OPM Solution Template

Visual Studio 2022 and Dotnet CLI – Web API Solution Template

Date Created:	6 May 2023
Author:	RW Bradley
Document Status:	Review
Version:	1.0.0
Template Version Supported:	1.7.6
Last Updated:	10 May 2023

Table of Contents

INTRODUCTION

This template is an implementation of the Clean Architecture pattern using ASP.NET Core 6. Clean architecture provides a starting point for building OPM ASP.NET Core 6 solutions. It is loosely coupled and relies heavily on the Dependency Inversion principle.

Clean architectures produce systems that are:

- Framework's agnostic. The architecture does not depend on the existence of some library of feature-laden software. This allows you to use such frameworks as tools, rather than having to cram your system into their limited constraints.
- Unit Testable. The business rules can be tested without the UI, Database, Web Server, or any other external element.
- Independent of Client. The client can change easily, without changing the rest of the system. The client could be replaced by a Single Page Application or other client framework, without changing the business rules.
- Independent of Database. It is database agnostic. Your business rules are not bound to any database.
- Independent of any external intervention. Your business rules simply don't know anything at all about the outside world.

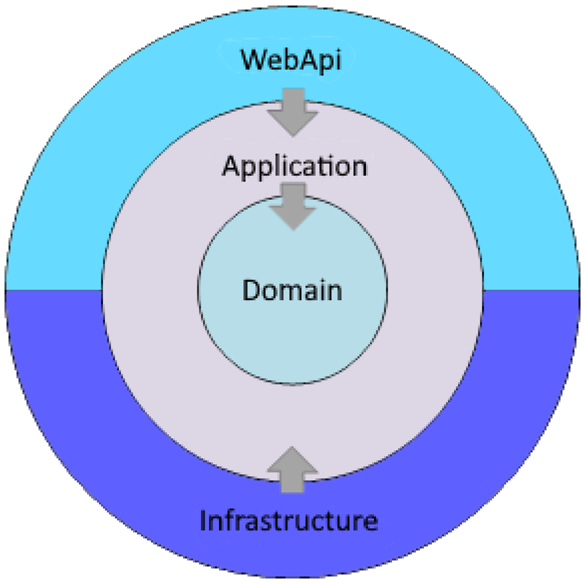
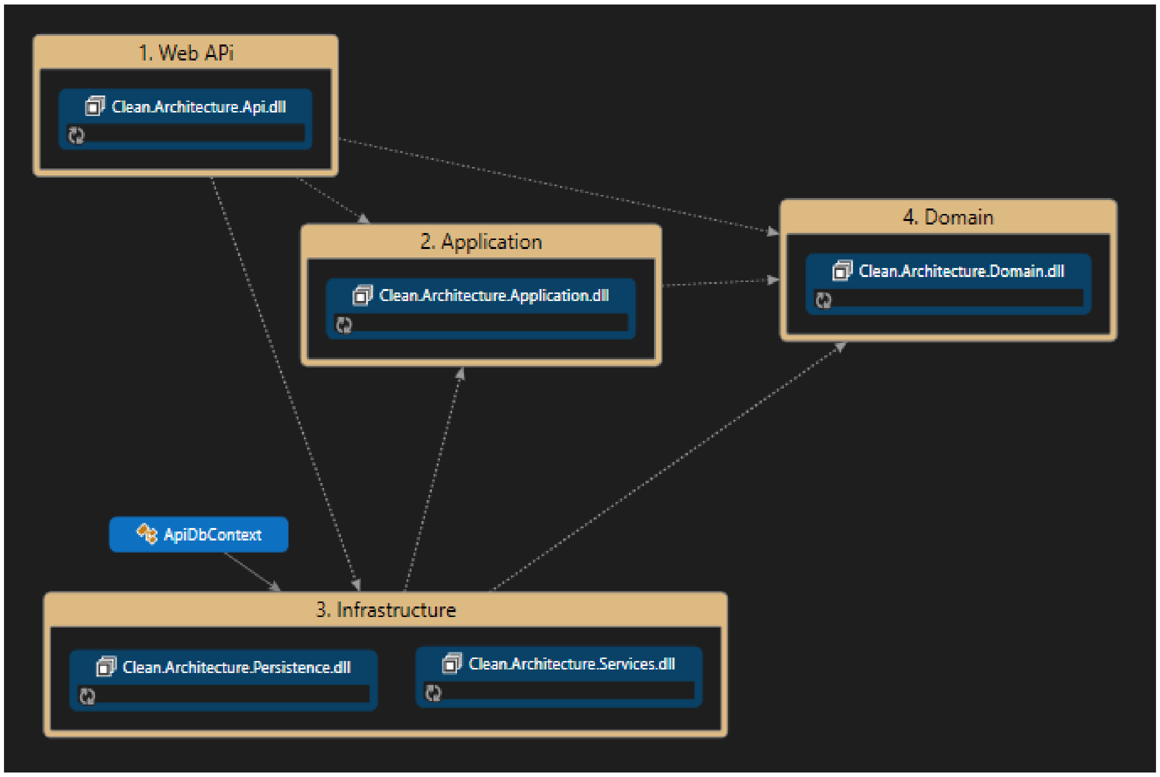


