Product Architecture Document

|  |  |
| --- | --- |
| Bert O’Neill | |
| Department: | Software Demo’s |
| Document Title: | Clean Architecture – Incorporating Repository Pattern |
| Author: | Bert O’Neill |
| Date: | 01-Jan-2023 |
| Version | 1.0 |
| SharePoint |  |
| Document Goals | Provide a detailed overview regarding how to implement a database first approach when using a Clean Architecture pattern. |

# Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Version | Description | Author |
| 01-Jan-2023 | 1.0 | Initial draft | Bert O’Neill |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Contents

[Revision History 1](#_Toc123404643)

[Introduction 3](#_Toc123404644)

[Purpose 3](#_Toc123404645)

[Scope 3](#_Toc123404646)

[Prerequisites 3](#_Toc123404647)

[Solution\Environment Setup 4](#_Toc123404648)

[Codebase 4](#_Toc123404649)

[SQL 4](#_Toc123404650)

[Visual Studio Extension 4](#_Toc123404651)

[Nuget Packages 4](#_Toc123404652)

[Quick Explanation of Clean Architecture 5](#_Toc123404653)

[Application Core: 6](#_Toc123404654)

[Infrastructure: 6](#_Toc123404655)

[UI Application: 6](#_Toc123404656)

[Visual Studio - Solution Structure 7](#_Toc123404657)

[Blazor Application Running 8](#_Toc123404658)

[Add Existing Database Entities and DBContext 9](#_Toc123404659)

[Entities 9](#_Toc123404660)

[DBContext 11](#_Toc123404661)

[Connection String (API Project) 13](#_Toc123404662)

[Clean Architecture – Our Solution’s Main Layers 14](#_Toc123404663)

[Domain (Class Library) 14](#_Toc123404664)

[Application - Use Case\Business Logic (Class Library) 14](#_Toc123404665)

[Infrastructure (Class Library) 14](#_Toc123404666)

[Application Layer – Interfaces 17](#_Toc123404667)

[DTO’s 18](#_Toc123404668)

[Swagger Enabled (Launch Settings API) 19](#_Toc123404669)

[Polly Retry Policy 20](#_Toc123404670)

[Testing Polly 21](#_Toc123404671)

[Logging (Server Side) 22](#_Toc123404672)

[Unit Tests – MSTest\Faker\Bogus 23](#_Toc123404673)

# Introduction

This document provides a detailed overview on how to incorporate a Repository Pattern into a Clean Architecture Pattern. Touching on as many common aspects you would have in your existing MVC pattern (Logging, Unit Testing, DI, EF Core etc.).

## Purpose

The Product Architecture Document (PAD) provides a comprehensive architectural overview on how to integrate a database first approach into a Clean Architecture approach – what components are involved and how they are related to each other.

## Scope

The scope of this PAD is to convey the concepts needed to produce a Clean Architecture pattern with a Repository pattern.

# Prerequisites

* An understanding of the MVC or DDD architecture patterns
* An understanding of Database First approach (Repository Pattern)
* Knowledge of SQL Server – running SQL scripts
* Knowledge of .Net Core\6
* SQL Server (inc. SSMS) and Visual Studio installed (free+ editions)
* Basic knowledge of MSTest (unit testing)

# Solution\Environment Setup

## Codebase

Clone the code-repo using this link <https://github.com/Bert0Neill/CleanArchitectureDemo.git>. The SQL script needed to create your database with seed data is also included in this clone (a folder within the Visual Studio solution).

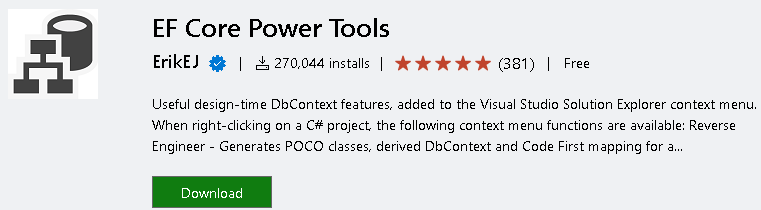
## SQL

Or you can use the SQL script attached below to generate your database and data:



## Visual Studio Extension

Download and install the *EF Core Power Tools* extension for Visual Studio from here -<https://marketplace.visualstudio.com/items?itemName=ErikEJ.EFCorePowerTools>



## Nuget Packages

To make the solution as realistic as possible, I have used the following packages and components that you would currently use with your existing architecture pattern – which have been incorporated into the solution projects, where they are used.

