Product Architecture Document

|  |  |
| --- | --- |
| Bert O’Neill | |
| Department: | Software Demo’s |
| Document Title: | Clean Architecture – 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](#_Toc123345559)

[Introduction 3](#_Toc123345560)

[Purpose 3](#_Toc123345561)

[Scope 3](#_Toc123345562)

[Prerequisites 3](#_Toc123345563)

[Testing Polly Retry (5 Retries) 11](#_Toc123345564)

# Introduction

This document provides a detailed overview of the technologies, components, design\ architectural patterns and communication between processes that will incorporate the addition of processing APAR loans within the ABSolute loans workflow.

## Purpose

The Product Architecture Document (PAD) provides a comprehensive architectural overview of the newly proposed features\processes within the existing application(s). It presents an architectural view to depict data flow between the various components. It is intended to capture and convey the significant architectural decisions which have been made to design the system.

## Scope

The scope of this PAD is to convey the architecture of the ABSolute APAR POC.

# Prerequisites

* An understanding of the MVC or DDD architecture patterns
* An understanding of Database First approqach (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 crearte your database with seed data is also included in this clone (a folder within the Visual Studio solution).

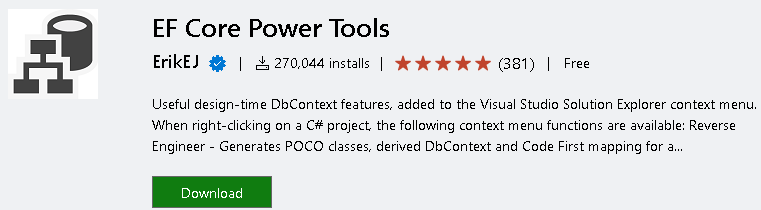
## SQL

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.

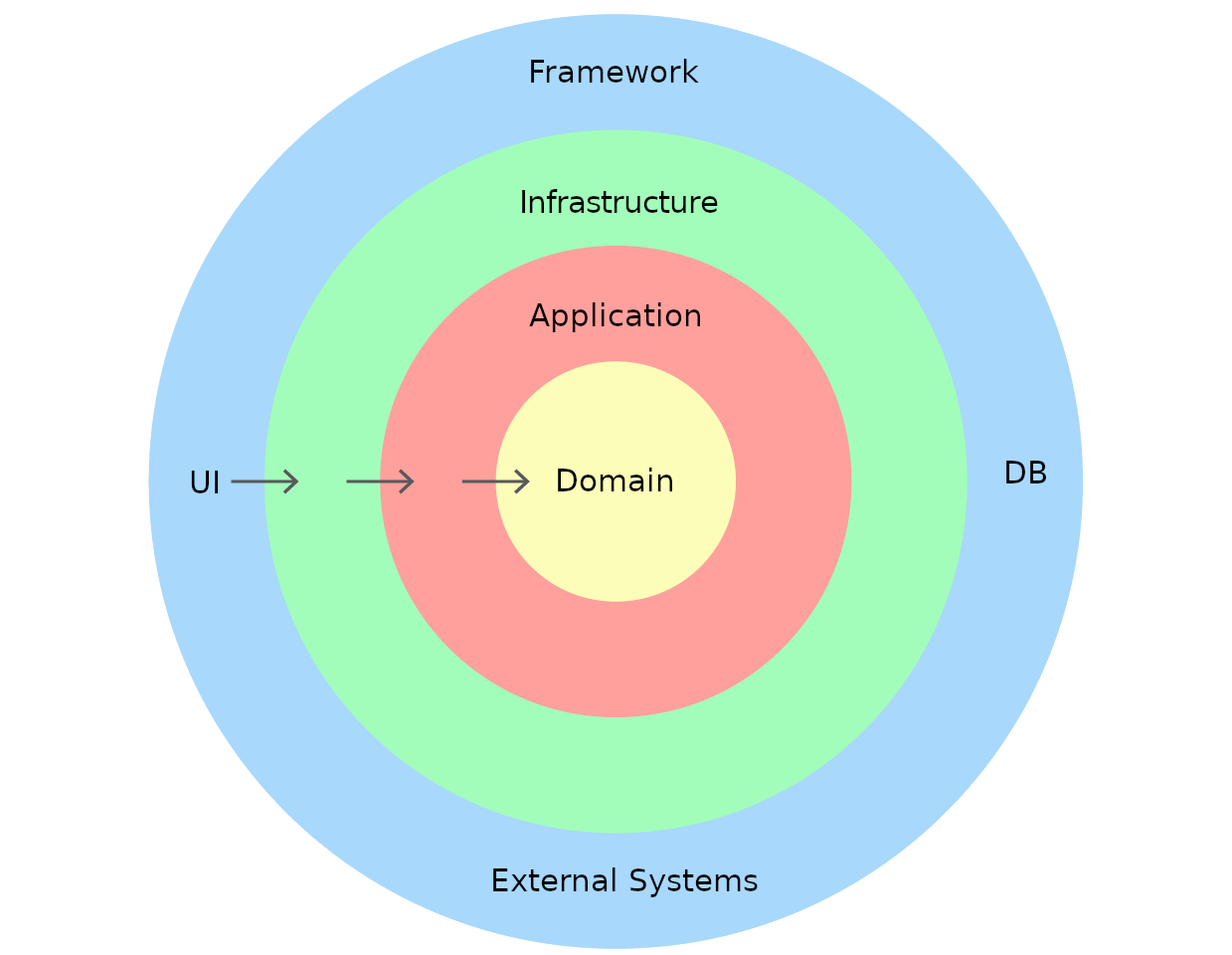
* MSTest
* Faker\Bogus
* MOQ
* IHttpFactory
* Http Polly (API retries)
* EF Core
* SQL Server
* EF Power Core Tools
* Middleware (Exception Handling)
* Logging (file based)
* GuardRails
* Blazor WASM
* Benchmark.Net
* AutoMapper
* Swagger UI