Figure 1- Clean Architecture

The figure above represents the overall architecture in terms of layers (projects) and their components. This demonstrates how the actual solution is layered and shows the interrelations between the layers, as opposed to showing the logical structure of a clean architecture.

The actual .NET 6 solution consists of several projects, separated into solution folders representing the layers for convenience.


1. Web APi
Clean.Architecture.Api.dll
2. Application
Clean.Architecture.Application.dll
4. Domain
Clean.Architecture.Domain.dll
ApiDbContext
3. Infrastructure
Clean.Architecture.Persistence.dll
Clean.Architecture.Services.dll

This figure is a code map generated from the solution template. It shows the different layers and projects, along with their interdependencies.

LAYERS

DOMAIN LAYER

The domain Layer implements the core, use-case-independent logic of the domain/system. This layer contains domain entities and has no dependencies on other layers or external libraries and frameworks.

The Domain project is the core and backbone project. It is the central project of the Clean Architecture design.

This package contains the high-level modules which describe the Domains Aggregate Roots, Entities, and Value Objects. This layer is a .NET 6 Class Project.

The domain layer contains these folders:

- Common Classes (base classes)
- Entities
- Enumerations
- Exceptions
- Interfaces
- Transfer Objects

COMMON

The common folder should contain the base classes. There is already a “*`BaseEntity`*” class located in the folder. It should be used to define all Entity's Id properties. The current base Id is configured as an integer.

ENTITIES

All entity classes are kept in this folder. An entity's class maps to a database table. The entity is a collection of properties that correspond to the database tables columns.

All Entity Classes must be included as “*`DbSet<TEntity>`*” type properties in the application “*`ApiDbContext`*” class.

ENUMERATIONS

All enumerations are kept in this folder. An enumeration defines a common type for a group of related values and enables you to work with values in a type-safe way within your code.

EXCEPTIONS

This folder contains exception classes. These classes are used by the custom exception middleware class in the API Layer. These classes help the solution to implement the “Fast-fail”[1] pattern. This pattern states that if a service must fail, it should fail fast and respond to calling entities as soon as possible.

INTERFACES

This folder should contain all the interfaces. Interfaces are important in keeping the application loosely coupled. All access to classes in the Applicating layer and the infrastructure layer should be made via interfaces that you would load using dependence injection.

Just a note. I create an interface before I create the implementation class.

TRANSFER OBJECTS

Transfer objects are also referred to as Data Transfer Objects (DTO). In the folder, there are two types of transfer objects, Request Objects, and Response Objects. Transfer Object should be prefixed with Request or Response based on the direction the transfer object is headed.

[1] https://en.wikipedia.org/wiki/Fail-fast

APPLICATION LAYER

The application Layer implements the business use cases of the application and is based on the business domain. A use case can be thought of as a specific user or functional operation. This layer contains the business use cases which orchestrate the high-level business rules. By design, the orchestration will depend on abstractions of external services (e.g., Repositories).

It references the domain layer but has no other references on any other layer or project. This layer is a .NET 6 Class Project.

The application layer contains these folders:

- Configuration
- Extensions
- Helpers
- Service Registration

CONFIGURATION

This folder comes with two classes but you're free to add other configuration classes. The classes are "*AppSettings*". This is a class that mirrors the application configuration items located in the "*appsettings.json*" file in the API layer. The second class, "*EmailSettings*" contains the required setting to send an email from the application. If not needed this class may be removed.

EXTENSIONS

Extension methods enable you to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type. Extension methods are static methods, but they're called as if they were instance methods on the extended type.

There are four extension classes in this folder. The first is "*ClaimsPrincipalExtensions*". This extension allows you to get the User Id, the Username, and the User's Email Address, from the JWT Access Token.

The second extension is "*EnumerationExtensions*". These extensions allow you to see if a value is part of an enumeration, exists in the enumeration, add another value/name to the enumeration, and remove a value for the enumeration.