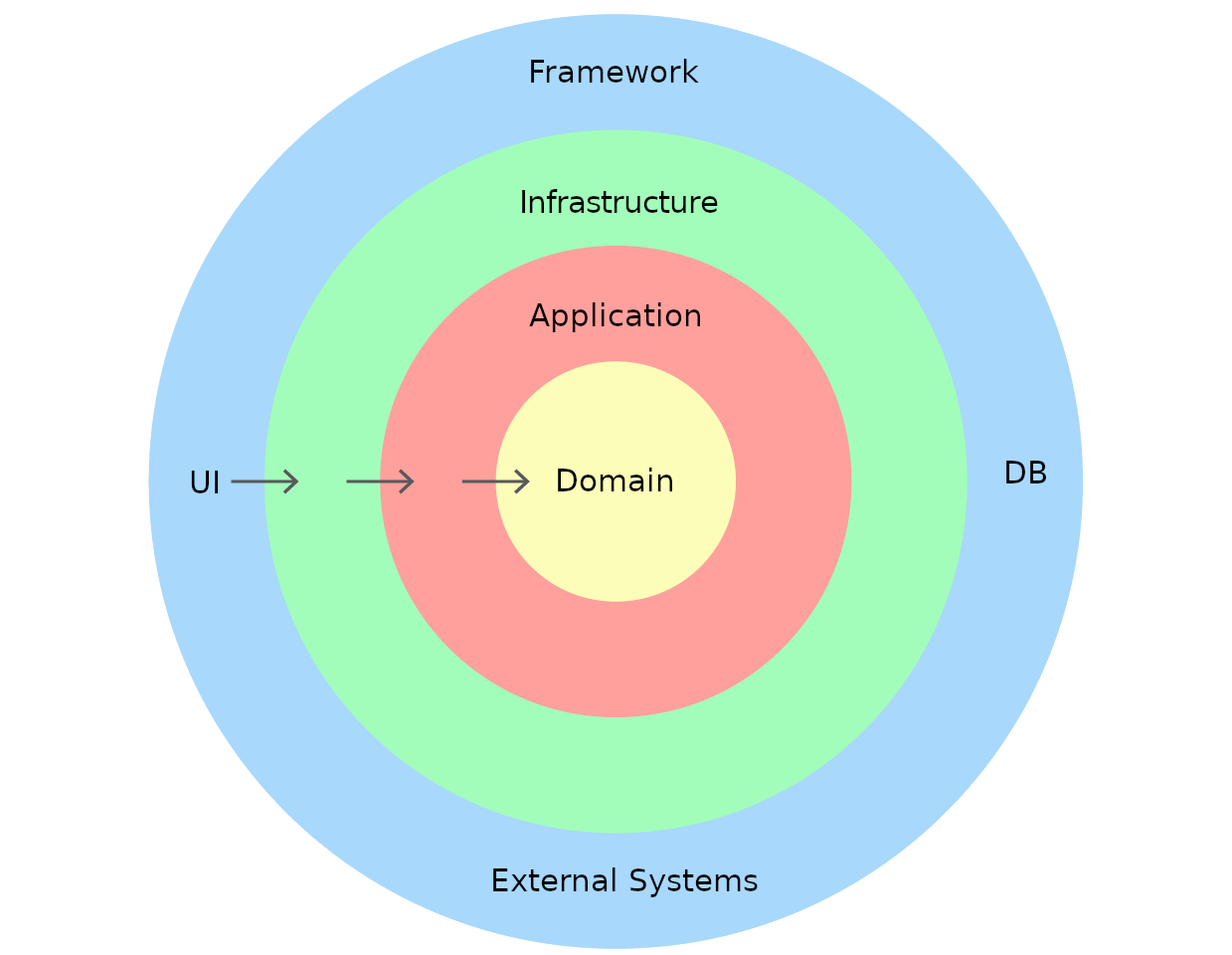
* MSTest
* Faker\Bogus
* MOQ
* IHttpFactory
* Http Polly (API retries)
* EF Core
* SQL Server
* EF Power Core Tools
* Middleware (Exception Handling)
* Logging (file based)
* GuardRails
* Blazor WASM
* Benchmark.Net
* AutoMapper
* Swagger UI

# Quick Explanation of Clean Architecture

The concept of a Clean Architecture pattern has been around for over a decade and initially conceived by Robert Martin (better known as [*Uncle Bob*](https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html)). The keyword from Uncle Bob is *Interchangeable.* In the image below, everything on the blue circle is interchangeable, for e.g. the UI can be swapped out from Angular to React, or the database can be converted from Oracle to MySQL, and nothing in the underlying layers need to change.

The concept of having all your interfaces (Infrastructure and Application) in one project makes it easier to Unit Test and mock.

But the main rational behind Clean Architecture, is that MVC doesn’t scale or allow for the same loose coupling of the layers. In Clean Architecture, the dependency is inward facing, only (this satisfies DI from SOLID principal). In MVC the Model View acts as the UI and Controller layer in one, this can get very large and difficult to test (because of the tight coupling). MVC has served the software industry for over 20 years, but the industry wants a new leaner architecture pattern, for the next 20 years – one that is scalable\interchangeable\decoupled.



