

Building ASP.NET Core Web APIs with Clean Architecture

Published Sep 30, 2018 • Updated Mar 7, 2020



Last year I wrote [a post introducing clean architecture](#) and attempted to explain how its layered approach and separation of concerns can help overcome some common software design pitfalls enabling us to create testable, loosely-coupled code that is easier to maintain and extend.

In this post, we'll revisit Clean Architecture in the context of a somewhat more real-world example by using its principles to design and build an ASP.NET Core based Web API. Understanding these principles is critical for this guide and I won't be covering the basics from scratch so if you're new to Clean Architecture I recommend you check out [my previous post](#) or [Uncle Bob's](#) to get up to speed.

This guide also assumes knowledge of other topics like MVC, dependency injection and testing so if you run into something you're not familiar with please take a moment to familiarize yourself with any new concepts.

Get notified on new posts

Straight from me, no spam, no bullshit. Frequent, helpful, email-only content.

Subscribe

A Story of Layers and Dependencies

At its absolute core, Clean Architecture is really about organizing our code into layers with a very explicit rule governing how those layers may interact.



Hi, I'm Mark Macneil — a .net guy who spends most days making software in friendly Halifax, NS.



The WYSIWYG Editor built to scale and developed in open source.

ADS VIA CARBON

Elsewhere

- [GitHub](#)
- [LinkedIn](#)
- [Email](#)

Related

[Better Software Design wi...](#)

Recent

- [Build an Authenticated Gr...](#)
- [Build an Authenticated Gr...](#)
- [Build an Authenticated Gr...](#)
- [Build an Authenticated Gr...](#)
- [User Authentication and L...](#)
- [Painless Integration Test...](#)
- [JWT Authentication Flow w...](#)
- [Building ASP.NET Core Web...](#)
- [Building a GraphQL API wi...](#)
- [User Authentication with ...](#)

Archive

[2020 \[2\]](#)

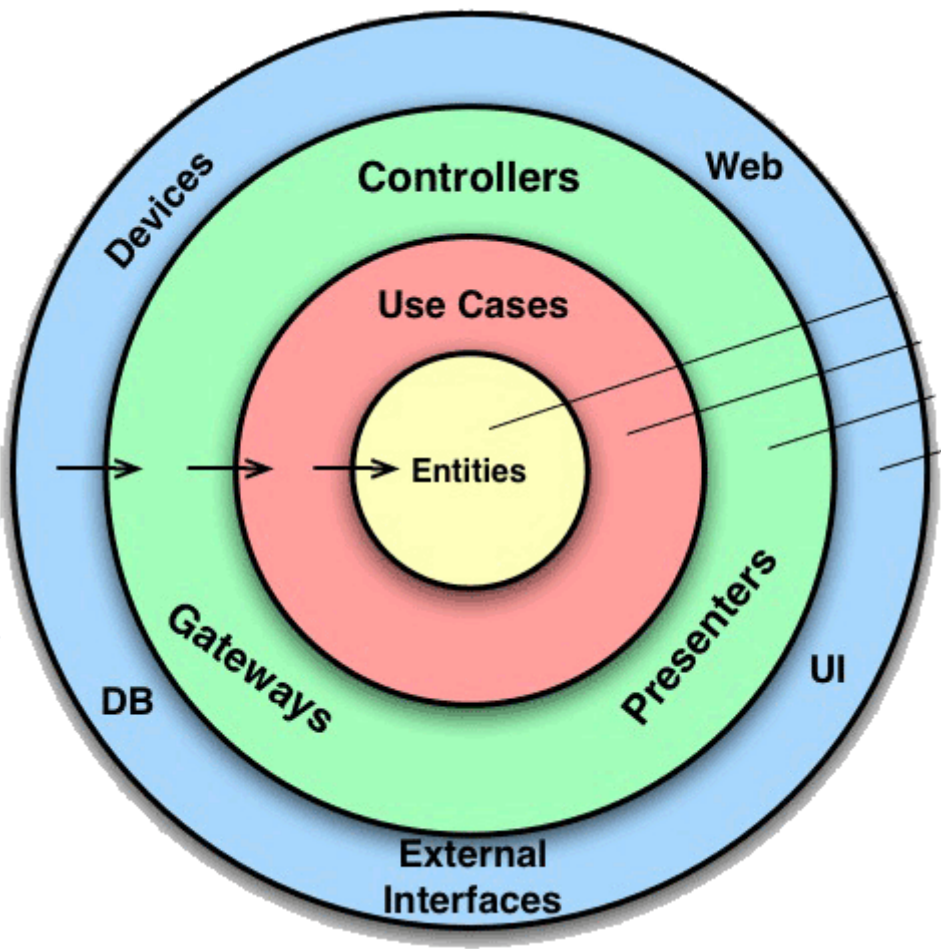
Get notified on new posts

Straight from me, no spam, no bullshit. Frequent, helpful, email-only content.