The third extension is "*EntensionMethods*". This extension file contains malicious string extensions. List, is the string a JSON string, format a string with an SSN, or Phone Number, or Title Case a string.

The last extension class is "*GenerateEmployeeHash*". This will turn a string containing an employee SSN and Birthday into the eOPF EmployeeHash.

HELPERS

The classes in this folder are helpers. These are classes you would instantiate to provide helpers to your use cases. There is one helper class in this folder "*AutoMapping*". This is a class where you would build your AutoMapper configuration classes.

SERVICE REGISTRATIONS

The only class in this folder is the "*ServiceExtensions*" class. This is where you would add your application layer "*AddScoped*", and "*AddTransient*" services. They will automatically be loaded at the solution startup. Using this class keep the "*Program.cs*" Class in the API layer manageable.

Infrastructure Layer

This layer is responsible to implement the Contracts (Interfaces) defined within the domain layer to the Secondary Actors[2]. The infrastructure layer supports the Application layer by implementing abstractions and integrations to 3rd-party libraries and external systems.

The infrastructure layer contains most of your application's dependencies on external resources such as file systems, web services, third-party APIs, databases, and so on. The implementation of these actors should be based on interfaces defined within the domain layer.

The infrastructure layer contains two projects. The Persistence Project and an optional Services Project. If you have a very large project with many external dependencies, it may make sense to add additional service-related projects in the infrastructure layer.

Persistence Project

This project handles database concerns and other data access operations. This project references the Application and the Domain layers. This project contains the implementations of the data-related interfaces (e.g., Repositories) that are defined in the Domain project.

For instance, an SQL Server Database is a secondary actor that is affected by the application use cases, all the implementations required to consume the SQL Server are created in the infrastructure (persistence project) layer.

For example, if you wanted to implement the Repository pattern you would do so by adding an interface within the Domain layer and adding the implementation within the Persistence project (Infrastructure layer). This layer is a .NET 6 Class Project.

The persistence layer contains these folders:

- App Data Context
- Repositories
- Migrations
- Service Registration

App Date Context

The DbContext instance represents a session with the database and can be used to query and save instances of your entities. DbContext is a combination of the Unit Of Work and Repository patterns.

Repositories

The repository classes in this folder contain data query methods and data persistence methods, coordinated by a Unit of Work (DbContext) when persisting or updating data.

Migrations

Though this folder is not in the initial template, it will be created by Entity Framework Core when you run your first data migration.

Service Registrations

The only class in this folder is the "*`ServiceExtensions`*" class. This is where you would add your application layer "*`AddScoped`*", and "*`AddTransient`*" services. They will automatically be loaded at the solution startup. Using this class keep the "*`Program.cs`*" Class in the API layer manageable.

[2]A secondary actor is a participant who assists the system in completing a use case, rather than initiating it. The system calls upon the secondary actor to obtain information or achieve a specific outcome during the execution of its use cases.

Services Project(s)

This project would provide all the necessary functionality needed to communicate and/or access external assets. These assets might be external databases, email services, external application endpoints, etc. This project contains the implementations of the service-related interfaces (e.g., Email Service) that are defined in the Domain project. This layer is a .NET 6 Class Project.

The service layer contains these folders:

- Service Registration

Service Registration

The only class in this folder is the "*`ServiceExtensions`*" class. This is where you would add your application layer "*`AddScoped`*", and "*`AddTransient`*" services. They will automatically be loaded at the solution startup. Using this class keep the "*`Program.cs`*" Class in the API layer manageable.

WEB API LAYER

The web API layer contains the endpoint (entry-points) elements that provide Client access to the application. The Web API layer has references on both the Application and Infrastructure layers, however, the reference on Infrastructure is only to support dependency injection. This layer is an ASP.NET 6 Web API.

The Web API layer only has two responsibilities; Receive requests from the client and respond, back to the client, to the outcome of these requests. There should be no business logic or data access logic in the controllers. Accessing the Business Use Cases in the Application Layer should be done through Interfaces.

The Web API layer contains these folders:

- Controllers
- Extensions
- Middleware

CONTROLLERS

This folder contains the application controllers. The Controllers should logically group similar actions.

EXTENSIONS

This folder contains extension classes for the Web API project. There are currently seven files in this folder. These service configuration classes will be loaded when the application, will automatically be loaded at the solution startup. Using this class keep the "*`Program.cs`*" Class manageable

MIDDLEWARE

This folder contains custom middleware classes. It contains one class, a global error handler that catches all errors and removes the need for duplicated error-handling code throughout the .NET API. It's configured as middleware in the configure HTTP request pipeline section of the "*`Program.cs`*" file.