## Application Core (Domain and Application):

* Application Core is a top layer or parent layer which will not depend on any other layer. So other layers like Infrastructure or UI depend on the 'Application' core.
* Application Core contains 'Entites', 'Interfaces', 'BusinessLogics', etc.
* So while creating projects if we want we can split 'Application Core' into 2 separate projects like *'Domain' and 'Application'*. So 'Domain' project mostly contains 'Entites'(table entities) then 'Application' project contains 'DTO's', 'Interfaces', 'BusinessLogics',etc. So the 'Application' project depends on the 'Domain'. So 'Domain' project can be sharable with other projects as well since it is the parent of layers. But please remember this splitting is only optional.

## Infrastructure:

Infrastructure deals with 'DataBases', 'External API Calls', 'Cache', etc. Basically, infrastructure deals with all external resources. Infrastructure depends on the 'Interface' inside of the 'Application Core'. Because of the dependency inversion, our 'Application Core' will be loosely coupled which is easy to test.

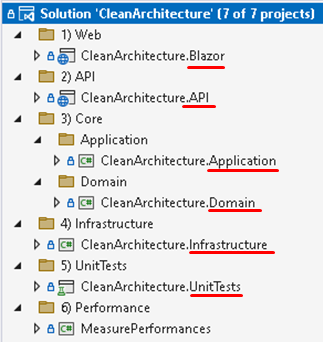
## UI Application:

UI Application consumes the 'Application Core' to produce the results. In a real-time scenario UI Application never depends on the infrastructure layer, but we have to reference the infrastructure layer into the UI project in the case to register the services dependency injection. So UI project should not use any code of the infrastructure layer other than dependency injection.

# Visual Studio - Solution Structure

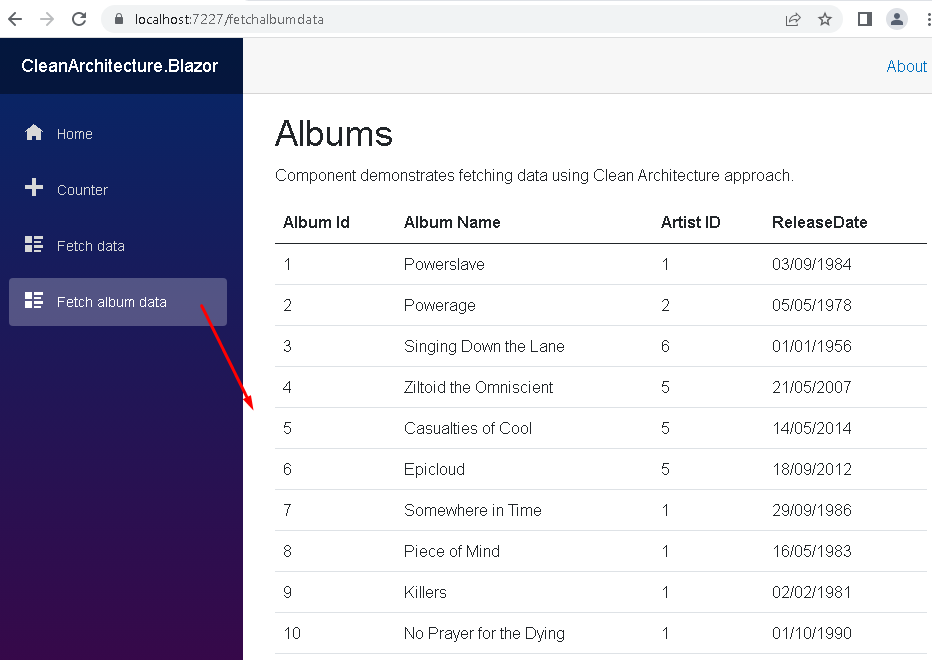
The solution is split up into *solution folders*, within each folder, there is a project (there can be multiple projects per layer depending on the size or complexity, thus splitting a layer into multiple projects may make more sense, maintenance wise):

* CleanArchitecture.Domain - 'Application Core' - A class library template project.
* CleanArchitecture.Application - 'Application Core' - A class library template project.
* CleanArchitecture.Infrastructure - 'Infrastructure' - A class library template project.
* CleanArchitecture.Api - 'API' - A web API template project.
* CleanArchitecture.Blazor- 'Web UI' - A web UI template project.
* CleanArchitecture.UnitTests- 'Unit Test' - A class library template project.