# Quick Explanation of Clean Architecture

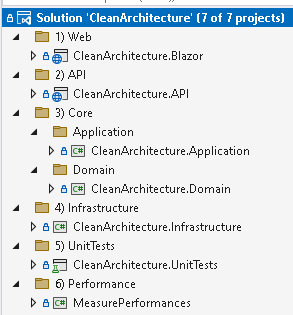
The concept of a Clean Architecture pattern has been around for over a decade and initially conceived by Robert Martin (better known as [*Uncle Bob*](https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html)). The keyword from Uncle Bob is *Interchangable.* In the image below, everything on the blue circle is interchanable, 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 satisifies 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\interchangable\decoupled.



# Visual Studio Solution Structure



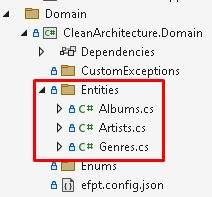
# Adding Database Entities and DBContext

# Clean Architecture - Main Layers

The Clean Architecture approach is very diversable, 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 dicingresults from multiple Infrastructe calls, which inturn 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 plugins 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!

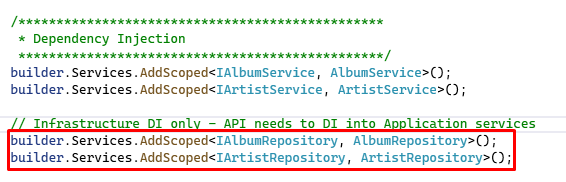
It is the outermost layer of the system and should have no knowledge of the inner layers.

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

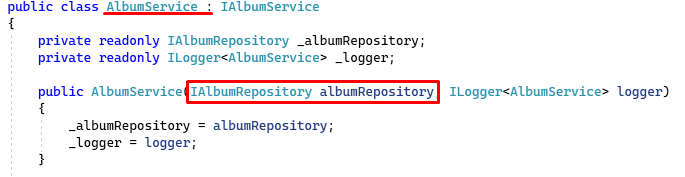
**NB:** The UI Application never depends on the infrastructure layer, but we have to reference the infrastructure layer into the UI project in the case to register the services dependency injection. So UI project should not use any code of the infrastructure layer other than dependency injection.

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).

