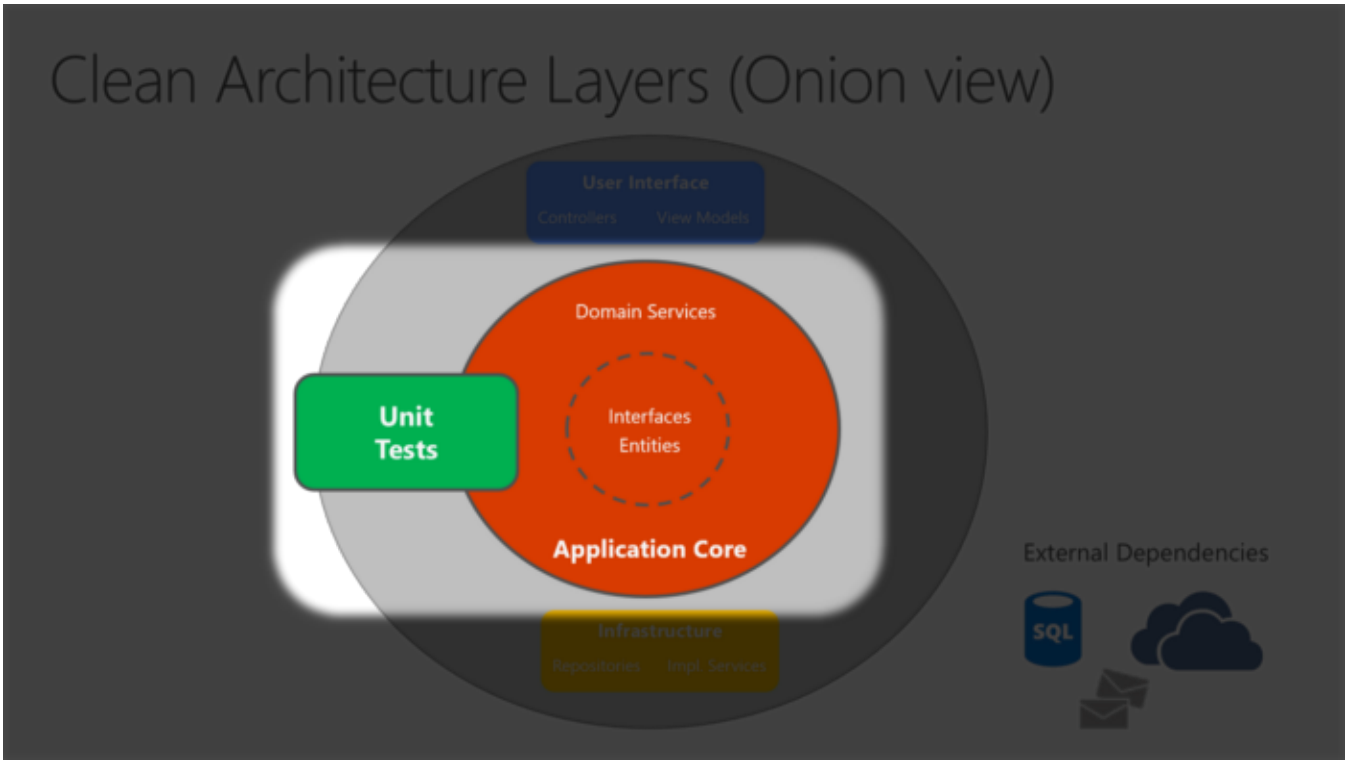
As per the previous article I introduced you to the basic practices of the Clean Architecture. Now we are going to build an application using ASP.NET Core 3, starting with directory structure. Quick recap before moving on.

Recap



In this diagram, dependencies flow toward the innermost circle. The Application Core takes its name from its position at the core of this diagram. And you can see on the diagram that the Application Core has no dependencies on other application layers. The application's entities and interfaces are at the very center. Just outside, but still in the Application Core, are domain services, which typically implement interfaces defined in the inner circle. Outside of the Application Core, both the UI and the Infrastructure layers depend on the Application Core, but not on one another (necessarily).

Because the Application Core doesn't depend on Infrastructure, it's very easy to write automated unit tests for this layer.



Since the UI layer doesn't have any direct dependency on types defined in the Infrastructure project, it's likewise very easy to swap out implementations, either to facilitate testing or in response to changing application requirements. ASP.NET Core's built-in use of and support for dependency injection makes this architecture the most appropriate way to structure non-trivial monolithic applications.

Prerequisites

1. Visual Studio
2. .NET Core SDK (make sure to look for the SDK that supports the Visual Studio version you're using, I'm on Visual Studio 2019 and .NET Core 3.1)
3. Basic understanding about .NET Core Web Applications, MVC, C#, SQL Server, Migrations, Identity, Entity Framework and Visual Studio

If you are on Visual Studio 2017 or earlier, .NET Core 2.1 will be the supported version. Get the correct version

This article is about how to setup the project structure for Clean Architecture, not about .NET Core

I recommend you clone this project and read the article step by step by checking the project, to avoid confusion.

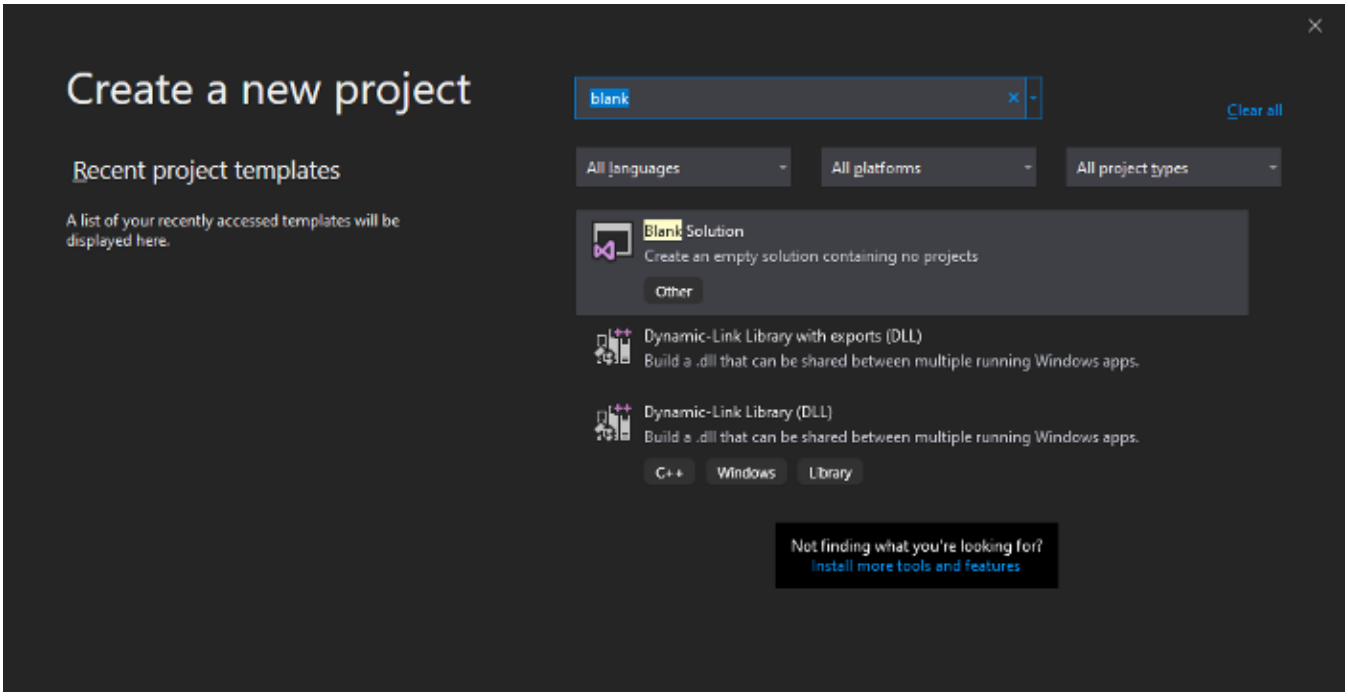
nishanc/CleanArchitectureDemo

A demo project built on .NET Core Clean Architecture - nishanc/CleanArchitectureDemo

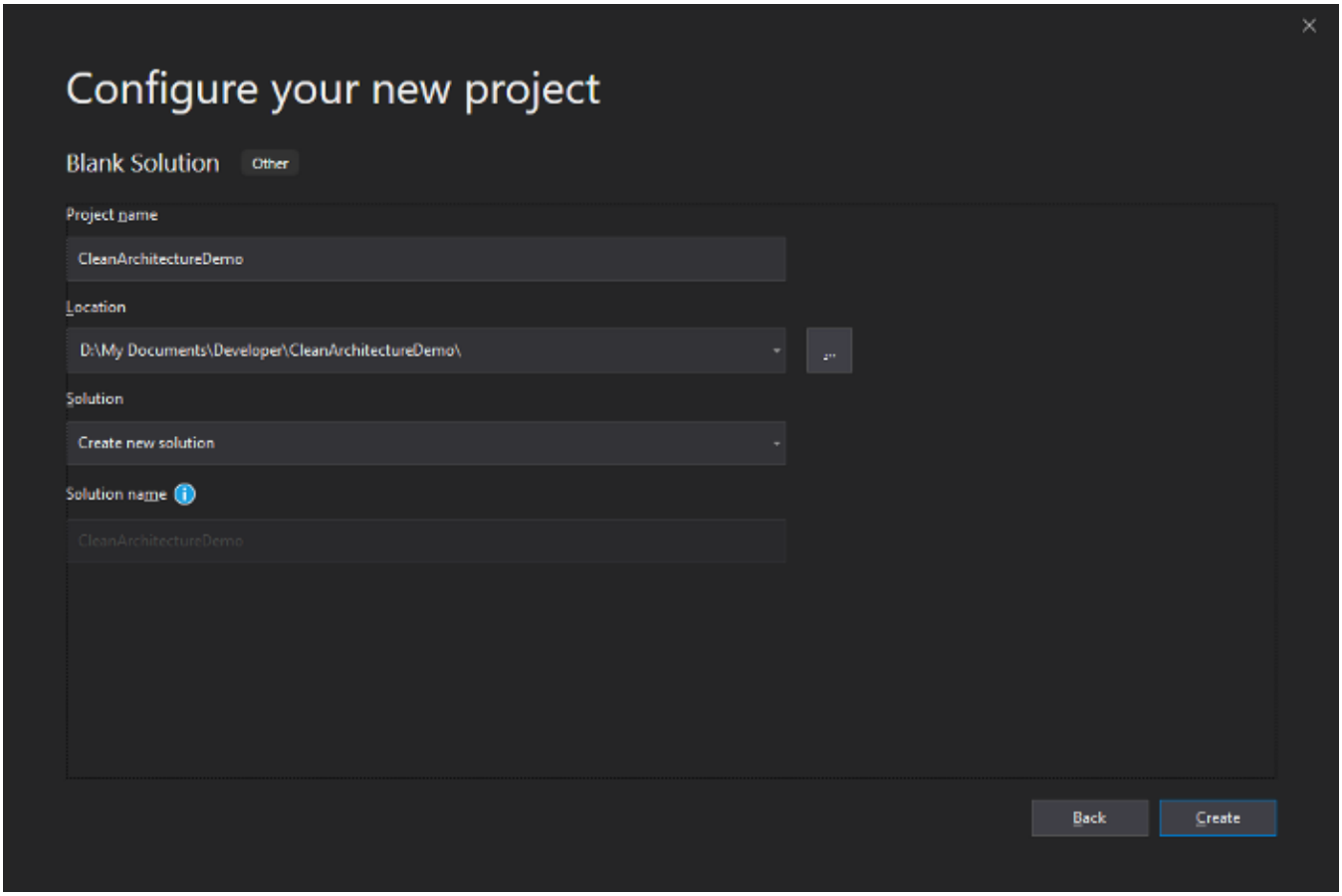
github.com

Directory Structure

Create a blank solution.

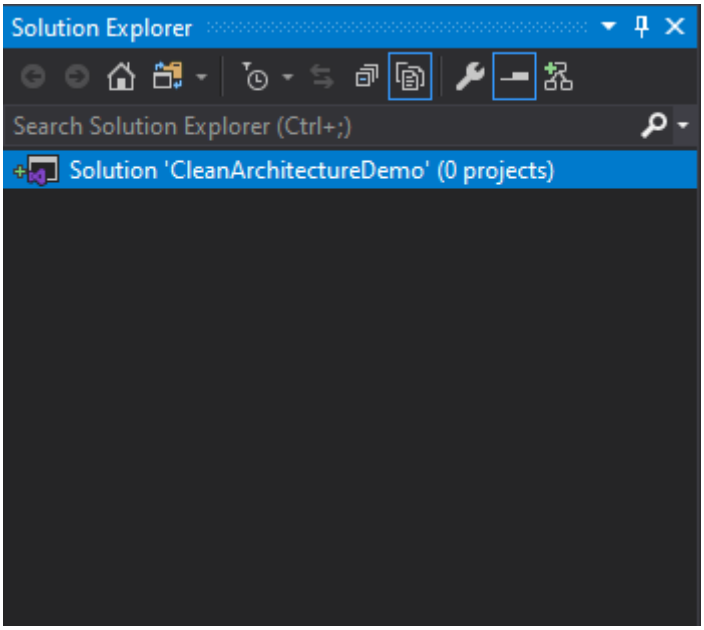


Search for the blank solution project in File -> New -> Project. Select ‘Blank Solution’ and click ‘Next’.



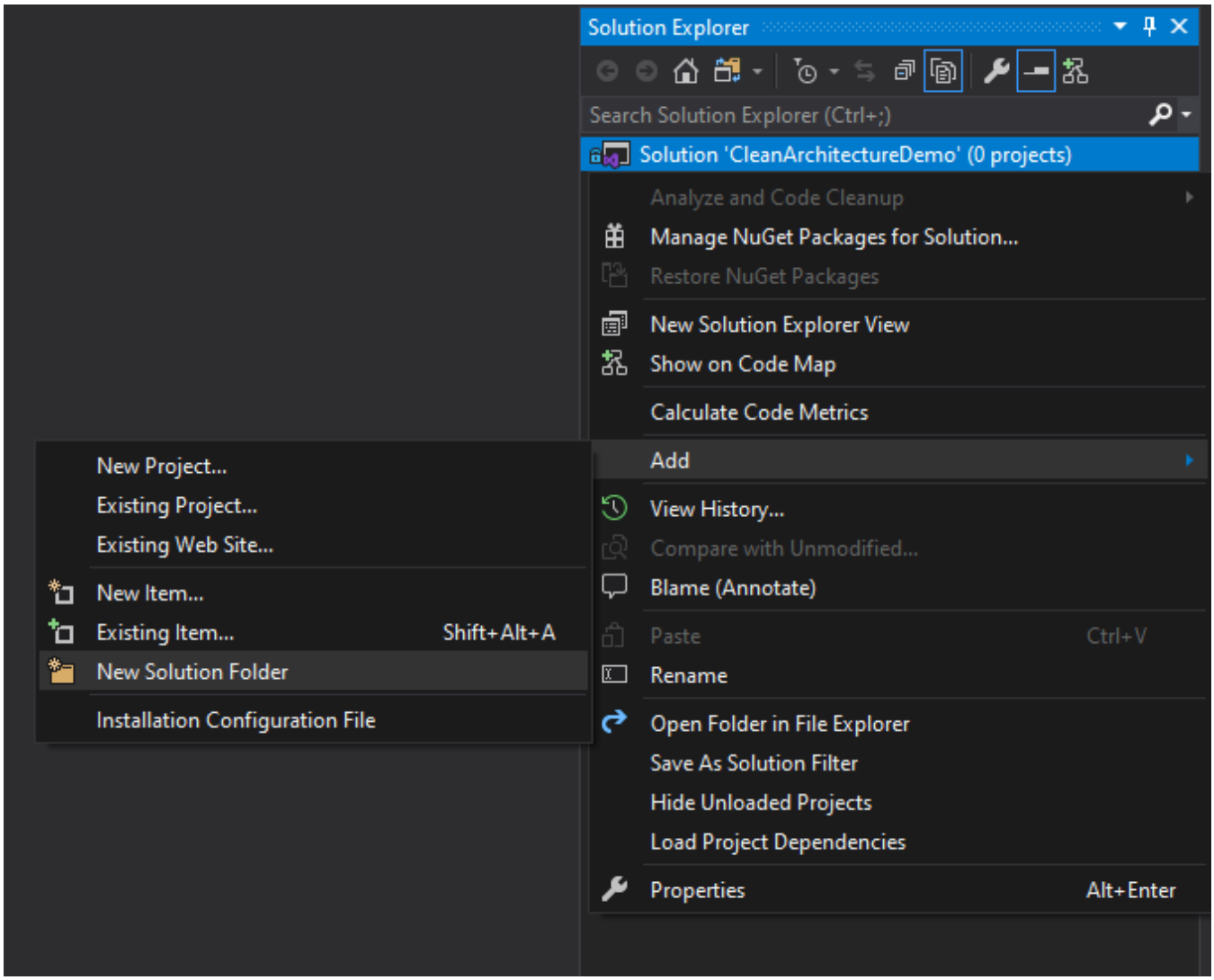
Give a name for your project, browse location and click ‘Create’.

Now your solution explorer should look like this.

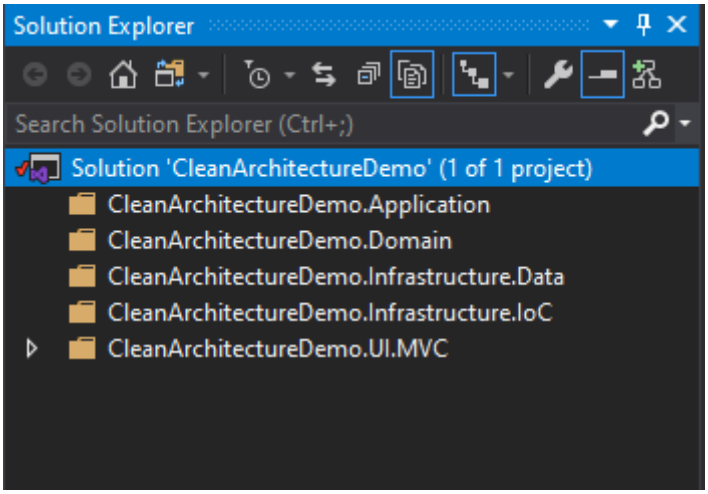


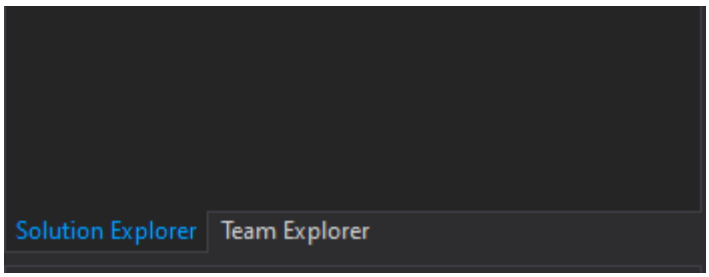
Now lets create the folders which we will use to store individual projects.

Create following folders inside your solution, this is just a suggestion.



Right click on the solution -> Add -> New Solution Folder



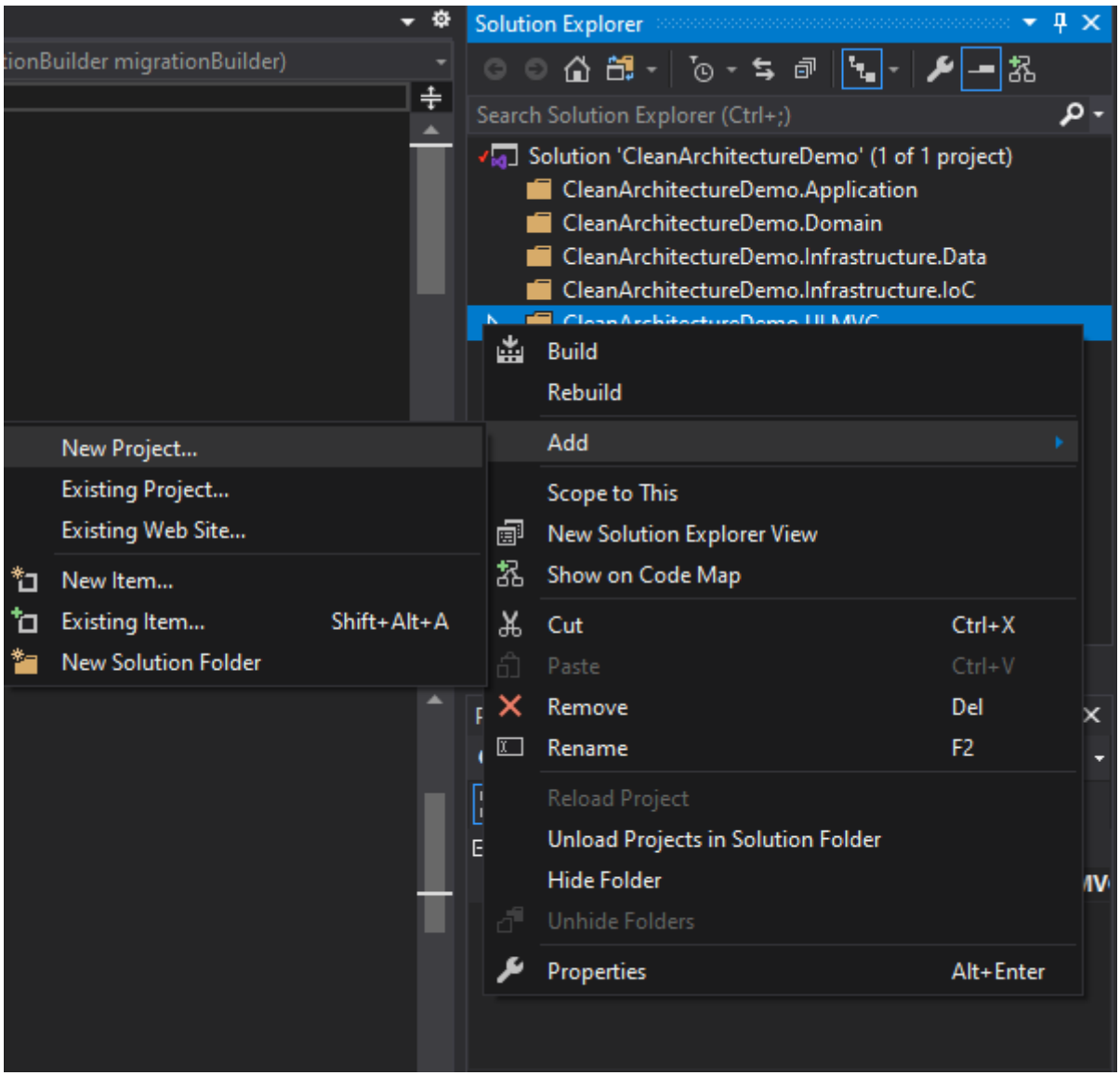


Create folder as seen here.

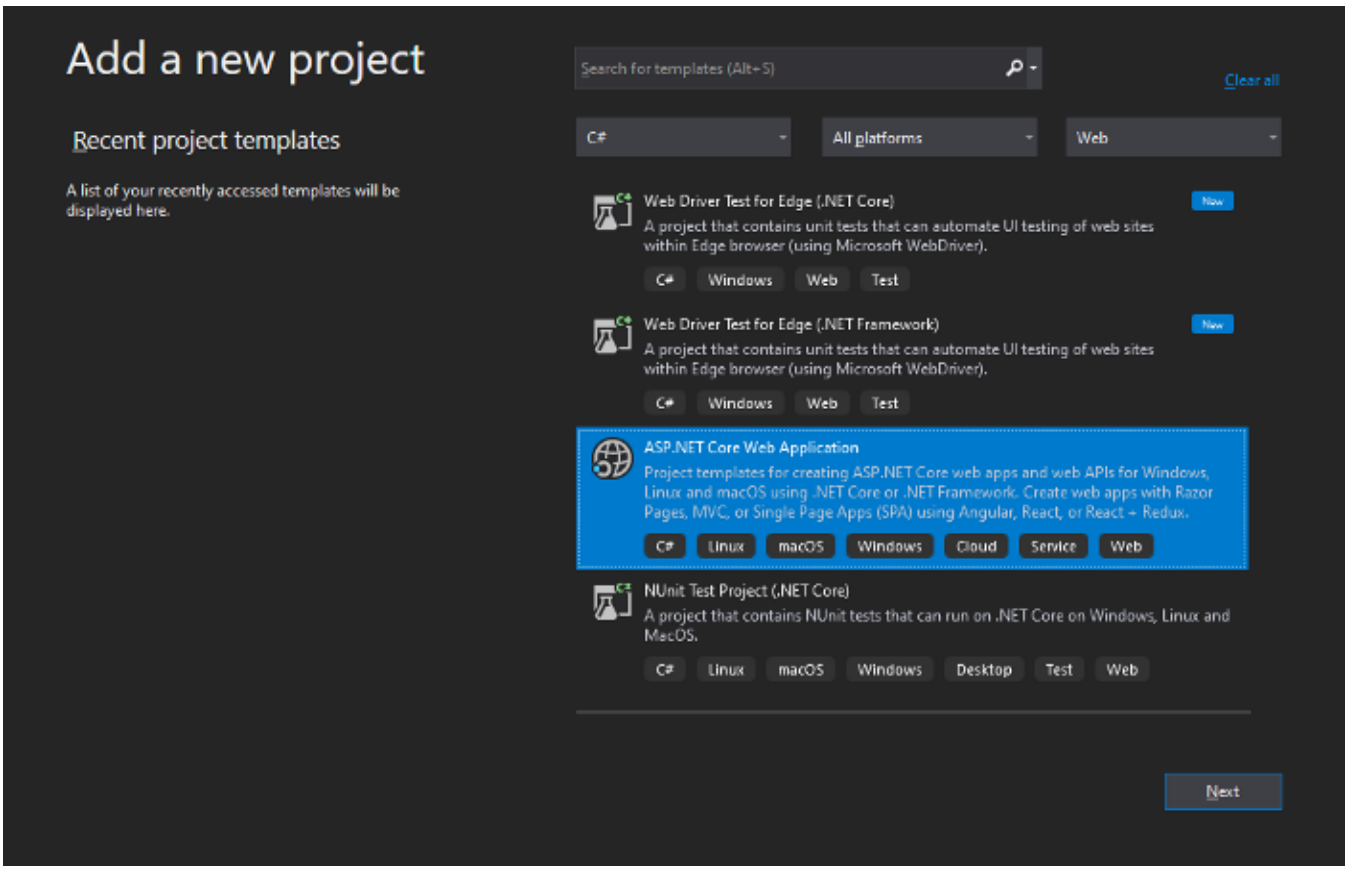
Application will take care of our interfaces, services, business rules. **Domain** consists of Entities, **Data** knows about how to access our data, **IoC** (Inversion of Control) will help us to dependency injection.

User Interface

Let’s create the MVC web application under UI.MVC folder.

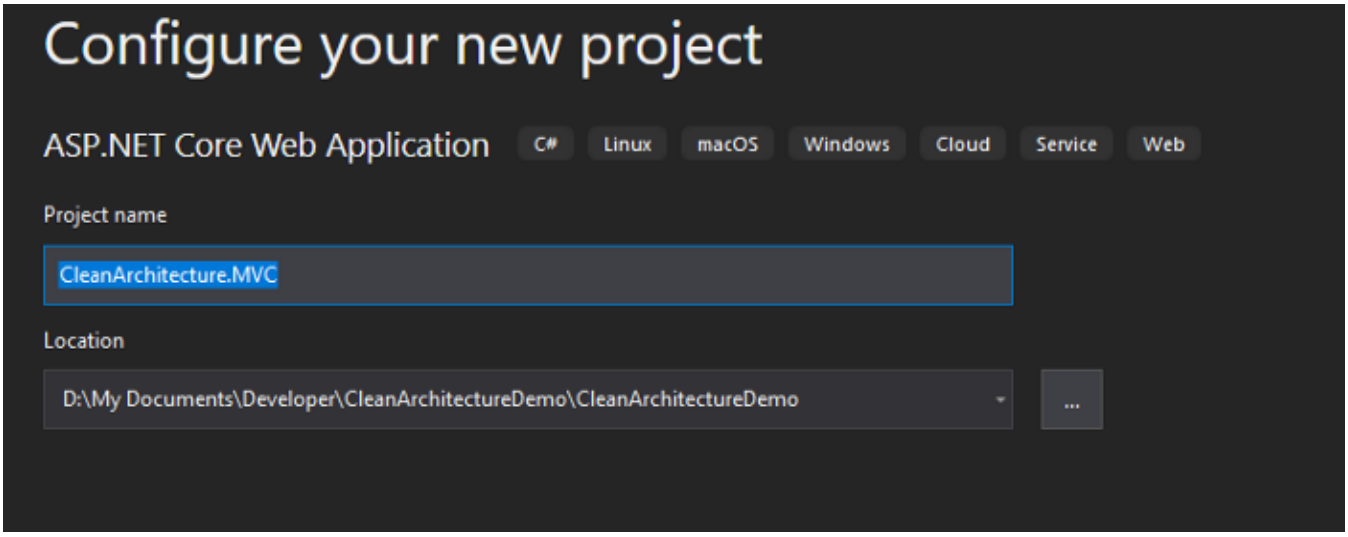


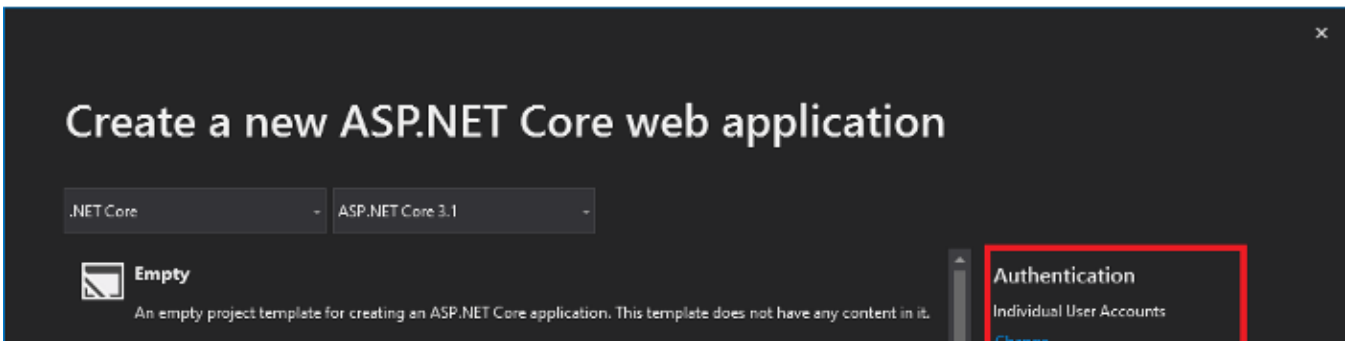
Right click on “CleanArchitecture.UI.MVC” -> Add -> New Project

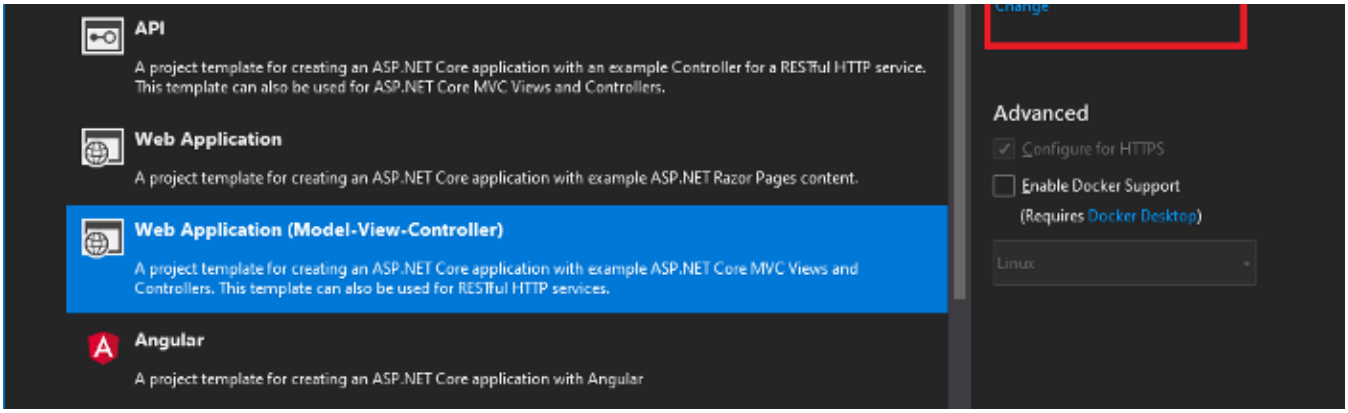


Select ASP.NET Core Web Application. (Follow this stackoverflow question if this project type does not show up)

Following the same naming convension we followed, name the project as ‘CleanArchitecture.MVC’ and hit ‘Create’.







Select Web Application (Model-View-Controller) and change the Authentication to ‘Individual User Accounts’

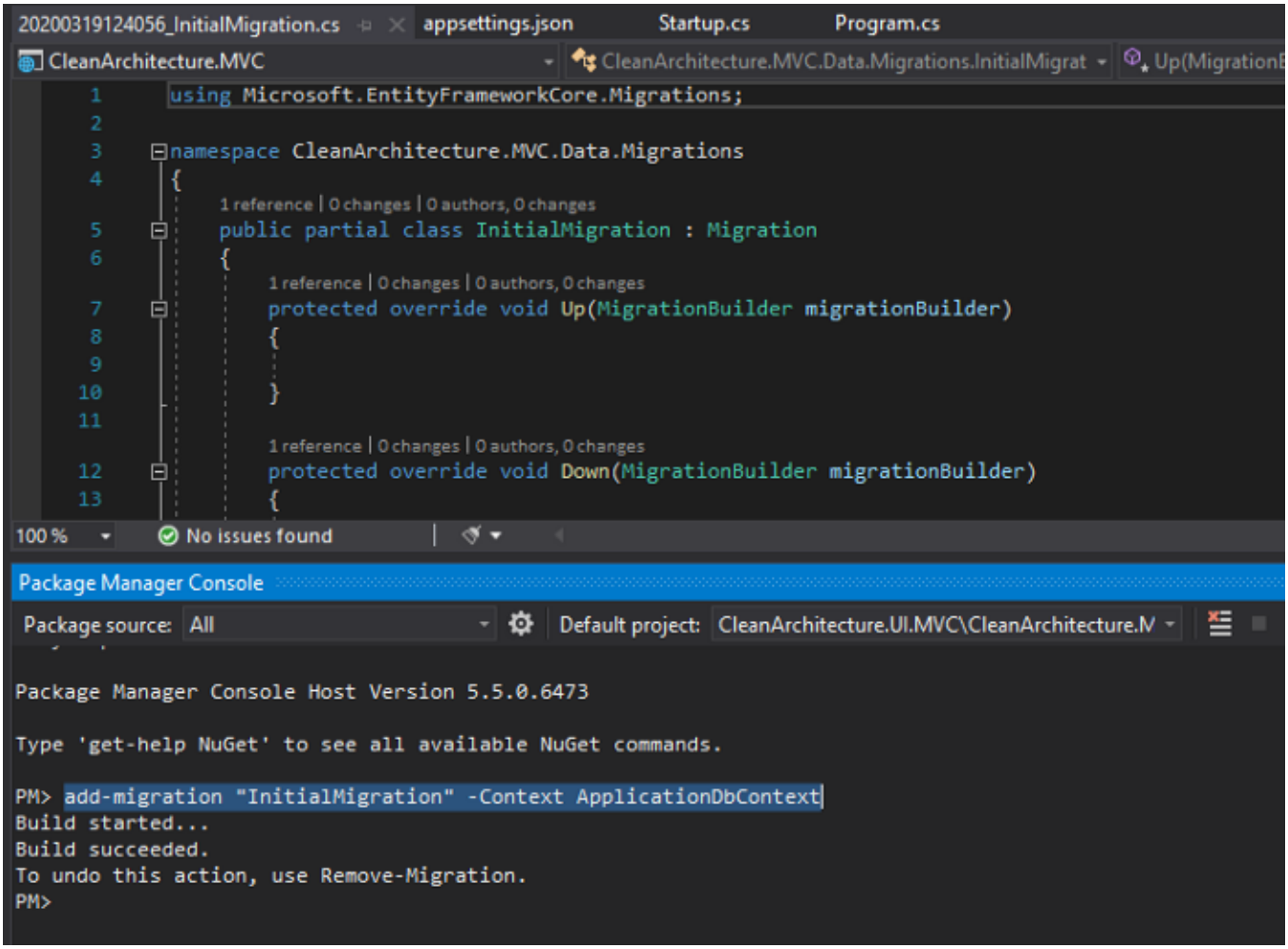
This will create a new project and scaffold some code. Later on we will move models to Entities later, since we are taking the Clean Architecture approach. Once that is done let’s quickly change the connection string to point to a SQL Server database. (If you son’t have SQL Server installed I recommend you downloading the **developer edition** from here, and SQL Server Management Studio from here.)

Go to `appsettings.json` and change the connection string to point your server, I will create a database named **Demo**.

Let’s update the database with Identity model classes using a migration. Open Package Manager Console (Tools -> NuGet Package Manager -> Pakage Manager Console.) and type,

```
add-migration "InitialMigration" -Context ApplicationDbContext
```

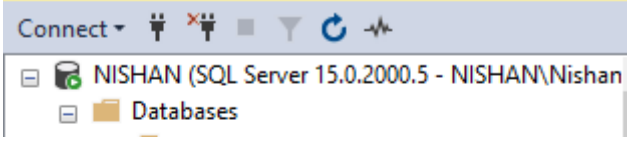
Notice we have specified the DbContext class name here, because **we will** have another DbContext in the future for other entities. This command will create a migration named “InitialMigration”, it looks empty but it will create the Identity tables set we want.

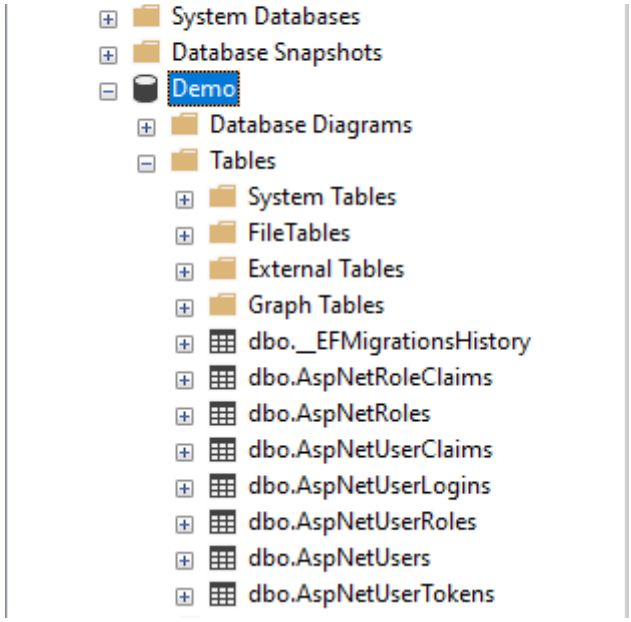


You can have another project for Identity Migrations as well if you want. I will demonstrate how to do it by moving the core entities out from MVC project. But first, after creating the migration, let’s update the database.

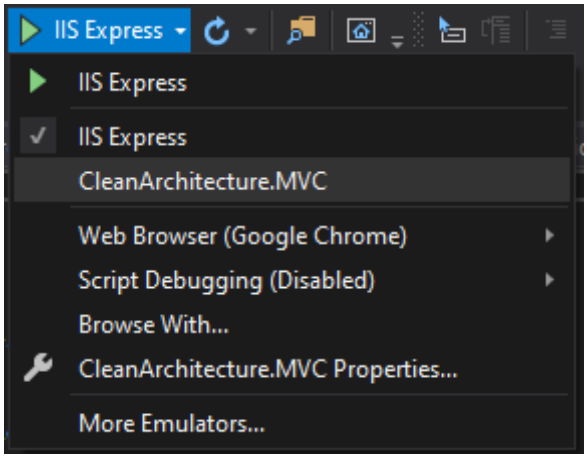
```
update-database
```

Once that’s done you will see your database has new set of tables.





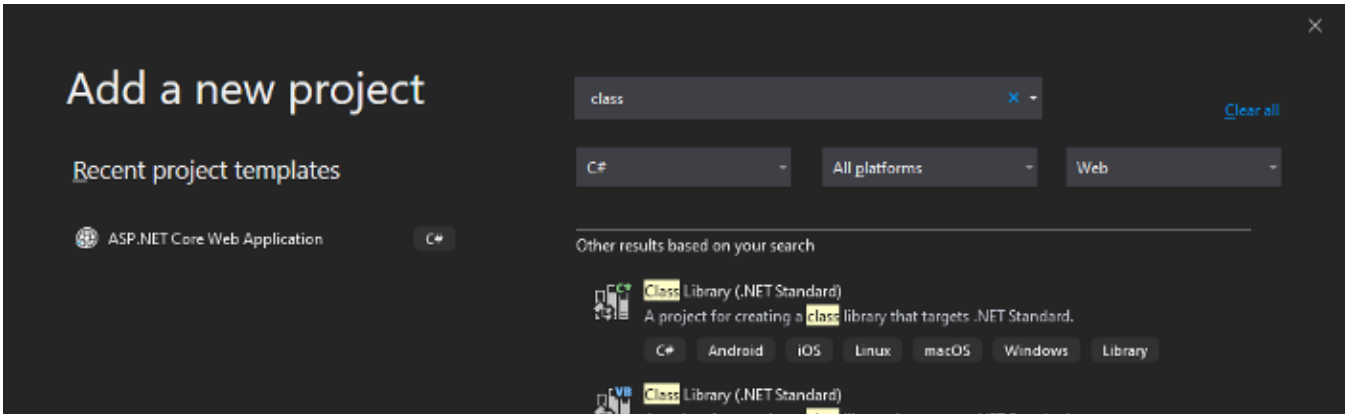
You can, at this point, run this project by changing the launch profile to ‘CleanArchitecture.MVC’ from slandered toolbar, and register a new user and see if that works.

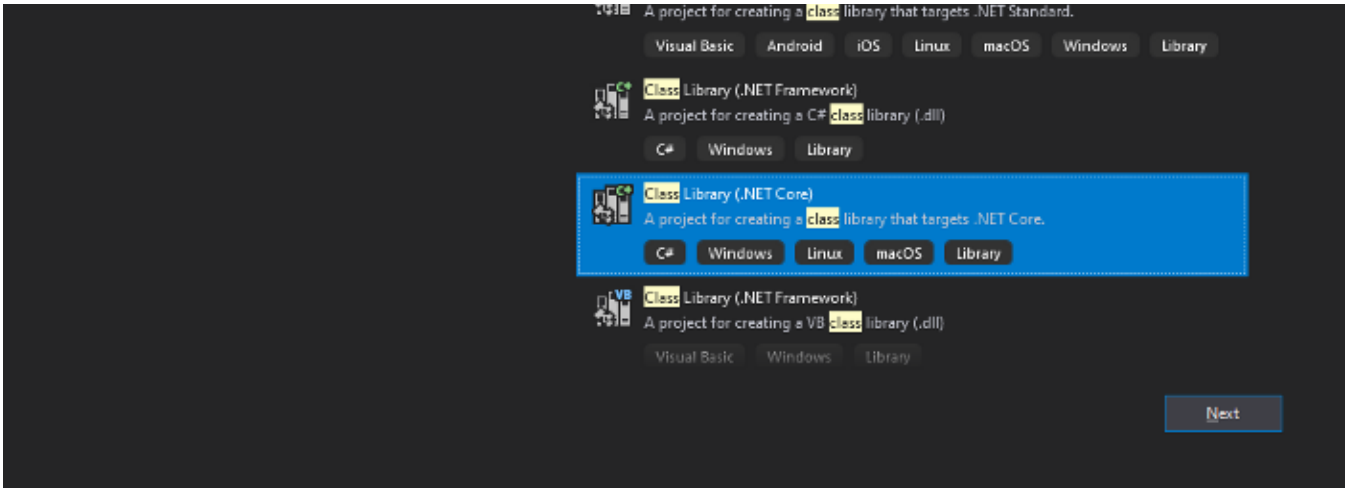


I should point out that you can rename your solution folders to reflect your application, doesn’t have to follow my way of naming things, but adhering to the clean architecture, of course.

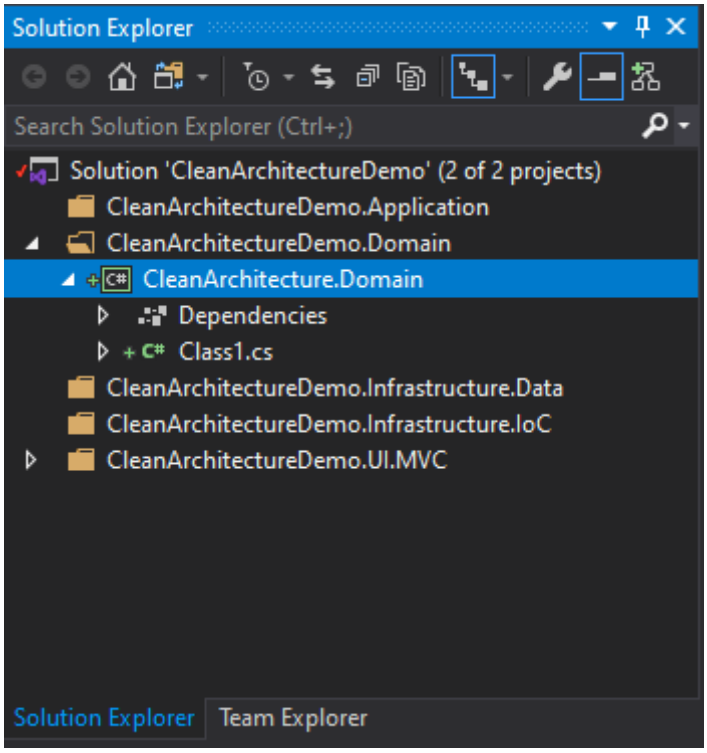
Domain

Let’s add a Class Library project to our domain layer which will hold the core entities.



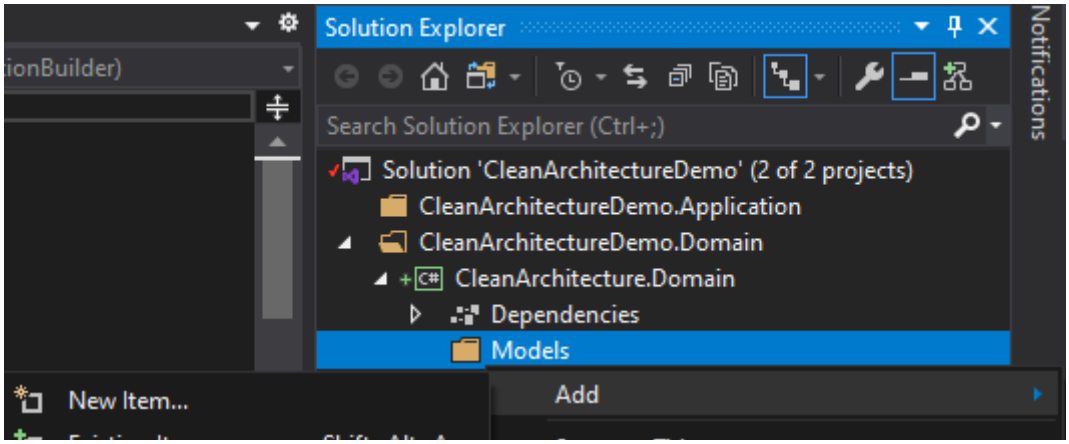


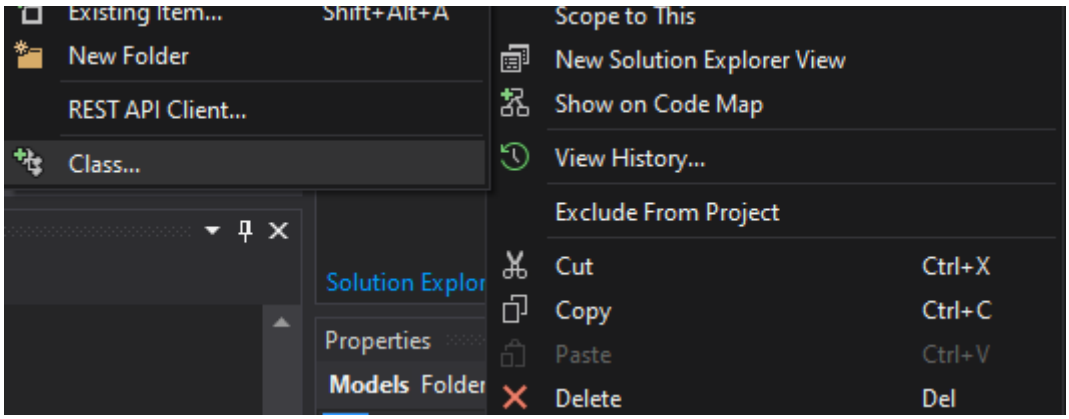
Select Class Library (.NET Core)



Name it as Domain suffix

Remove **Class1.cs** and add a new folder **Models** (or Entities), for the sake of this article let's assume we are building a library system, so we will add a **Book** model to the Models folder.



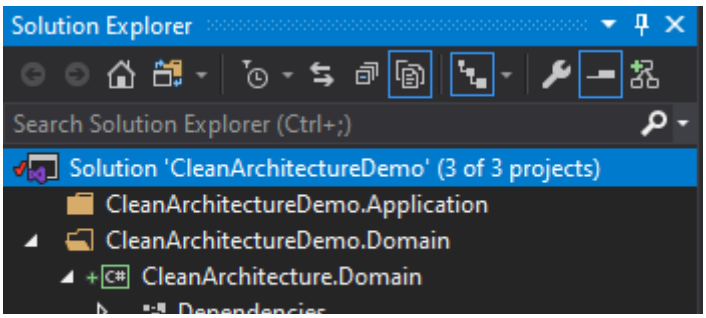


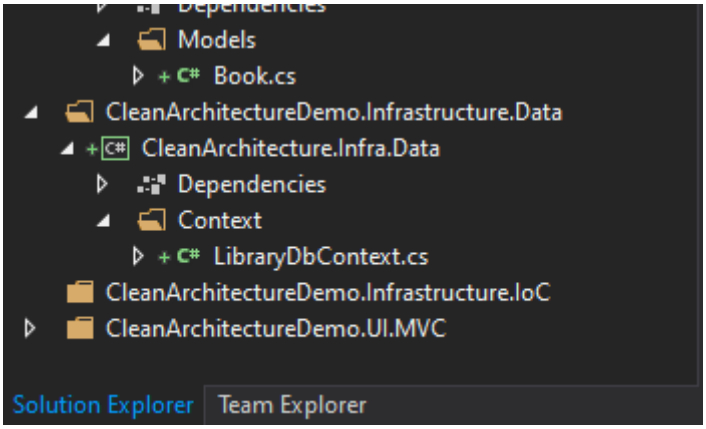
Right click -> Add -> Class, name it as Book and click Add.

Add the properties you want to the entity to have.

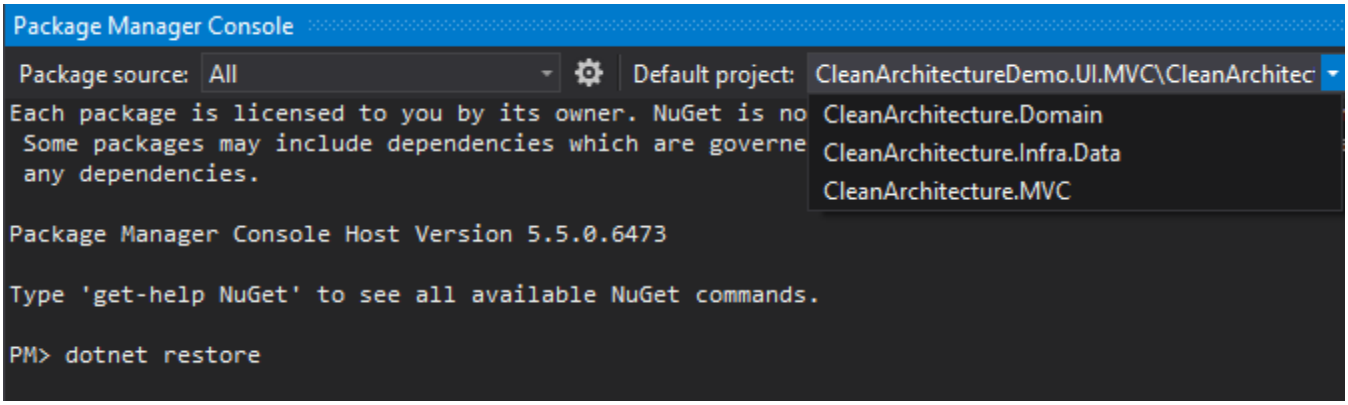
Notice that I have made the class **public.**

Now let’s create a new database context so we can update the database with this new model. We will be doing it in our **Infrastructure** layer. Under **Infrastructure.Data** add new class library called **CleanArchitecture.Infra.Data** and under that folder named **Context**. Next create a class named **LibraryDbContext.**





Let's go ahead and add packages we need to configure this project. But before that, in Package Manager Console, execute `dotnet restore` for all 3 projects we have so far. After that do **Build -> Rebuild Solution**



Right click on **Dependencies** of **CleanArchitecture.Infra.Data** and select **Manage NuGet Packages**. Then click on browse tab and install,

```
Microsoft.EntityFrameworkCore
Microsoft.EntityFrameworkCore.Design
Microsoft.EntityFrameworkCore.SqlServer
Microsoft.EntityFrameworkCore.Tools
```

Make sure all of them are stable versions (not previews) and are in the same version.

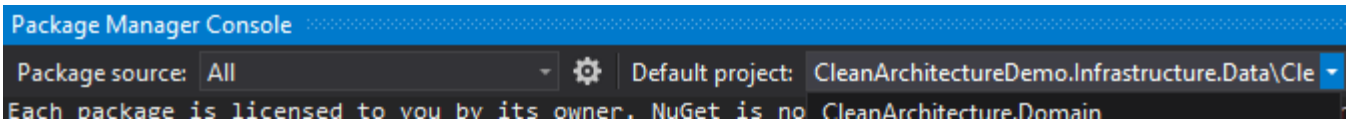
Now we can inherit our LibraryDbContext from DbContext provided by **Microsoft.EntityFrameworkCore**. And you can see that we are referring to the Book entity provided by **CleanArchitecture.Domain.Models .** Add following content to the**LibraryDbContext**

Let’s configure this new DbContext in Startup.cs of MVC project. Add following lines to **ConfigureServices** method.

Notice the **LibraryConnection** connection string, so we need to add it to appsettings.json. This new connection string points to a different database, where we will store our domain entities.

```
"ConnectionStrings": {
    "DefaultConnection":
"Server=.;Database=Demo;Trusted_Connection=True;MultipleActiveResultSets=t
    "LibraryConnection":
"Server=.;Database=Library;Trusted_Connection=True;MultipleActiveResultSet
}
```

After that is done, execute `add-migration` against **LibraryDbContext**



```
Some packages may include dependencies which are governe
any dependencies.

Package Manager Console Host Version 5.5.0.6473

Type 'get-help NuGet' to see all available NuGet commands.

PM> add-migration "InitialMigration" -Context LibraryDbContext
```

CleanArchitecture.Infra.Data
CleanArchitecture.MVC

Make sure you have selected CleanArchitecture.Infra.Data as the default project.

This will create a new migration under the infrastructure.

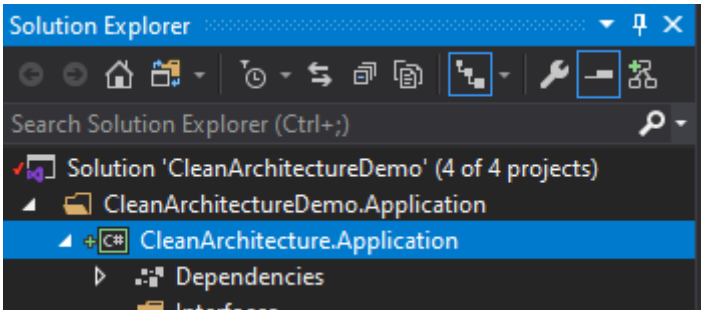
```
1 reference | 0 changes | 0 authors, 0 changes
public partial class InitialMigration : Migration
{
    0 references | 0 changes | 0 authors, 0 changes
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Books",
            columns: table => new
            {
                Id = table.Column<int>(nullable: false)
                    .Annotation("SqlServer:Identity", "1, 1"),
                Name = table.Column<string>(nullable: true),
                ISBN = table.Column<string>(nullable: true),
                AuthorName = table.Column<string>(nullable: true)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Books", x => x.Id);
            });
```

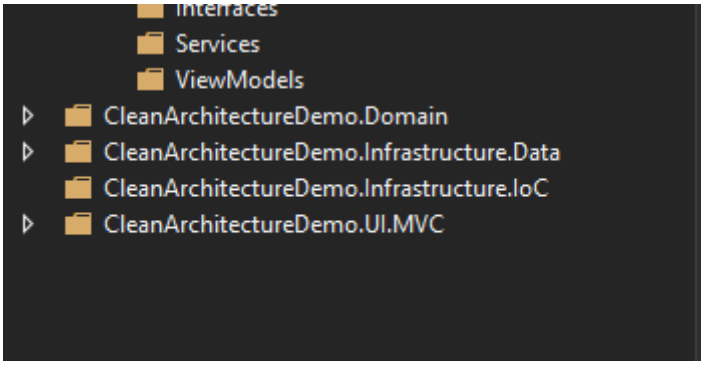
Update database using,

```
update-database -Context LibraryDbContext
```

Application Core

All of our services, interfaces and ViewModels will go here. Like in previous steps, create a new Class Library (.NET Core) under '**CleanArchitectureDemo.Application**' folder named **CleanArchitecture.Application** and add folders for them as follows.

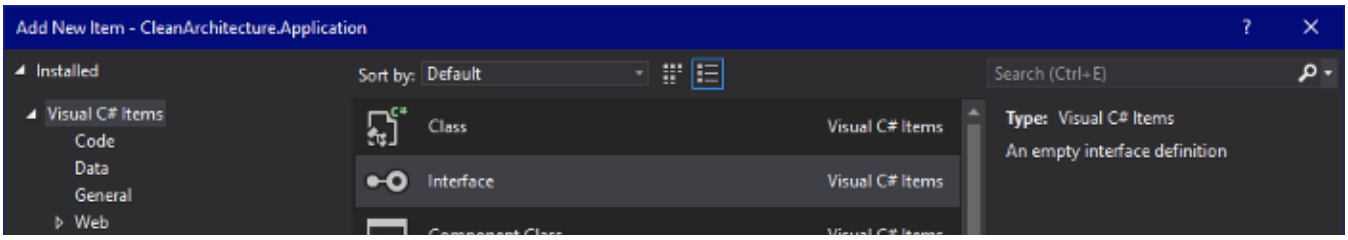


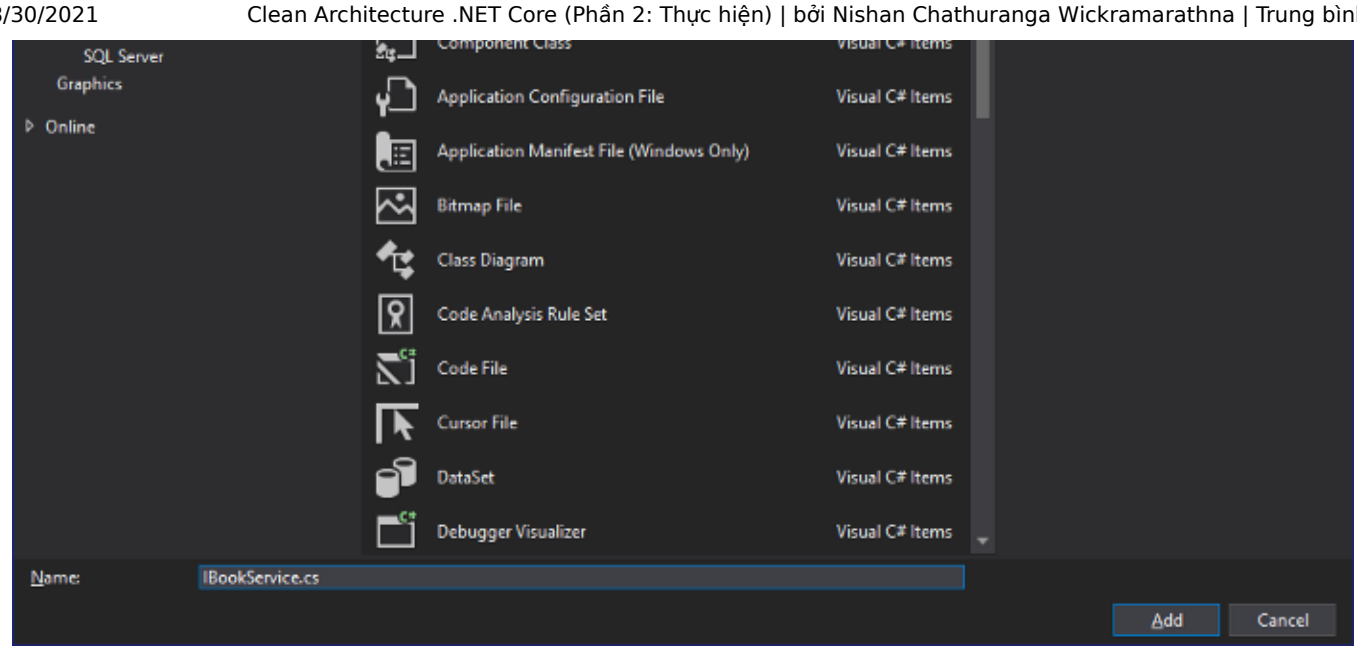


Lets work on the ViewModels now, A `view model` represents the data that you want to display on your view/page, whether it be used for static text or for input values (like textboxes and dropdown lists) that can be added to the database (or edited). It is something different than your `domain model` . It is a model for the view. In other words, it creates a mask for the domain models.

Create a new class under ViewModels folder named **BookViewModel.** For the time being we will get a list of Books from the database. Add following code to **BookViewModel.cs** , notice we are bringing in the **Book** from **CleanArchitecture.Domain.Models**

Create a new interface to act as a contract to the functionality that we're trying to implement. I hope you are familiar with interfaces and why we create interfaces in OOP, since those things are out of the scope of this article I am not going to discuss them here. Under **Interfaces** folder create a new **interface**,





named, **IBookService.cs**

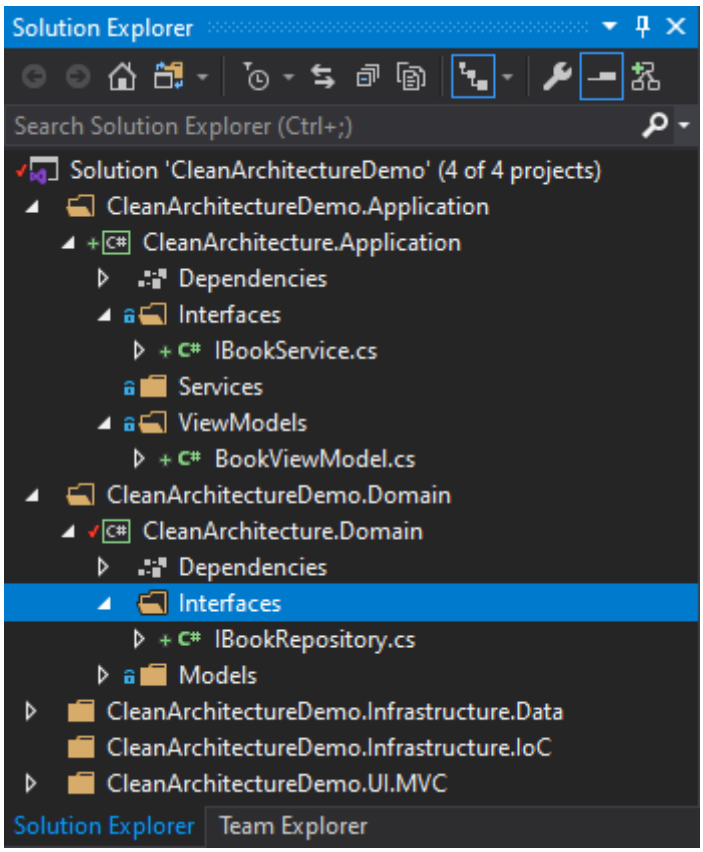
When implemented this method will return list of books, and it only knows about the ViewModel, not about the core domain model Book, so we are abstracting the core entity by doing this, rather than having everything in one place.

Before we write the implementation for IBookService, we have to define a way to get the data from the database, To do that what we normally use in .NET is an ORM called Entity Framework, but we will use the **Repository pattern** to decouple the business logic and the data access layers in our application.

The Repository Design Pattern in C# Mediates between the domain and the data mapping layers using a collection-like interface for accessing the domain objects. In other words, we

can say that a Repository Design Pattern acts as a middleman or middle layer between the rest of the application and the data access logic.

Add another folder called Interfaces under **CleanArchitecture.Domain** project. Add a new interface named **IBookRepository.cs**



At this point our project should look like this.

Add following method, notice this time it’s not the ViewModel, it’s the domain entity itself, Book.

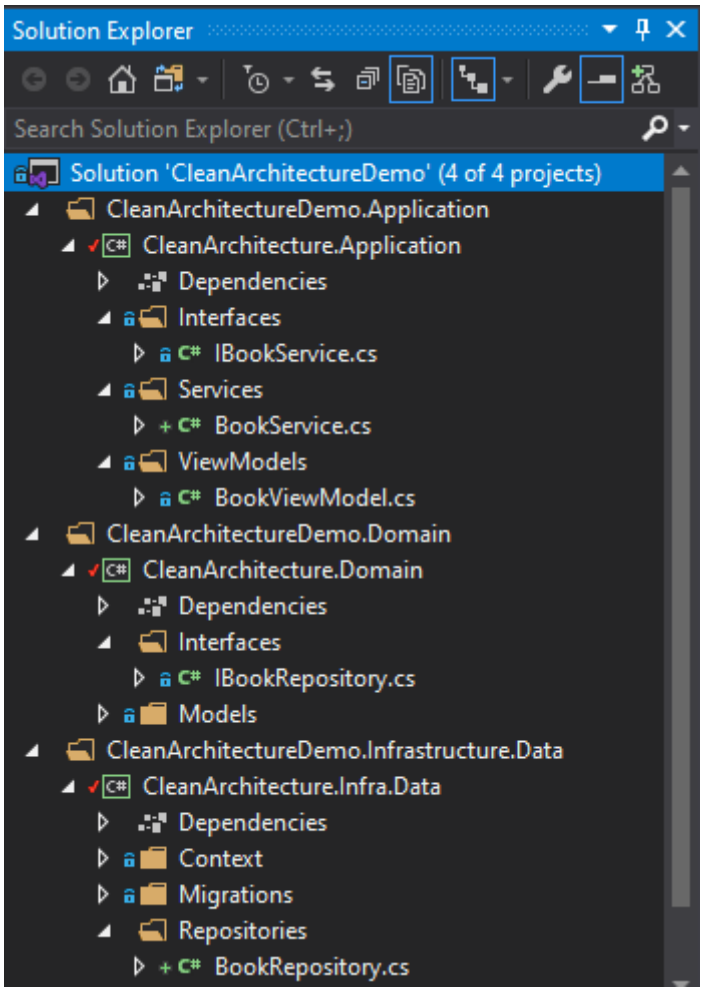
Okay so at this point, if the MVC project, or the **Presentation Layer (**which has no idea about the domain entity **Book)** says, “hey, I want a list of books!”, it needs to talk to the **BookService** (which we haven’t implemented yet, using **IBookService**), and **BookService** need to get it from **BookRepository** (which also we haven’t implemented yet, using **IBookRepository**)

So let’s implement them. First, **BookService.**

Under **CleanArchitecture.Application** project, under services folder, add a new class, **BookService.cs,** and inherit it from **IBookService.**

Now we need to inject the IBookRepository, Inject it as you would normally do dependency injection in .NET

Next the **BookRepository**. Under **CleanArchitecture.Infra.Data** project create a new folder named **Repositories,** under that create new class, **BookRepository**.



Just like we implemented the **BookService,** we have to implement the **BookRepository** from **IBookRepository**, then inject the database context, so we can talk to the database.

We did not implement the methods yet in **BookService and BookRepository**, so let’s go ahead and implement those methods.

First, go to **BookRepository,** and using the injected context, retrieve the books in the database.

Next, the **BookService.**

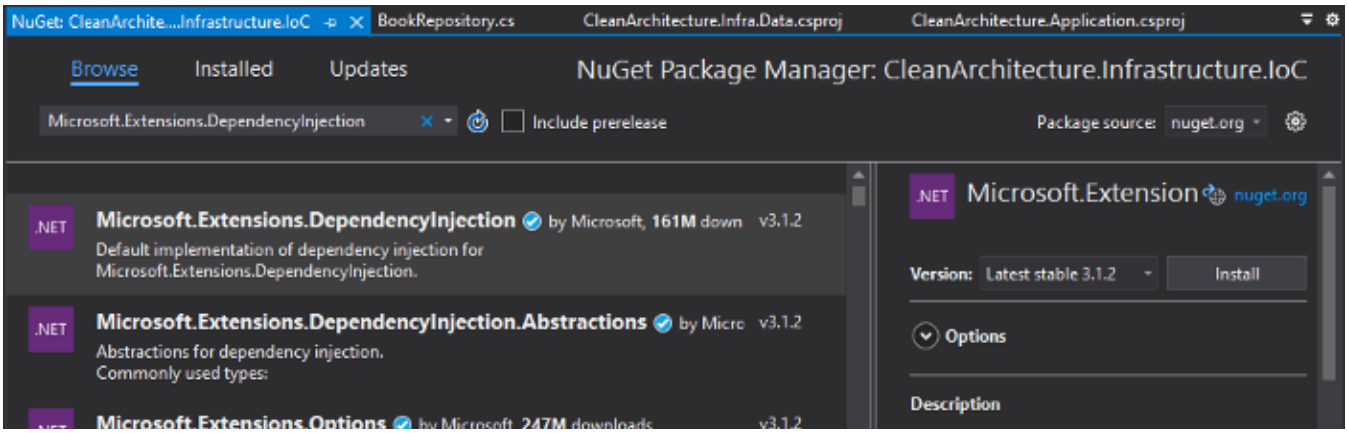
Next we need to look at the implementation of IoC project, which will help us to contain and separate the dependencies.

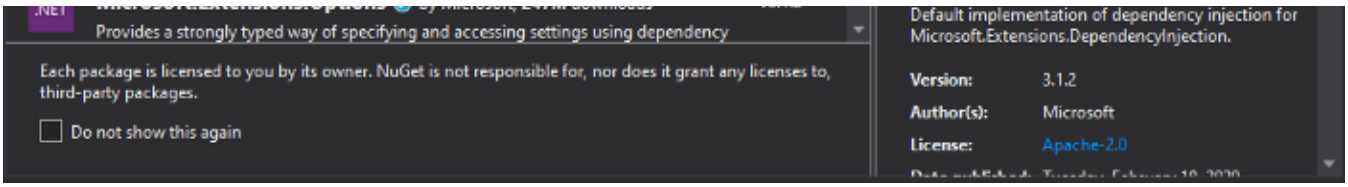
Inversion of Control

So under the **CleanArchitectureDemo.Infrastructure.IoC**, create a new .NET Core Class Library just like wee did in earlier projects named **CleanArchitecture.Infrastructure.IoC.** Get rid of the Class1.cs, right click on dependancies -> Manage NuGet Packages -> go to the browse tab and search for,

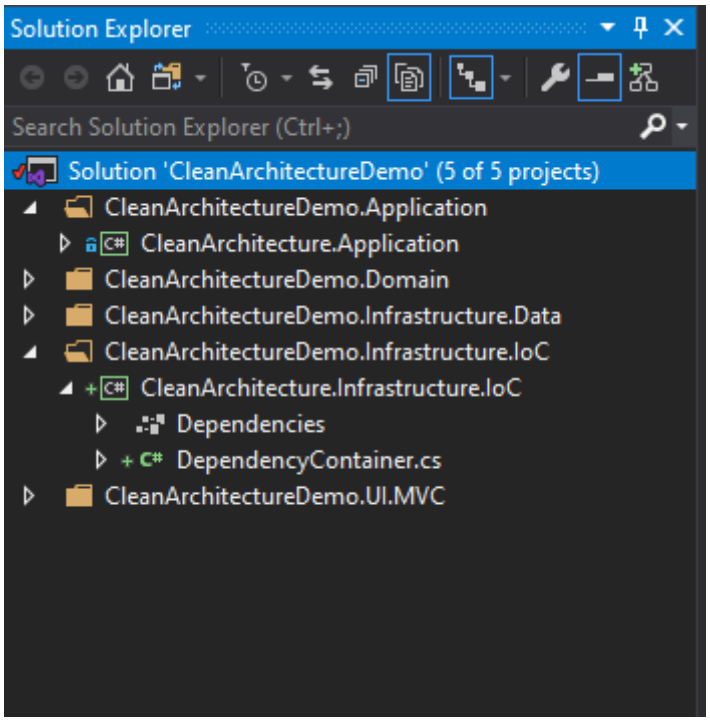
```
Microsoft.Extensions.DependencyInjection
```

Make sure you install the same version as the previous dependencies we installed and are stable versions.





Now let’s create the Dependency Container class,
under **CleanArchitecture.Infrastructure.IoC** project.



Notice how it connects our interfaces and their implementations from multiple projects into single point of reference. That is the purpose of IoC layer. Also, notice **AddScoped,** if you want to know the differences read more here.

Now we need to tell about this new services container to MVC project. Head over to **Startup.cs,** after the **Configure** method add following method.

Now inside the **ConfigureServices** method we need to call this method.

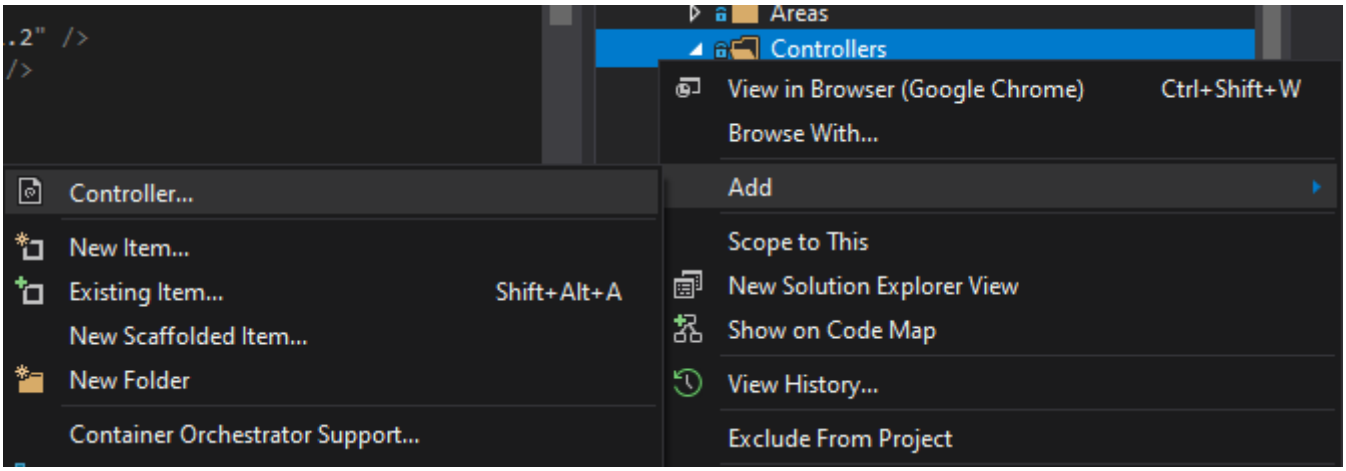
```
RegisterServices(services);
```

Full **Startup.cs** looks like this.

So now all our layers are ready. So next we will take a look at how to create the **Controller**s, using everything we discussed here and implement the UI.

First let's add some data to the database, since we are only implementing the **GetBooks** method, we need to add some data manually.

Once you have the data in the database, let’s create a controller in **CleanArchitecture.MVC** project, under **Controllers** folder.



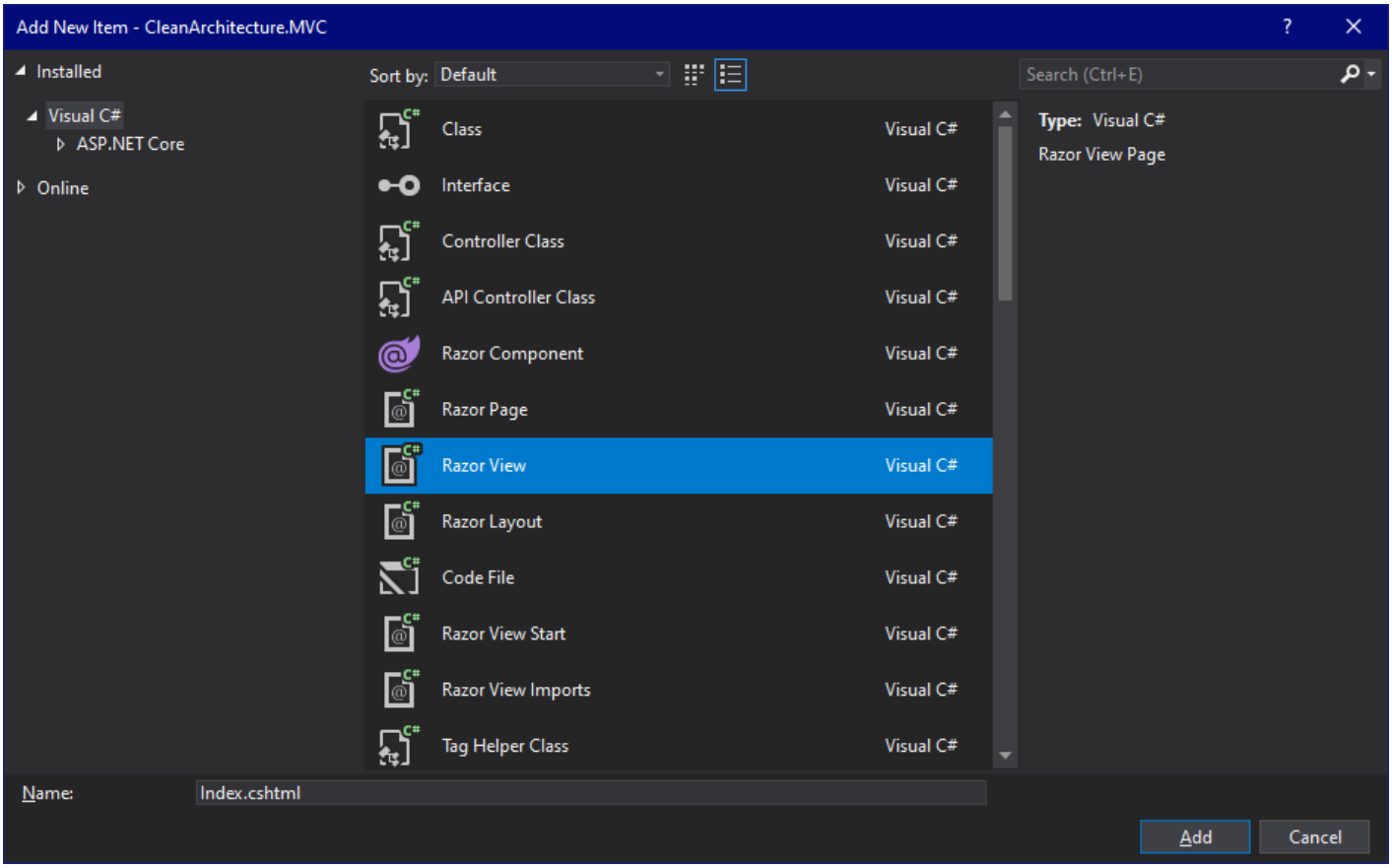
Right click Controllers folder -> Add -> Controller, name it as **BookController**following the MVC convention. Then we will inject the **BookService** to this controller as follows.

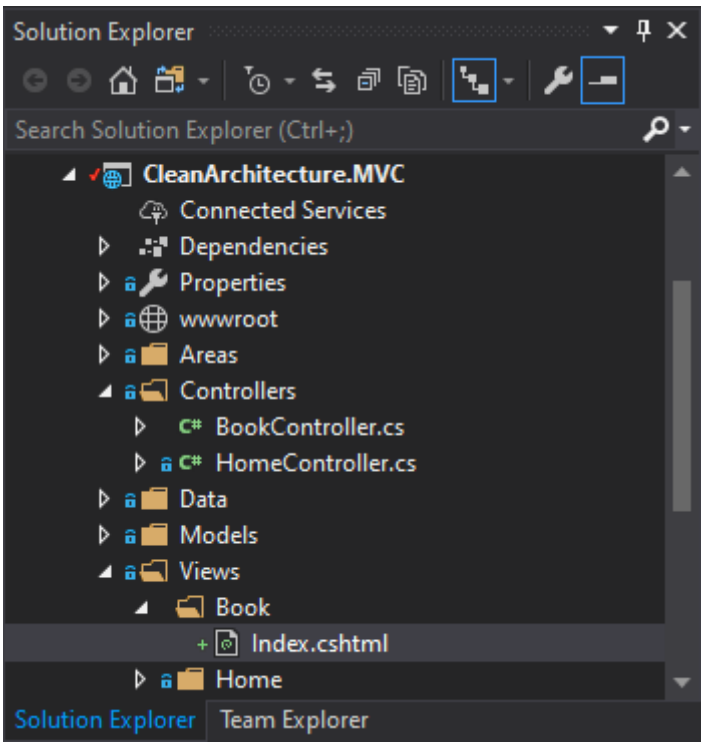
Notice we are referring to the IBookService in Application layer.

Now we will pass the BookViewModel to the view.

See how **clean** that looks. That is the beauty of the clean architecture. So now we have passed the BookViewModel to our View, let's go ahead and define that view.

Right click on **Views** folder, add new folder named **Book.** Right click **Book folder** and add a **Razor View** named Index.cshtml.

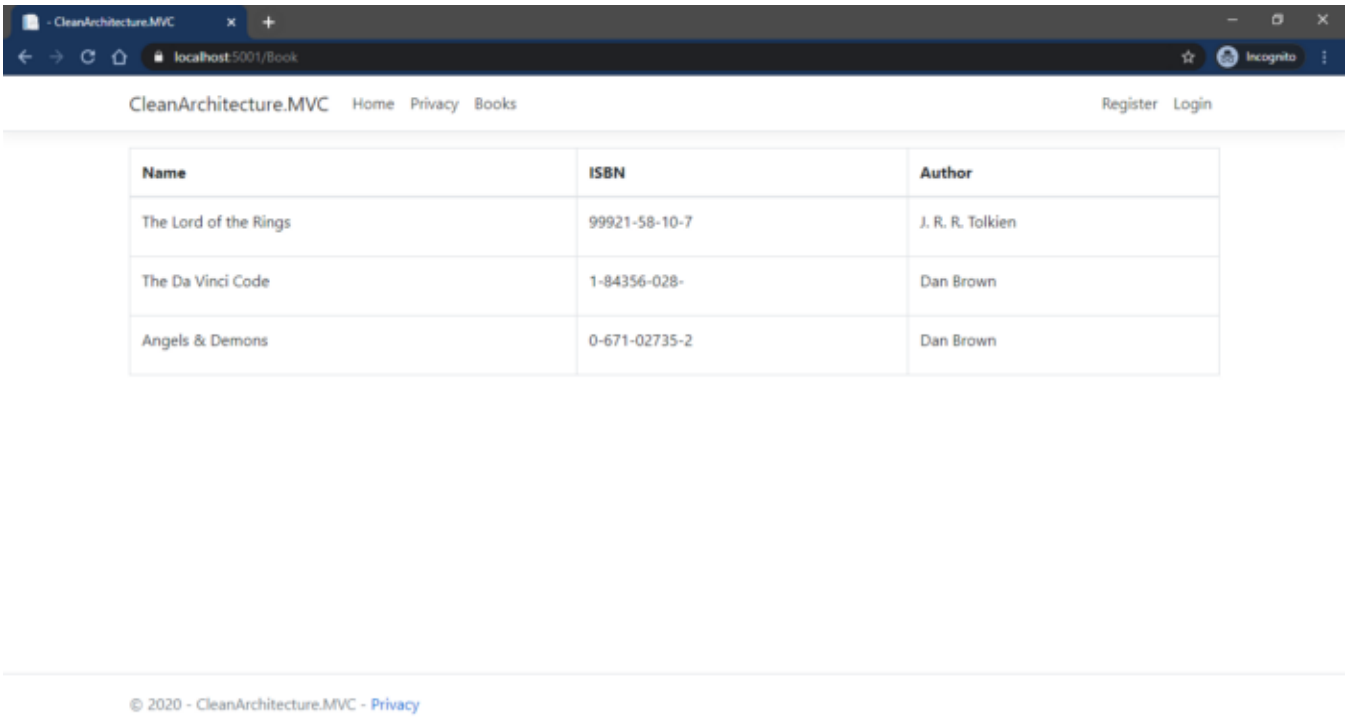




In the **Index.cshtml** add following code segment, which will take the BookViewModel and loop through every item (book) in the model and show it in a table.

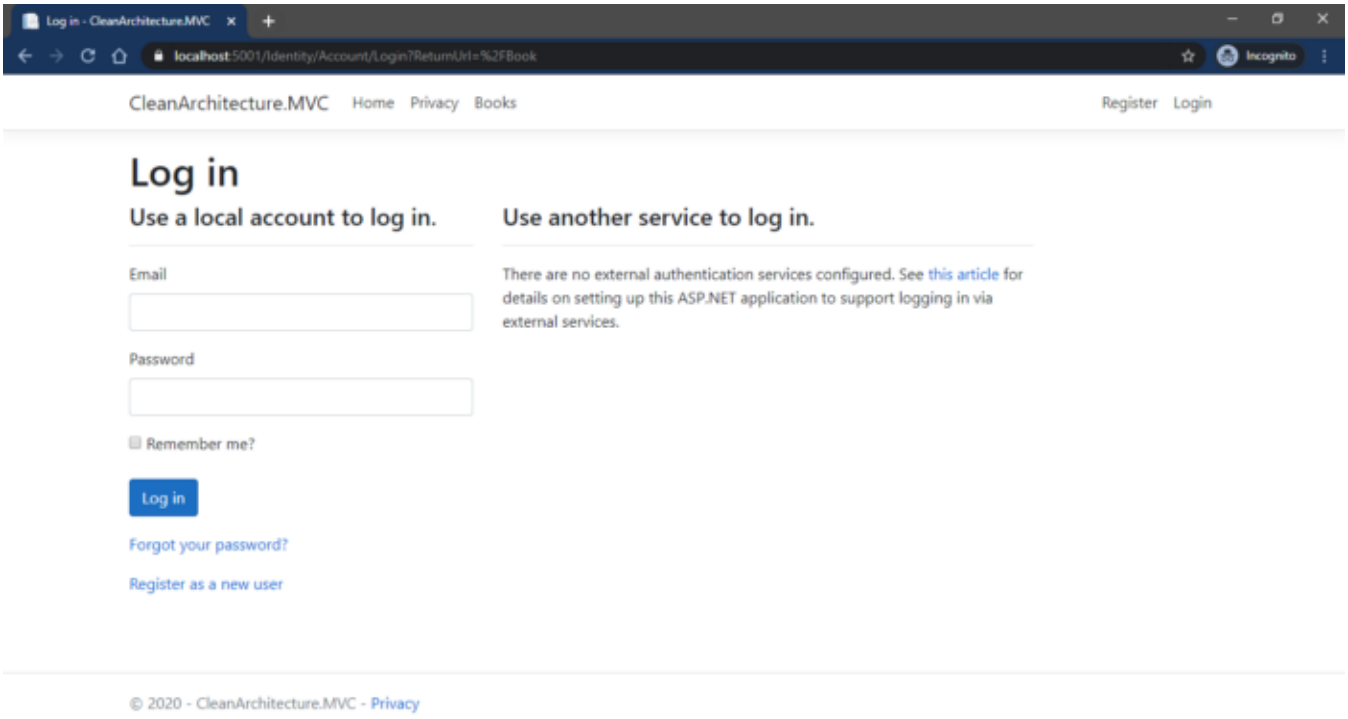
After that go to **_Layout.cshtml** under **Views/Shared** and add another navigation link to the View we just created. Notice it points to the **Book** controller and **Index** action.

Now you can run the MVC project and navigate to https://localhost:5001/Book to see the Books!



But we’re not logged in yet, so allowing users to view the data without authentication is a problem, which is easy to solve.

Add **[Authorize]** attribute to the Index() method to the controller itself and then try to navigate to https://localhost:5001/Book again.



You will be asked to log in. What you can also do is put breakpoints in the services, repositories we created and see how they work in the run time.

Well that's it, Thank you for coming this far. This was just an introductory guide on how to setup everything, there are tons of different ways to implement the clean architecture but with core concepts that I have discussed here. So next time when you come across more extensive tutorial you will not get lost.

If you want to go further from this I recommend you watch this video by JASON TAYLOR or read his amazing article on Clean Architecture.