# Blazor Application Running

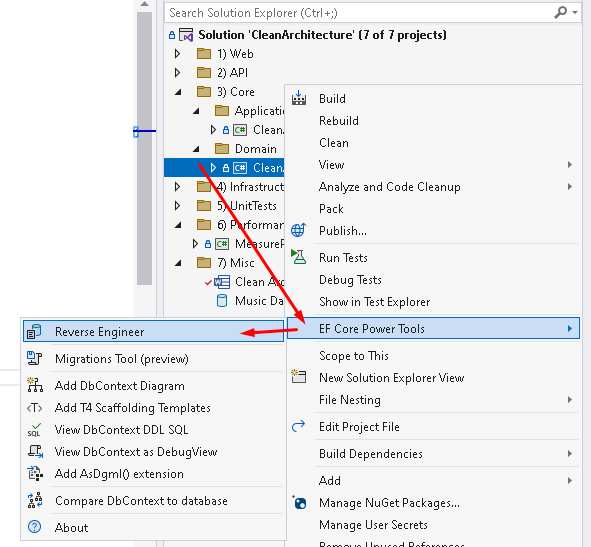
The image below display the final UI screen, which is a call from the client web application, to the API controller, which in turn calls the Application service (use cases\business logic), which calls the respective Infrastructure class to perform external tasks.



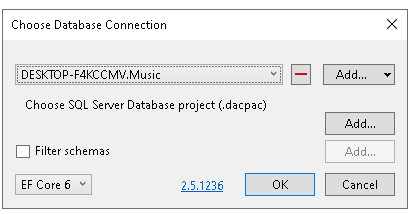
# Add Existing Database Entities and DBContext

## Entities

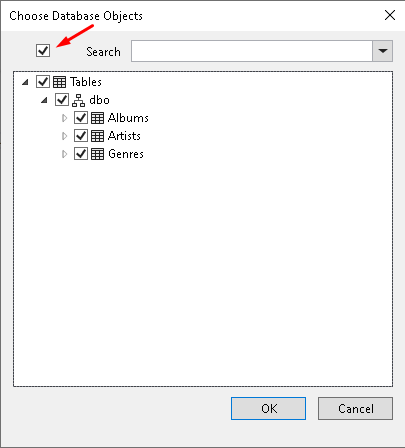
The database entities will be saved within the Domain project itself, right click the Domain project and select *EF Core Power Tools* from context menu, then select *Reverse Engineer*.



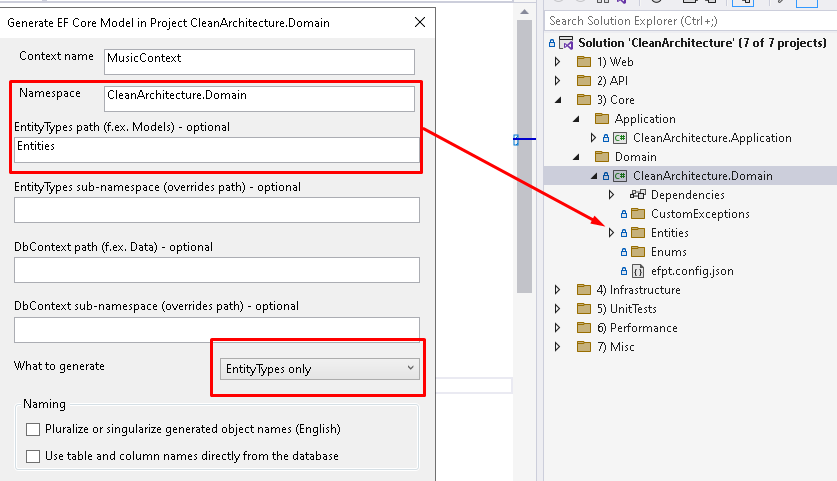
You will then be prompted to select the database you wish to reverse engineer the database models.



Next, you can select the appropriate database tables to generate the entities for:



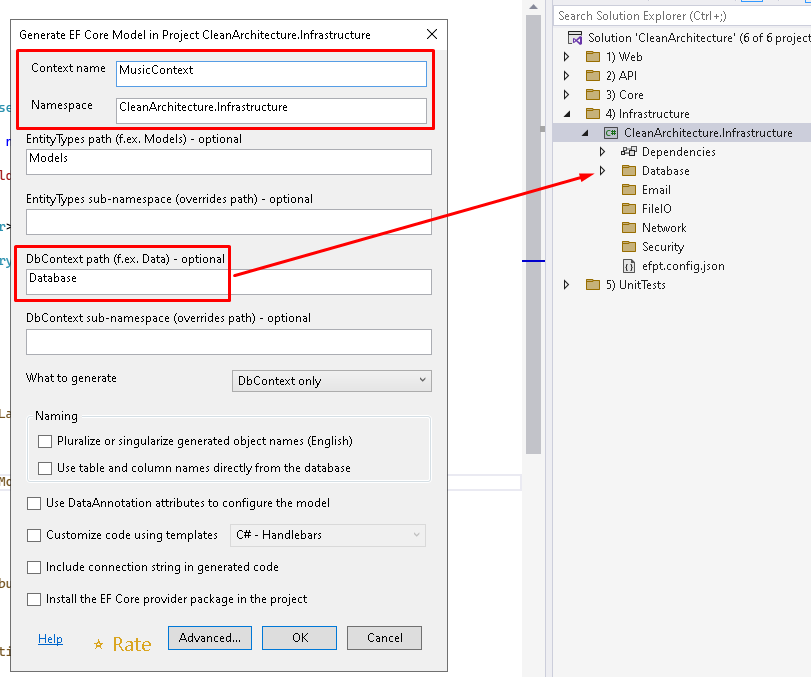
Ensure to select *EntityTypes Only* from the *What to generate* dropdown, as we only wish to create the entities and not the DBContext in the Domain project. Enter the path to create the entities, the namespace should be correct as you right clicked on the Domain project.



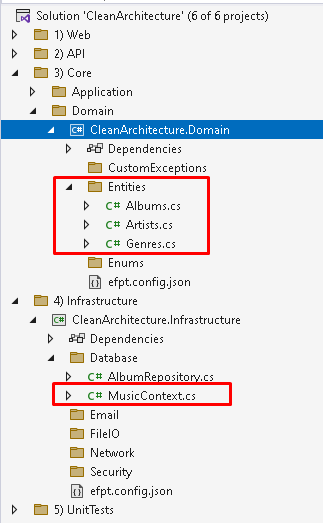
## DBContext

Right click on the Infrastructure project and again select *EF Core Power Tool*, to *Reverse Engineer* the DBContext (into the Application project).

Ensure to select DbContext Only from the dropdown and enter the path to where you would like the DBContext class generated:



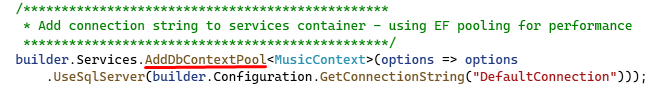
This will then generate the structure below for the entities and the DBContext, in the appropriate projects:



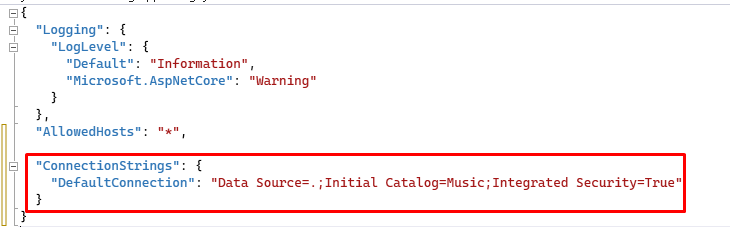
## Connection String (API Project)

The connection string is placed in the *Programs class of the* ***API project***, because we are consuming the Application project, which in turn will need the Infrastructure classes injected that will use the DBContext.

Add a reference to the DBContext (MusicContext from the Infrastructure project) into your Program.cs class. If you are prompted to add a reference to the Infrastructure project – accept this, as it is only allowing DI to be perform into the Application classes.



An entry in the Appsettings file to the connection string:

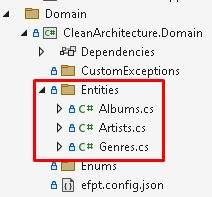


# Clean Architecture – Our Solution’s Main Layers

The Clean Architecture approach is very diversifiable, in that it can cater for any design pattern you wish to use (Factory with Decorator for e.g.), but in our example, I am using the common repository pattern approach.

## Domain (Class Library)

The domain project will host enterprise wide entities\models, custom Exceptions, Enumerations etc., but it has no dependencies, no project or class reference, no business logic etc.



*Tip* – The term ‘Entity’ comes from the SQL Server ‘Id**entity**’ property – meaning that the entity must have a primary key.

## Application - Use Case\Business Logic (Class Library)

Consider these services as your application’s business logic\use case layer, a pass through from the UI to the Domain and performing the necessary application logic needed for your solution. The Application layer will consume the Domain models, but use the Infrastructure layer to communicate with the outside world, thus using those results to perform its business logic (slicing and dicing results from multiple Infrastructure calls, which in turn gets passed back to the client as (for e.g.) a DTO).

**NB:** Only Domain is added as reference project.

## Infrastructure (Class Library)

These classes are responsible for external infrastructure communications like database storage, file system, external systems/APIs/Services and so on. We can add more class libraries under this project folder for external plug-in’s or SDK’s.

In essence, the Infrastructure layer is technically not needed, as you could design an application that doesn’t interact with the outside world, and does all it own business logic, this would certainly be the exception to the norm!