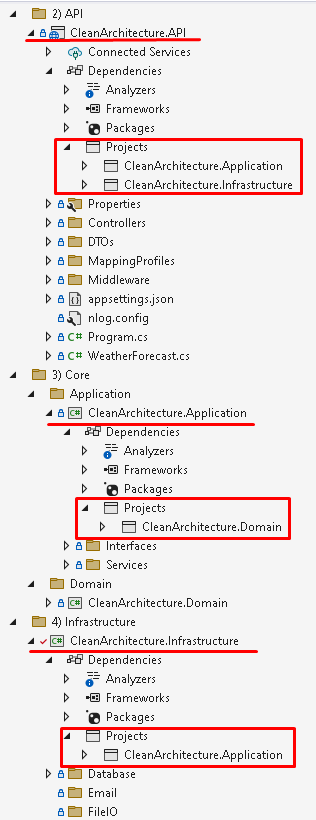
Web UI Programs.cs configuring Infrastructure DI:



Application service consuming the injected Infrastructure classes:



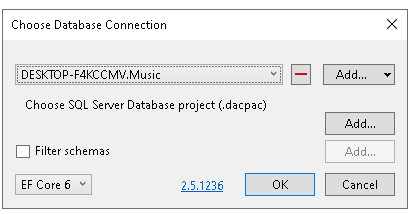
In the image below, you can see the inward reference structure between the projects. The API project must reference the Application layer so that it can makes calls to the various business logic services

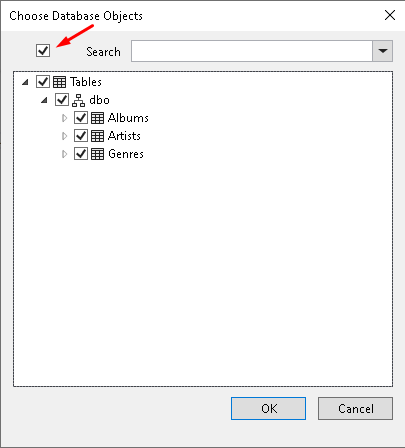


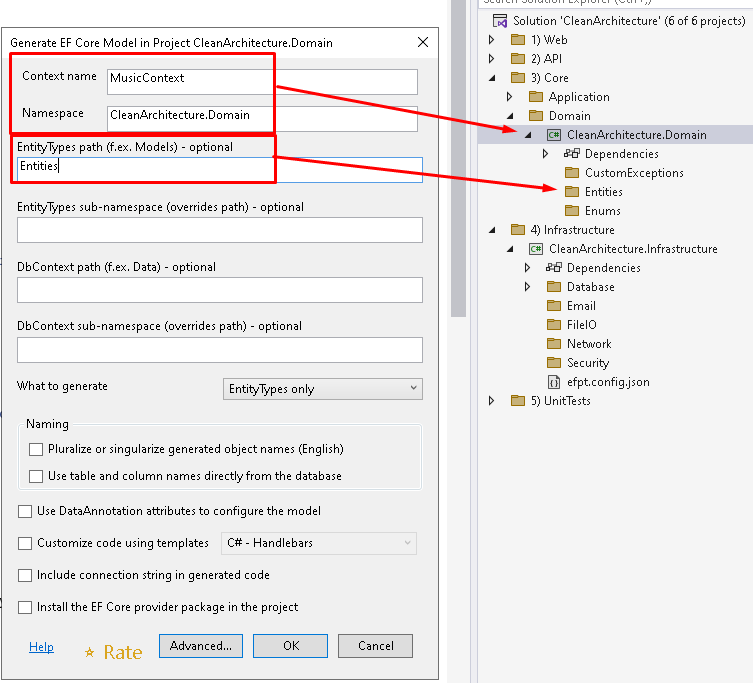
# Data Flow from UI to DB Service and Back

# Unit Tests

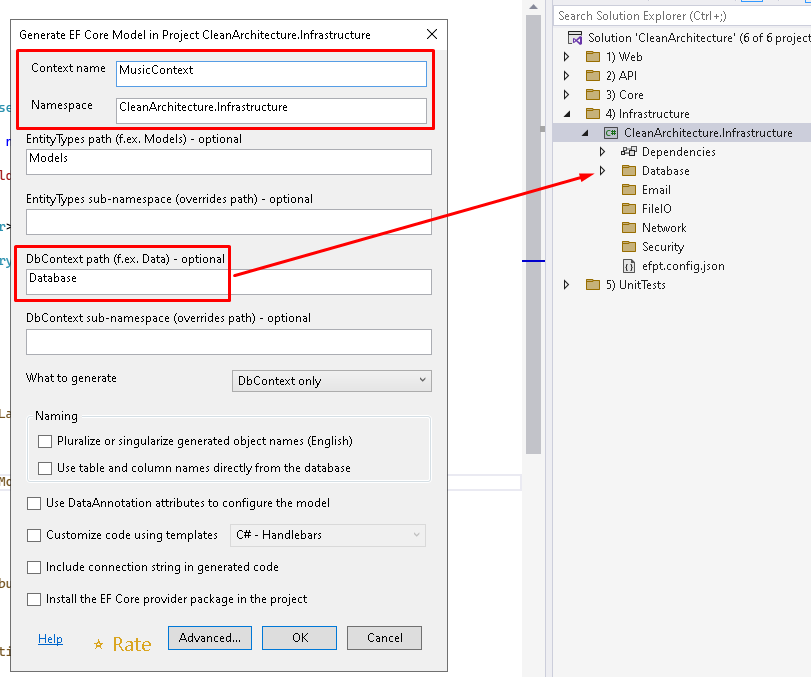
Models In Domain

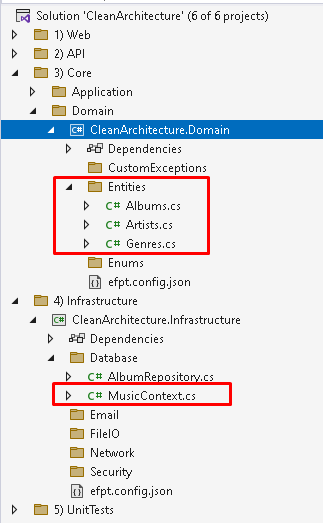






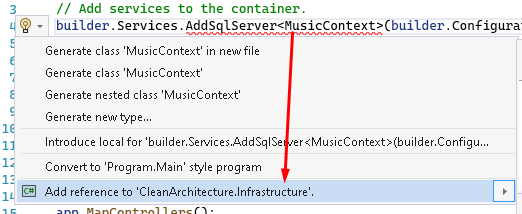
DBContext In Infrastructure

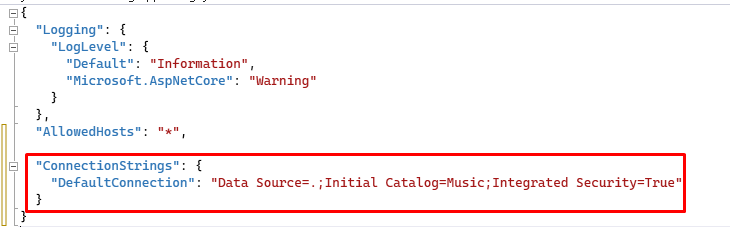




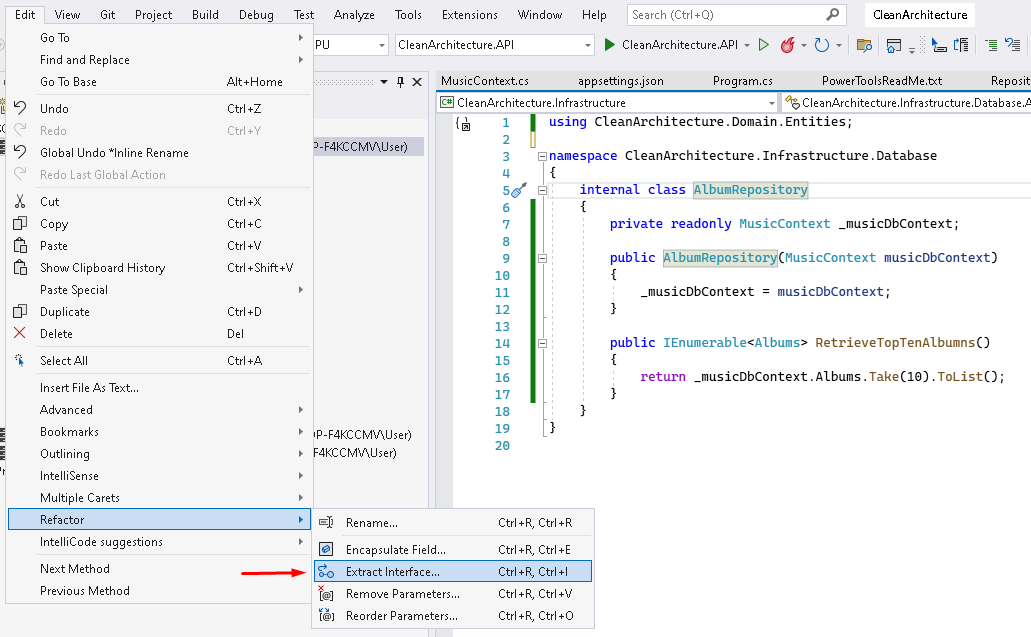
Connection String in API (Program.cs)