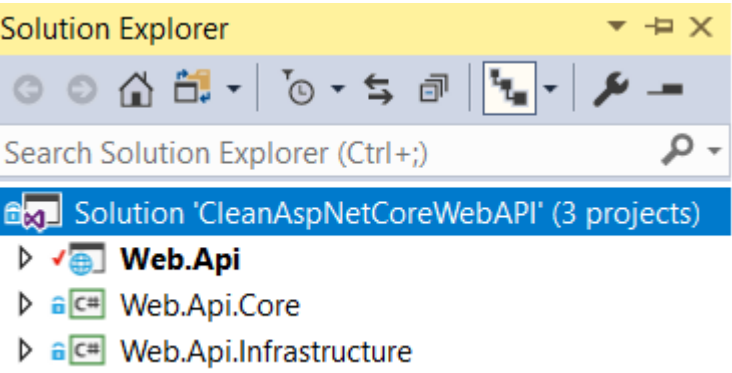
Subscribe

The overriding rule that makes this architecture work is The Dependency Rule. This rule says that source code dependencies can only point inwards. Nothing in an inner circle can know anything at all about something in an outer circle.

— Uncle Bob



With that in mind, to get started; I've fleshed out a project structure that should represent each of the logical layers in the diagram.



Let's break it down a little further:

Web.Api - Maps to the layers that hold the *Web*, *UI* and *Presenter* concerns. In the context of our API, this means it accepts input in the form of http requests over the network (e.g., GET/POST/etc.) and returns its output as content formatted as JSON/HTML/XML, etc. The Presenters contain .NET framework-specific references and types, so they also live here as per *The Dependency Rule* we don't want to pass any of them inward.

Web.Api.Core - Maps to the layers that hold the *Use Case* and *Entity* concerns and is also where our *External Interfaces* get defined. These innermost layers contain our domain objects and business rules. The code in this layer is mostly pure C# - no network connections, databases, etc. allowed. Interfaces represent those dependencies, and their implementations get injected into our use cases as we'll see shortly.

Web.Api.Infrastructure - Maps to the layers that hold the *Database* and *Gateway* concerns. In here, we define data entities, database access (typically in the shape of repositories), integrations with other network services, caches, etc. This project/layer contains the physical implementation of the interfaces defined in our domain layer.

Get notified on new posts X

Straight from me, no spam, no bullshit.
Frequent, helpful, email-only content.

Subscribe

Alright, we now have a skeletal solution spun up that looks to support the high-level layers requested by Clean Architecture - cool! 😊

Using Tests to Guide Development (TDD)

With our project foundation in place, we're ready to get down to brass tacks and write some interesting code. To start, we're going to build out our use cases "from the outside in" by defining them first and then using tests to iteratively tease out and implement only the bits of functionality required to pass them - *red-green-refactor* - this is the essence of TDD.

Our API needs to provide the capability for new users to create an account from the client application so our first use case will be [drumroll]... *RegisterUserUseCase*.

I quickly fleshed out a test containing the things I think must happen to satisfy the use case. As you can see below, there's lots of red here at the moment, and this won't even build, but it's a terrific guide that tells us precisely what we must implement thereby *driving* our development and ensuring we only work on the bits required to get this from red to green.

[Fact]

0 references | mmacneil, 2 days ago | 1 author, 5 changes | 0 exceptions

```
public async void Can_Register_User()
{
    // arrange

    // 1. We need to store the user data somehow
    var userRepository = new UserRepository();

    // 2. The use case and star of this test
    var useCase = new RegisterUserUseCase(userRepository);

    // 3. The output port is the mechanism to pass response data from the use case to a Presenter
    // for final preparation to deliver back to the UI/web page/api response etc.
    var outputPort = new OutputPort();

    // act

    // 4. We need a request model to carry data into the use case from the upper layer (UI, Controller etc.)
    var response = await useCase.Handle(new RegisterUserRequest("firstName", "lastName", "me@domain.com", "userName", "password"),
                                         outputPort);

    // assert
    Assert.True(response);
}
```

RegisterUserUseCase is the object under test here and currently has a single dependency on **UserRepository** which we know will have the responsibility of persisting the user's identity. We're not sure exactly *how* that will happen at this point and that's fine, we'll come back to that shortly.

To implement **RegisterUserUseCase** I first created a simple interface named [IRegisterUserUseCase](#) which in turn implements [IUseCaseRequestHandler](#) that will define the shape of all of our use case classes.