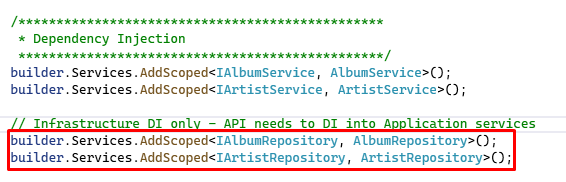
It is the outermost layer of the system and should have no knowledge of the inner layers.

**NB:** Application Class is added as reference.

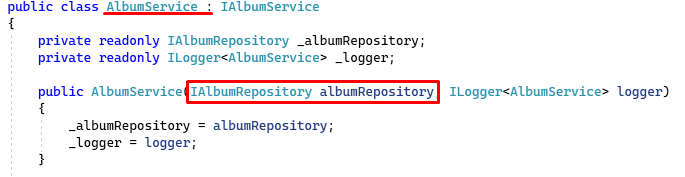
**NB:** The UI Application never depends on the infrastructure layer, but we have to reference the infrastructure layer into the UI project in the case to register the services dependency injection. So UI project should not use any code of the infrastructure layer other than dependency injection.

The Application layer needs the Infrastructure classes injected, so this has to come from the Web API layer (the UI layer makes no use of Infrastructure, only to DI into Application layer).

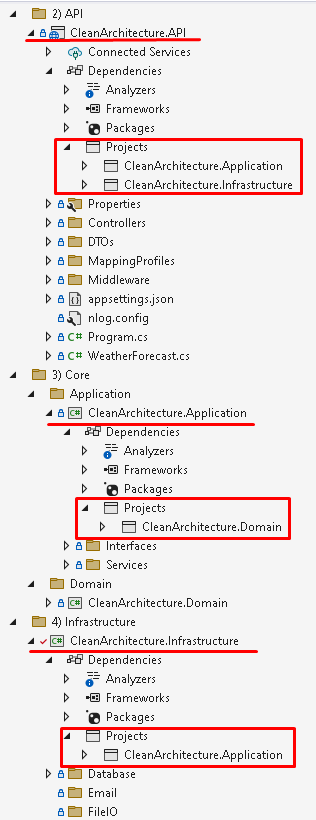
Web UI Programs.cs configuring Infrastructure DI:



Application service consuming the injected Infrastructure classes:



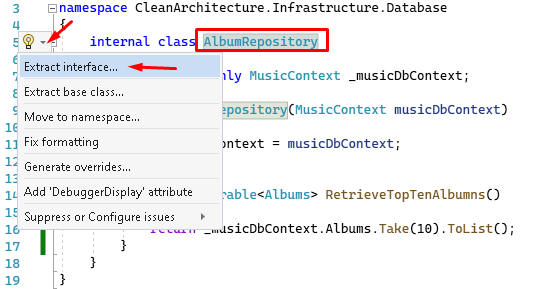
In the image below, you can see the inward reference structure between the projects. The API project must reference the Application layer so that it can makes calls to the various business logic services (it references the Infrastructure for DI purposes only).



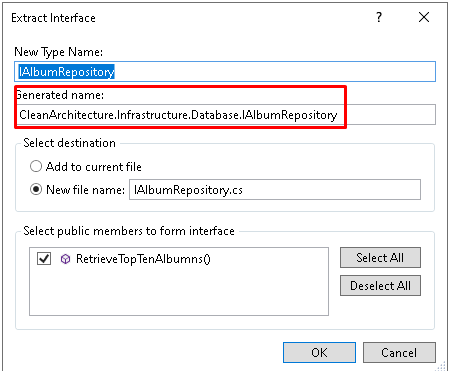
# Application Layer – Interfaces

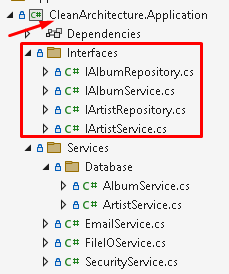
One of the main characteristics of a Clean Architecture approach is having ALL your interfaces in one place, namely the Application project. This indicates that only this layer needs to be mocked put when generating unit tests for other parts of the solution. All you’re mocking in one place makes for easier unit testing.

So, below I am generating an interface for the Infrastructure Repository class (in the Infrastructure project) and then moving it to the Application project.



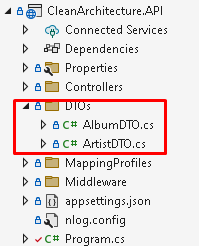
**NB:** It’s going to create the interface by default in the Infrastructure class, we will just move it after it has been crearted.