Add reference to DBContext (Infrastructure prj)

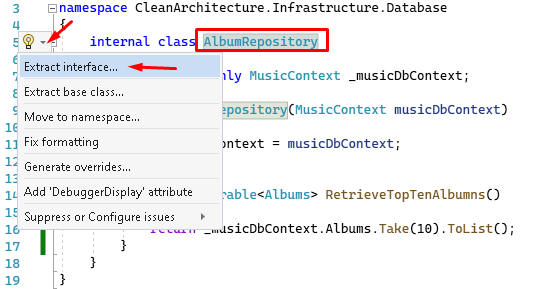


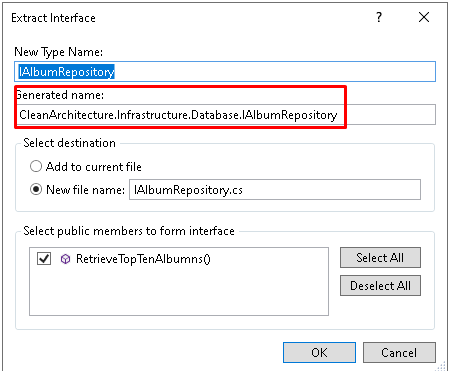


Create Interface



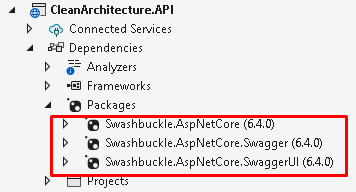
Create Interfaces and move to Application Prj





Swagger Enabled (Launch Settings API)



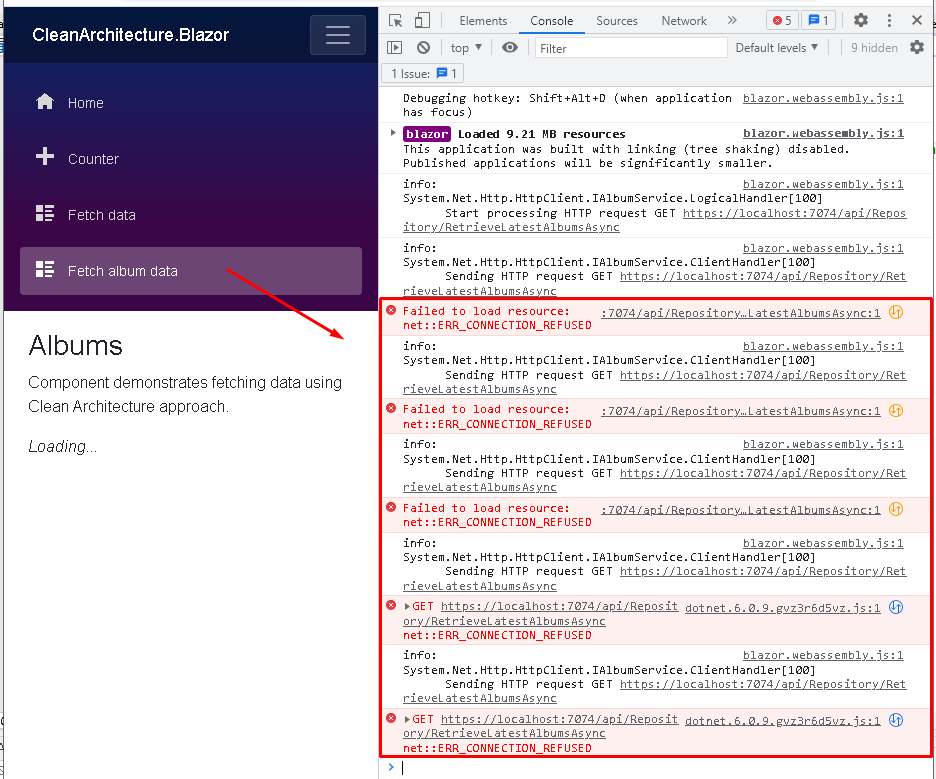


After Programs.cs



## Testing Polly Retry (5 Retries)

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, plus it jitters the retries, so as not to overwhelm the server with a call right away.



# Logging (Server Side)