Note on naming; you may see the term *Interactor* being used in place of *Use Case* or even combined, i.e., *Use Case Interactor* in other Clean Architecture implementations. As far as I can tell they serve the same purpose and both represent the classes where the business logic lives. I prefer *Use Case* as I find it's a little more intuitive.

Next, I took a crack and implemented the absolute bare minimum amount of code required by the use case at this point. My first cut looked like this:

Get notified on new posts X

Straight from me, no spam, no bullshit.
Frequent, helpful, email-only content.

Subscribe

4 references | mmacneil, 2 days ago | 1 author, 4 changes

```
public sealed class RegisterUserUseCase : IRegisterUserUseCase
{
    private readonly IUserRepository _userRepository;

    2 references | mmacneil, 4 days ago | 1 author, 1 change | 0 exceptions
    public RegisterUserUseCase(IUserRepository userRepository)
    {
        _userRepository = userRepository;
    }

    6 references | mmacneil, 2 days ago | 1 author, 4 changes | 0 exceptions
    public async Task<bool> Handle(RegisterUserRequest message, IOutputPort<RegisterUserResponse> outputPort)
    {
        var (success, id, errors) = await _userRepository.Create(new User(message.FirstName, message.LastName, message.Email,
                                                                    message.UserName), message.Password);

        outputPort.Handle(success ? new RegisterUserResponse(id, true) : new RegisterUserResponse(errors));
        return success;
    }
}
```

TDD is an iterative process (red->-green->refactor...repeat) and it will take a few iterations to refine this gradually.

Input and Output Ports

You'll notice the signature for the use case's `Handle()` method contains two parameters: `message` and `outputPort`. The `message` parameter is an *Input Port* whose sole responsibility is to carry the request model into the use case from the upper layer that triggers it (UI, controller etc.). [This class](#) is simply a lightweight DTO owned by the Core/Domain layer. I created it during the previous step when I set up the use case handling infrastructure.

The second, `outputPort` as you can probably guess is responsible for getting data out of the use case into a form suitable for its caller. The critical difference is that it needs to be an interface with at least one method available for our *Presenter* to implement. The physical implementation of the presenter lives in the outer layer and may contain UI/View/Framework specifics and dependencies that we don't want bleeding into our use cases.

In this approach, we use [DIP](#) to invert the control flow so rather than getting a return value with the response data from the use case we allow it to decide when and where it sends the data through the output port to the presenter. We could use a callback, but for our purposes, as you'll see shortly, our presenter creates an http response message that is returned by our controller. In the context of a REST API, this response is effectively the "UI" or at very least the output data to be displayed by whatever application might be consuming it. The [question of whether or not the use case should call the presenter](#) is a good one, and the answer is likely whichever is most viable for your solution.

The `IOutputPort` contract is pretty simple; it exposes a single method accepting a generic use case response model it will transform into the final format required by the outer layer. We'll see an example implementation shortly.

```
public interface IOutputPort<in TUseCaseResponse>
{
    void Handle(TUseCaseResponse response);
}
```

Next, I created a contract representing `IUserRepository` with a single method for creating new users.

```
public interface IUserRepository
{
    Task<CreateUserResponse> Create(User user, string password);
}
```

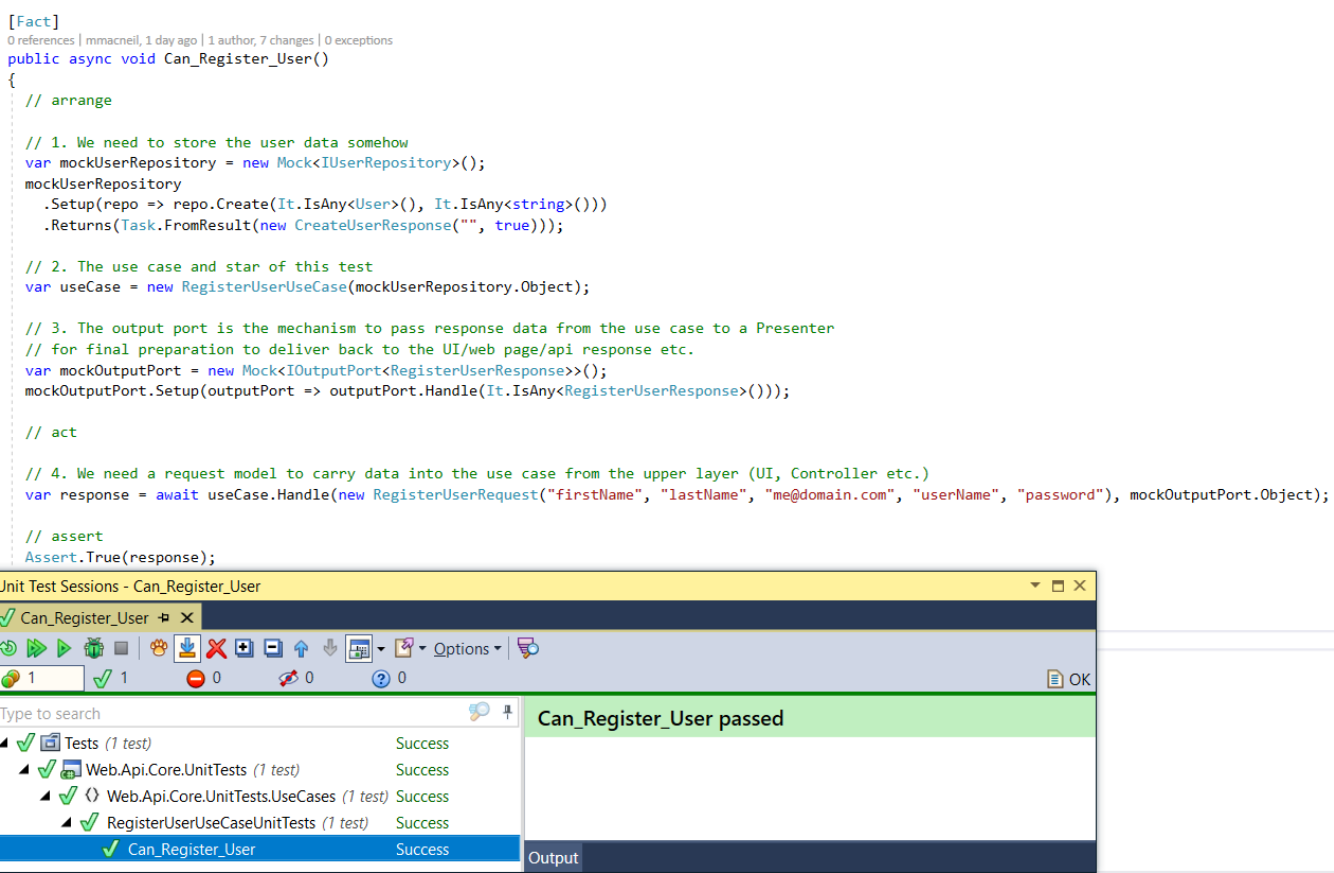
Get notified on new posts X

Straight from me, no spam, no bullshit.
Frequent, helpful, email-only content.

Subscribe

With a few more iterations the interfaces are complete, and the use case has no more red squiggles. I moved back to the unit test and touched it up by creating mocks for both collaborators using the suitably named [Mog](#) mocking library.

Things are looking much better - my project compiles, I can run [the test](#) and it passes! We have a thin slice of functionality working to register new users.



The Data Layer

Most programs require data to make them somewhat interesting or valuable, and our API is no different. Moving up a layer we'll find the data layer. This layer contains our data access and persistence concerns and any frameworks, drivers or dependencies they require. In this layer, we're moving away from the blissful ignorance we had in the domain layer as we now begin to make decisions and lay down concrete implementations of the various data providers our application requires. Persistence and data access are handled using Sql Server and Entity Framework Core. All data access related dependencies and code live in [Web.Api.Infrastructure](#).

I provisioned the infrastructure project with nuget packages for Entity Framework Core and the [ASP.NET Core Identity Provider](#) along with code for the [UserRepository](#) and a data entity to represent our [User](#).

The implementation of **UserRepository** at this point is pretty simple. It depends on **UserManager** which we get out of the box from the identity provider and **IMapper** which comes from [AutoMapper](#) and enables us to elegantly map domain, data and dto objects across the layers of our application while saving us from littering our project with repetitive boilerplate code. **UserManager** provides APIs for managing users in the membership database.

In our **Create()** method we start off by mapping our domain user (currently a tad [anemic](#)) to the data entity **AppUser** and then call the **CreateAsync()** method on **_userManager** to do the deed of creating the user in the database and returning the result. All of the work to validate the input parameters and create the user is handled by **UserManager** and everything is abstracted behind the **IUserRepository** interface. In this sense, it is acting mostly as a [facade](#) to keep these implementation details encapsulated in the *Infrastructure* layer.

Get notified on new posts X

Straight from me, no spam, no bullshit.
Frequent, helpful, email-only content.

Subscribe

Subscribe

```
internal sealed class UserRepository : IUserRepository
{
    private readonly UserManager<AppUser> _userManager;
    private readonly IMapper _mapper;

    public UserRepository(UserManager<AppUser> userManager, IMapper mapper)
    {
        _userManager = userManager;
        _mapper = mapper;
    }

    public async Task<CreateUserResponse> Create(User user, string password)
    {
        var appUser = _mapper.Map<AppUser>(user);
        var identityResult = await _userManager.CreateAsync(appUser, password);
        return new CreateUserResponse(appUser.Id, identityResult.Succeeded,
identityResult.Succeeded ? null : identityResult.Errors.Select(e => new Error(e.Code,
e.Description)));
    }
}
```

Using Migrations to Spin Up a Database

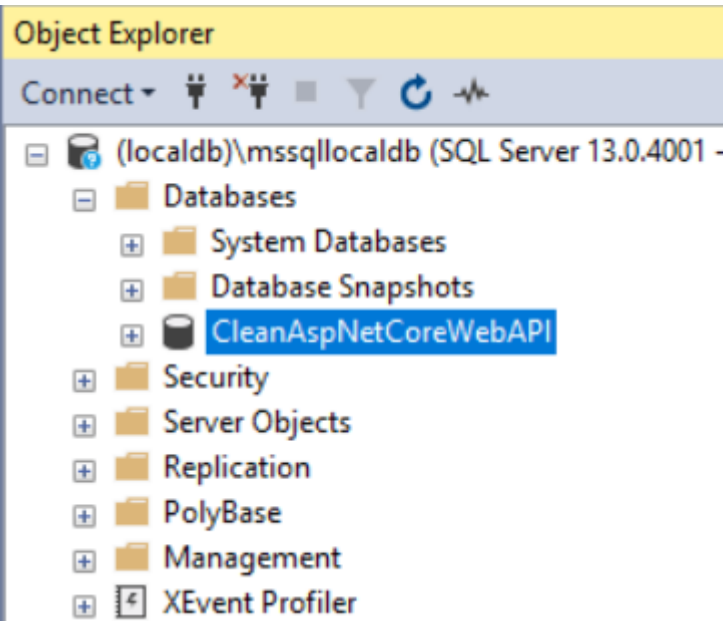
We're ready to use Entity Framework's migrations to create the physical sql database by running two simple dotnet CLI commands from within the infrastructure project folder.

```
Web.Api.Infrastructure>dotnet ef migrations add initial
```

```
Web.Api.Infrastructure>dotnet ef database update
```

These commands are run on the infrastructure class library project because we've encapsulated all our entity framework-related code there. It's common to have these dependencies/responsibilities live in the root Web API project however because we're seeking a higher level of decoupling in our Clean Architecture we've pulled them out into the infrastructure layer.

After running these commands, a new database appears in my sql server localdb instance. Note, you'll need to ensure you've installed *Sql Server Express LocalDB* under *Individual Components* in the Visual Studio installer.



Dependency Injection

At this point, I wired up [Autofac](#) and registered the required components and services in the Web API project's [Startup](#) and within individual [module](#) files in each class library project. Modules provide a very nice way to organize your dependencies on a per-project basis when working with Autofac. It also plays a pivotal role in our Clean Architecture by enforcing the *Dependency Inversion Principle* across the application

Get notified on new posts

Straight from me, no spam, no bullshit.
Frequent, helpful, email-only content.

Subscribe

Cancel

The Presentation Layer

This layer is where the rubber effectively meets the road by composing the different components across our architecture into a single, cohesive unit - i.e., our application. Here, we find concerns related to GUIs, web pages, devices, etc. It also contains our *Presenters* which, as mentioned, are responsible for formatting the response data from our use cases into a convenient format for delivery to whatever human interface our user happens to trigger it from, e.g., a web page, mobile app, microwave, etc.

In the context of a REST API, this layer is relatively simple as there is no GUI, view state or complex user interactions to handle. The request/response semantics of REST means we're mostly just delivering data back to the user. Our presenters, in this case, will construct http messages containing the data and status of the request and return these as the response from our Web API controller operations.

[RegisterUserPresenter](#) implements **IOutputPort** with all of its work happening in **Handle()**. You'll remember this is the method the use case triggers. From there, we're just building a regular MVC **ContentResult** and in this case, directly storing the serialized results from the use case and setting the appropriate **HttpStatusCode** based on the result of the use case.

```
public sealed class RegisterUserPresenter : IOutputPort<RegisterUserResponse>
{
    public JsonResult ContentResult { get; }

    public RegisterUserPresenter()
    {
        ContentResult = new JsonResult();
    }

    public void Handle(RegisterUserResponse response)
    {
        ContentResult.StatusCode = (int)(response.Success ? HttpStatusCode.OK :
HttpStatusCode.BadRequest);
        ContentResult.Content = JsonSerializer.SerializeObject(response);
    }
}
```

Setting Up the Controller

The piece at the very furthest edge of our Clean Architecture is the API controller. The controller will receive input in the form of http requests, trigger our use case and respond with some meaningful message based on the outcome.

I added a new [AccountsController](#) with a single POST action to receive new user registration requests.

```
// POST api/accounts
[HttpPost]
public async Task<ActionResult> Post([FromBody] Models.Request.RegisterUserRequest
request)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    await _registerUserUseCase.Handle(new
RegisterUserRequest(request.FirstName,request.LastName,request.Email,
request.UserName,request.Password), _registerUserPresenter);
    return _registerUserPresenter.ContentResult;
}
```

Get notified on new posts X

Straight from me, no spam, no bullshit.
Frequent, helpful, email-only content.

Subscribe

Not much happening here, we validate the input model (the user's information) using [FluentValidation](#) implicitly and if that looks good, trigger the use case and return the `ContentResult` the presenter has generated based on the results of the operation.

Unit Testing the Controller

I created a new [Web.Api.UnitTests](#) project and applied some TDD to iteratively build out the required bits for the controller action with tests from [AccountsControllerUnitTests](#). It's useful to test how our controller behaves based on the validity of the inputs and validate its response based on the result of the operation it performs.

Manual End-to-End Testing with Swagger

Now the moment of truth. I'd like to test the API for real by submitting requests to it externally in the same fashion as its consuming applications will. To help me out, in the Web API project I installed and configured [swagger](#) tooling which includes a very handy UI for testing and exploring our API. With that in place, I fired up the project and tested the `/api/accounts` endpoint by submitting a user registration request via the Swagger UI.

Description

Example Value | Model

```
{  "firstName": "Mark",  "lastName": "Macneil",  "email": "mark@fullstackmark.com",  "userName": "mmacneil",  "password": "Pa$$W0rd1"}}
```

Cancel

Parameter content type

application/json-patch+json

I got back a successful response in the Swagger console - so far so good.

Server response	
Code	Details
200	<div>Response body<div><pre>{ "id": "f33e837a-577e-48c7-84f9-6ef73412105e", "success": true}</pre></div></div>

Finally, I queried the database to confirm the associated record was successfully inserted into the `AspNetUsers` table. 🐶

Get notified on new posts

X

Straight from me, no spam, no bullshit.
Frequent, helpful, email-only content.

Get notified

Subscribe

SQLQuery1.sql - (I...\Mark.MacNeil (53))*

select * from AspNetUsers

100 %

Results

Messages

	Id	UserName	NormalizedUserName	Email
1	f33e837a-577e-48c7-84f9-6ef73412105e	mmacneil	MMACNEIL	mark@fullstackmark.com

Implementing a Login Use Case

With the user registration API feeling pretty solid I went ahead and implemented a login use case to authenticate user credentials and issue JWT tokens. The process was nearly identical to the one we followed for registration, so I'm going to save some repetitive keystrokes by not documenting it here but if you're interested please check out [the code](#) or give it a spin in swagger using the **/api/auth/login** endpoint.

Wrapping Up

If you made it this far - you rock! 🙌

I hope you can take some value from this guide with an understanding of how Clean Architecture can help produce well-designed, expressive code that is more loosely-coupled and testable.

As we saw, its approach is very flexible in that it can be used with or without other design activities like TDD and DDD and plays nicely with existing application structures like MVC, MVVM, etc.

Putting Clean Architecture or any design pattern into practice successfully takes skill and judgment. Like anything, there's a learning curve associated, and teams must understand and accept its strict rules and conventions to apply it successfully.

Finally, this is just my interpretation of Clean Architecture using a simple example so your mileage may vary applying this to your projects in the real world. As always, if you have any feedback, questions or suggestions for improvement, please let me know below in the comments.

[Source code here](#)

Get notified on new posts

Straight from me, no spam, no bullshit. Frequent, helpful, email-only content.

Subscribe

Related Posts

2.9k Shares

f

in

Get notified on new posts

Straight from me, no spam, no bullshit. Frequent, helpful, email-only content.

Subscribe



Better Software Design with Clean Architecture

Jun 30, 2017

[Read more](#)

38 Comments

FullStack Mark

1

Login

Recommend 11

Tweet

Share

Sort by Best

Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

 **Farhan Ali** • a year ago

Hi!


This is great Article! I found this amazing as a starter for an API. It would be really helpful if you add all methods (CRUD) for a single Entity to explain how exactly a beginner should go along this clean architecture.

6 ^ | v • Reply • Share ›

 **Archie** ➔ Farhan Ali • a year ago

correct, Exactly.


^ | v 1 • Reply • Share ›

 **Eirik Rasmussen** • 2 years ago • edited

I'm curious, would you implement a separate class for every single usecase?, for example for Get requests for querying the database by id, name, get all.

It seems that this implementation is tailored for POST requests where the request includes a message body. For example each Get request would require an empty or very simple class with one or two properties.

4 ^ | v • Reply • Share ›

 **Mark Macneil** Mod ➔ Eirik Rasmussen • 2 years ago

I think for a single entity versus a list, ie. `GetAll()` you would likely have separate use case classes as those are almost certainly unique functions in your application. If it's just a straight fetch of an entity or list, you may not need a use case at all as it may just be overhead in

Get notified on new posts X

Straight from me, no spam, no bullshit.
Frequent, helpful, email-only content.

Get notified



Patrick Harris → Mark Macneil
• a year ago • edited

I switched to using Specification Pattern. I have two functions (i.e. GetCustomer and GetCustomers). A specification class is passed in which has the expression to use for the query. See a Nuget package called NSpecification.

1 ^ | v • Reply • Share ›



Nitin Sontakke • 6 months ago

Hello Mark,

First thank you very much for putting this together.

A real quick question, primarily to have the clarity. Assuming there will be 100 methods across 20 entities to be implemented using this approach, will there be 100 use cases, 20 repositories, 100 request and 100 response classes?

3 ^ | v • Reply • Share ›



Mark Macneil Mod → Nitin Sontakke • 6 months ago

Hey Nitin, no - there really isn't a formula to arrive at the number of use cases. Ideally, you'll use them to exercise the domain logic captured in your entities to carry out the core features in your application. So, a small and simple application may only have a handful of use cases, while a larger, more complex application will naturally have a higher number.

1 ^ | v • Reply • Share ›



Antonio Valentini • 2 years ago

Hi Mark, great article!

I'm recently approaching to .net core and I was wondering, .net core has its own built in dependency injection system. Can this be enough to handle all the applications needs in order to avoit using Autofac? Maybe i'm just losing something important about autofac...

3 ^ | v • Reply • Share ›



Mark Macneil Mod → Antonio Valentini • 2 years ago

Absolutely Antonio - you can use the out of the box dotnet core service container to handle the majority of your web application's IoC requirements in a near identical fashion to Autofac.

3 ^ | v 1 • Reply • Share ›



PG → Mark Macneil • a year ago • edited

So why Autofac is used? What's the main benefit of using this instead native?

1 ^ | v • Reply • Share ›



Luka • a year ago

Hi Mark,

Thanks for a really comprehensive article and example - really appreciate it when bloggers take the time to explain their thinking clearly.

I just have a couple of questions:

1. The use of a 'Repository' pattern when using Entity Framework is often considered redundant since EF itself implements both the Repository and Unit of Work patterns. Given this, would you feel comfortable simply renaming your 'UserRepository' to 'UserFacade', and doing the same to all your other 'Repositories' in your production code?
2. You state that this architecture plays well with DDD but DDD stipulates that "validations, calculations, business rules...should be in a domain object". So how do you see the UseCase objects conforming to this guideline?

4 ^ | v 1 • Reply • Share ›

Get notified on new posts X

Straight from me, no spam, no bullshit.
Frequent, helpful, email-only content.

Get notified

Subscribe

it is true that EF provides the Repository and the UoW, the clean architecture must not know how the infrastructure is implemented, in that sense the core knows that there will be an implementation of the repository but don't know how the implementation is. This will allow you to even change the whole implementation of the infrastructure and keep the code working.

1 ^ | v • Reply • Share ›



Sergey Sagan • 2 years ago

Ugh you need to stop using underscores they are so out, instead you should be using `this` in your constructors

2 ^ | v 1 • Reply • Share ›



Mark Macneil Mod ➔ **Sergey Sagan** • 2 years ago • edited

Thanks for the callout [@Sergey Sagan](#). Prefixing private vars with an underscore is a fairly common convention in C# and one I happen to adopt. `this` in this context, while indeed explicit is redundant and not required by the compiler so in the spirit of fewer keystrokes and less code I omit those.

4 ^ | v • Reply • Share ›



Kevon Houghton ➔ **Mark Macneil** • 8 months ago

I agree on the dislike for `_` variables. I've taken to naming the method parameter differently from the private variable to avoid confusion.

Not sure what other people think of this, I'd be curious to know.

I would re-write it like this:

```
internal sealed class UserRepository :
    IUserRepository
{
    private readonly UserManager<appuser>
        userManager;
    private readonly IMapper mapper;

    public UserRepository(UserManager<appuser>
        injectedUserManager, IMapper injectedMapper)
```

[see more](#)

^ | v • Reply • Share ›



Patrick Harris ➔ **Mark Macneil** • a year ago • edited

Agreed. "this" is redundant.

^ | v • Reply • Share ›



Mohamed Asan Sali • 6 months ago

HI
Someone let me know shall i use this design pattern for ASP.NET MVC Web application core

1 ^ | v • Reply • Share ›



Mark Macneil Mod ➔ **Mohamed Asan Sali** • 6 months ago

Hi Mohamed, yes, absolutely. You could easily apply the same architecture to an ASP.NET Core MVC application.

^ | v • Reply • Share ›



Mohamed Asan Sali ➔ **Mark Macneil** • 5 months ago


Thanks Mark I started development but how to do the login and logout functionality and how to do the persistence. Please give me some

Get notified on new posts X

Straight from me, no spam, no bullshit.
Frequent, helpful, email-only content.

Get notified

Subscribe




Jonathan Meyer

→ Mohamed Asan Sali

• 5 months ago

<https://github.com/Sl1ver/w...>

^ | ▾ • Reply • Share ›



Cifroes


• 8 months ago

How would you leverage this architecture into a scenario where you create a REST API using clean architecture and then the company has a mobile app and a webapp that want to consume the same REST API?

I can see it working nicely if the webapp is a SPA that calls directly the REST API from the browser but I want to keep the webapp as a .NET MVC. Would that be 'weird' to have a .NET MVC calling the REST API?

Or a more generic question could be: an architecture for a company that wants to have a core 'business' services that are then shared by a mobile app and a web app?

1 ^ | ▾ • Reply • Share ›




Bob Archer

→ Cifroes

• 7 months ago

It's not weird at all... actually a pretty common implementation. Of course, the other option is to have two UI's in the project... The API UI and the MVC UI... MVC can call directly into the core just like the API does.

^ | ▾ • Reply • Share ›




Nicholas

• a year ago

Thanks for taking the time to write this articulate example. In principle I think this approach is a good one, but feel that implementing the output port and passing it into the use case as a "call back" is counter intuitive. It seems that for very basic scenarios you can encapsulate work into a single use case, but what about composing a use case from others which exist within the same application domain? Would you create X number of presenters to map or just return the data to be then passed into the next use case? I can see how this could work, but it just feels like it could get over complicated. Have you used this approach on a large project in it's absolute purity?

1 ^ | ▾ • Reply • Share ›




Abhilash Bandi

• 2 years ago

How about using CQRS to handle usecases?

1 ^ | ▾ • Reply • Share ›




Ivan Spahiyski

• 19 days ago

Hi Mark!

Thank you very much! Great article!

^ | ▾ • Reply • Share ›



Khalil Mansour

• 7 months ago

Hi, we are currently working on a project following this architecture but are having trouble with asynchronicity. For example, if a web service sends multiple individual requests within a short amount of time (ms intervals) to the same endpoint, let's say for adding something to a database, the response objects get mixed up and overwritten when they are returned to the caller. Is it because of the way the return is set up in the controllers ? A typical occurrence of this is when uploading file objects(ex. File1 to File5), sometimes the responses sent are mixed up and we'll notice that "File5" has overwritten some of them. When the web service is refreshed, everything seems fine which means that on the infrastructure side of things everything is fine. We're wondering if there's a know solution to this, thank you.

^ | ▾ • Reply • Share ›

Get notified on new posts

✕

Straight from me, no spam, no bullshit.
Frequent, helpful, email-only content.

Get notified

Subscribe



Hi Mark,

Really awesome to have such a good and elaborate example of clean architecture for a .NET Core API! I've got a few questions, though.

1: Why have two different entities for a user? I see that you have an AppUser that adds to IdentityUser, but also User, that gets mapped to and from.

2: There's a BaseEntity class in Infrastructure, but it's not used as base type for User, for example. In the dbContext, there's some casting involved to BaseEntity, but I'm not seeing any class being a BaseEntity. How does that work?

Thanks in advance!

^ | v • Reply • Share ›



plainionist • 9 months ago

Hi Mark,

great article about Clean Architecture and how to apply it to a real project!

In fact I started a blog series myself on "Implementing Clean Architecture" here: <http://www.plainionist.net/...>

If you find some time to read it your feedback would be highly appreciated!

thx

seb

^ | v • Reply • Share ›



Alex Kvitchastiy • a year ago

Hi Mark.

I have a question.

If I need to combine data from few useCases to one, what should I return from my controller ?

Can I create common presenter for them ?

^ | v • Reply • Share ›



Archie • a year ago

how did the table names in the initial migration generated? such as AspNetUserRoles, AspNetUserTokens etc. and all those indexes? since I couldn't find how those happened? because usally what ever are the properties of a class when placed inside a dbset it will be created as is.

How to add new table using migrations on this?

Get notified on new posts X

Straight from me, no spam, no bullshit.
Frequent, helpful, email-only content.

Get notified

Subscribe