# DTO’s

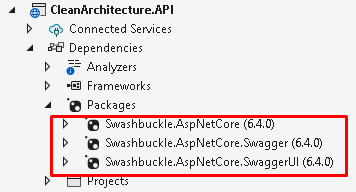
Although DTO’s are in themselves entities, and thus you may be thinking that they should go into the Domain project, the rule of thumb with DTO’s in any project, is that they should go into the project where they are (most) used. For web related solutions, this would be the API layer – as data coming from the client will be a DTO and data going to the client will be mapped entities (into DTO’s).



# Swagger Enabled (Launch Settings API)

After adding the Swagger packages to the API project, you can configure you project as follows:





Alter your Programs.cs class to take account of your Swagger settings:

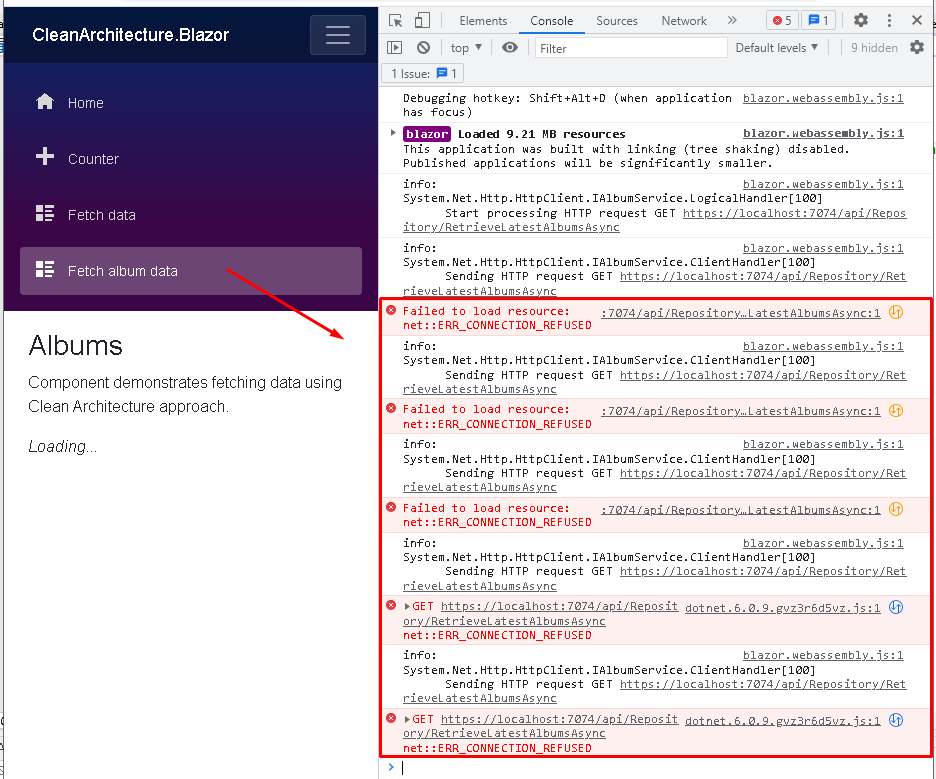


## Polly Retry Policy

Polly is configured within the Blazor UI application Program.cs file, below I am using the appsettings.json file to pickup various settings (retry count and base URL) and assigning them to each API call – below I am stating to retry the API call a max of 5 times, plus it jitters the retries, so as not to overwhelm the server with a call right away.

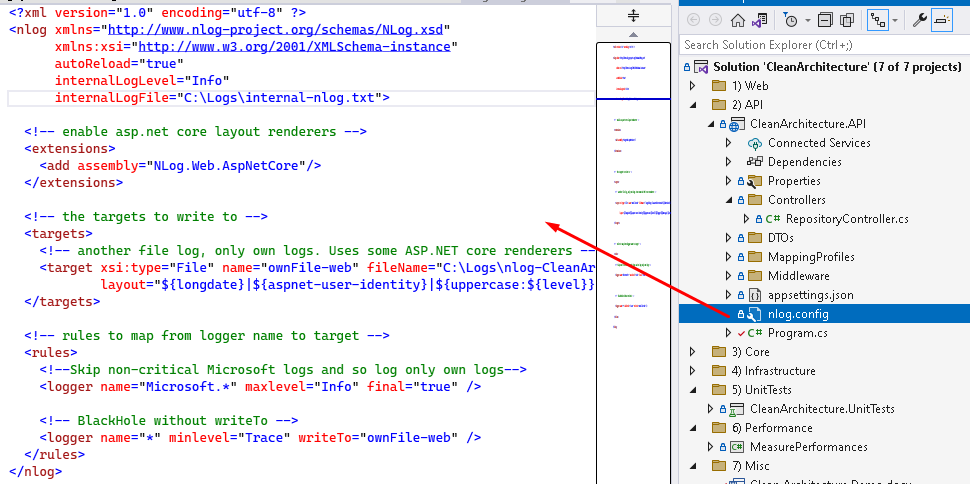


## Testing Polly

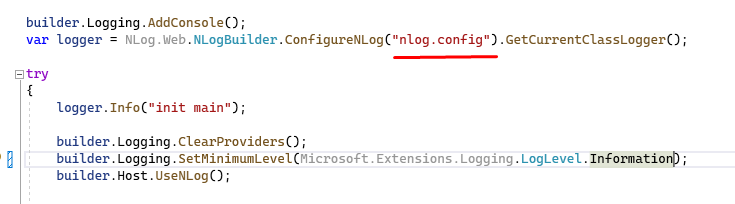
Only run the Blazor application (not in conjunction with the API), then call the Fetch Alnum data option again from the left hand menu – notice that it will retry 5 times (the 5 coming from our appsetting) before giving up.

# Logging (Server Side)

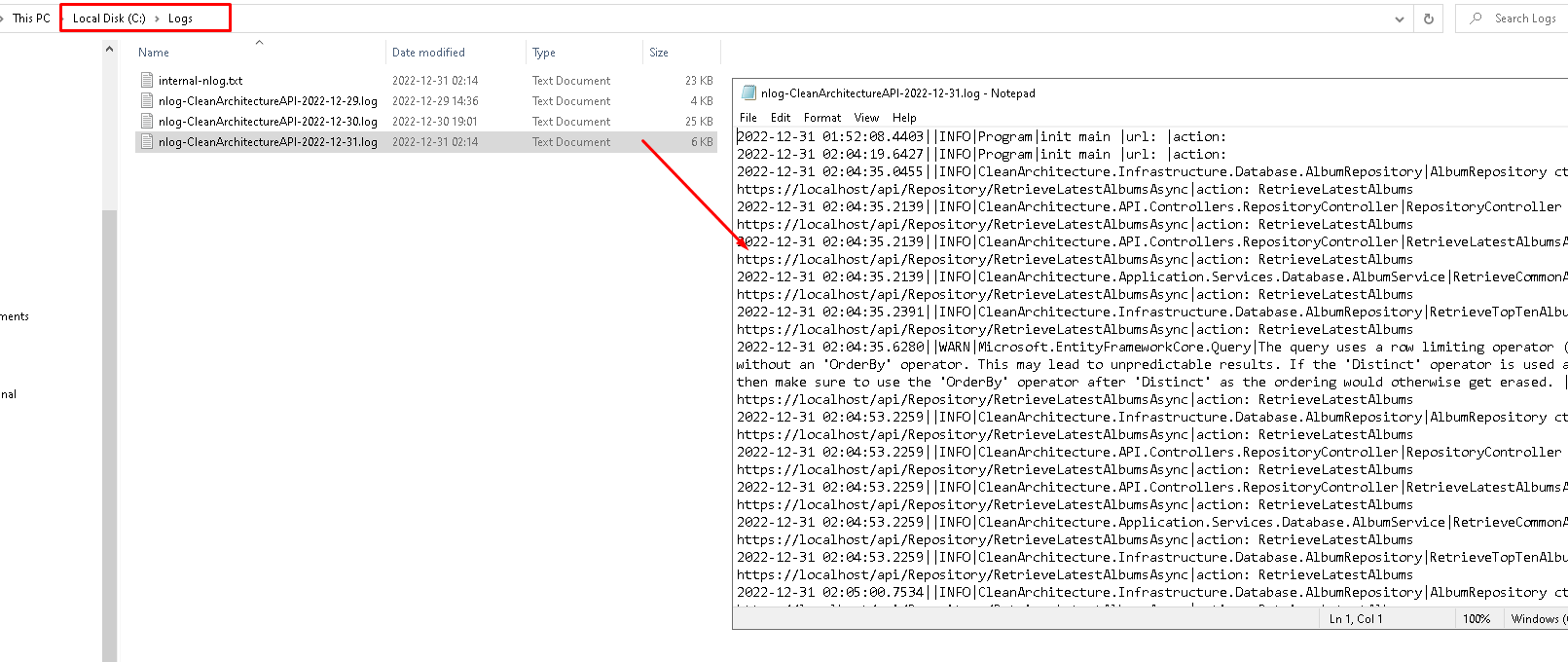
Configure NLog to how you wish to log your data; below I am just setting it up in the API project to log everything to a log file (using the nlog.file on the project root)



Configuring NLog within Programs.cs:



Example output:



# Unit Tests – MSTest\Faker\Bogus

Below, I am configuring the unit tests with mock data (using the Bogus package), where I am faking the data returned from the Repository calls, thus I can unit test the business logic\use cases held within the Application project – you can view the unit test code in the solution project:

