The Code Wrapper.

Wrapping practical developer solutions in a beautiful box

Implementing a Clean Architecture in ASP.NET Core 8

In Architecture, dev

Tags .net, asp.net, clean architecture, cqrs, ddd, dependency inversion, domain-driven design, ef core

September 26, 2021 thecodewrapper

Introduction

about building your own clean architecture.

This part is merely an overview of the overall architecture. More details on the internals and implementation will follow in separate posts.

Also, this post will not describe the concepts of a clean architecture in detail, nor will be an introduction to Domain-Driven Design or best practices. If you need to learn more about clean architecture concepts in detail, you can follow one of the links on the bottom of this post.

The intention of this design is to act as a starting point for building ASP.NET Core solutions. It is loosely-coupled, relies heavily on Dependency Inversion principle, and promotes Domain-Driven Design for organizing your core (although this is not forced). The goal for which I set out to do this, was to build a bare-bone, maintainable and extendable architecture.

Abstraction is key

My secondary goal was to abstract away concepts like domain events, CQRS and event sourcing from dependencies on specific implementations – or even other abstractions brought by some well-known packages – such as MediatR, MassTransit, EventStore etc.

Another particular thing I wanted to do with this, is to be able to combine and easily switch between relational (or non-relational for that matter) persistence methods and a streaming or event sourcing method without affecting any of the underlying architecture. I've seen quite a few implementations where the decision to use event sourcing for persistence is deeply rooted (or at least largely apparent) in how the core is organized. I wanted to avoid that and abstract it.

Although this is not a fully-fledged event store implementation (the default implementation is using Entity Framework Core with SQL Server to persist events, instead of something like an EventStore, which would be more technically appropriate), it serves the purpose of the aforementioned abstraction. The specifics of the actual implementation of event sourcing using Entity Framework Core, and the inclusion of snapshots and retroactive events, will be detailed in a future post.

I do not claim that this design is in anyway better or cleaner than any others out there. This is simply what worked very well for me in several of my projects. Most software developers out there recognize that there is no single architecture which is best. The best architecture is the one that works well for you, your team and your type of project. This is by no means a production-ready solution, it is not a framework, it is merely a starting point.

Furthermore, this solution is not by all means complete in terms of features. There are several features which can be beneficial depending on the type of project you are building (caching, analytics etc.) which are not included at the time of this writing, but can be easily added if needed. I try to maintain this and add new features as much as time permits, so if you have any suggestions feel free to ask or contribute.

History

When I set out to build this bare-bone design I of course took input and advice from some very well-designed open-source architectures on GitHub:

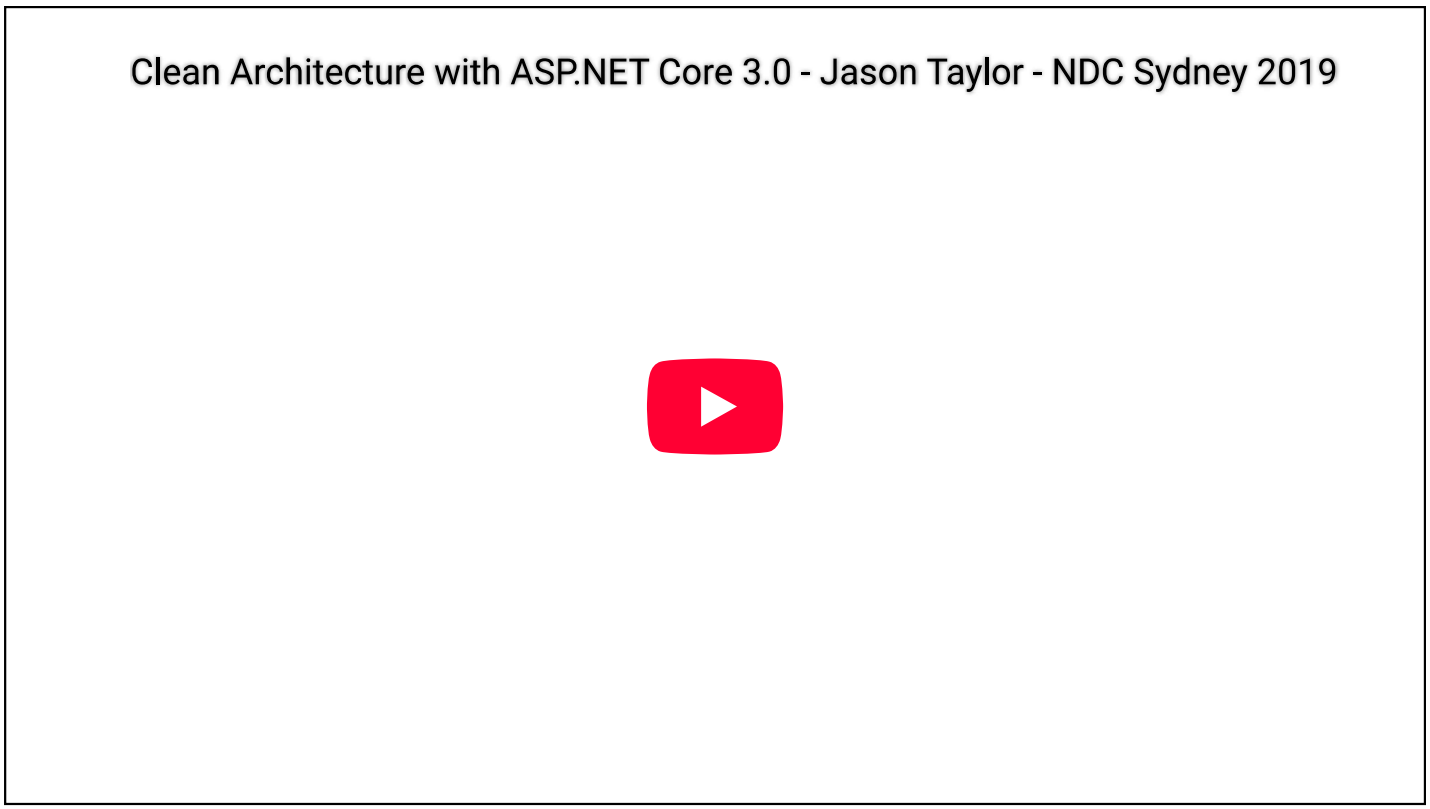
https://github.com/ardalis/CleanArchitecture

https://github.com/dotnet-architecture/eShopOnContainers

https://github.com/jasontaylordev/CleanArchitecture

https://github.com/blazorhero/CleanArchitecture

There is great article from Uncle Bob (Robert C. Martin), who is a huge advocate of clean code and clean architecture practices: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html. His book Clean Architecture is also a great resource of valuable information about universal rules of software architecture, so I definitely recommend giving it a read if you are interested in building maintainable, loosely-coupled, long-lasting solutions.



Technologies used

- ASP.NET Core 8
- Entity Framework Core 8
- MassTransit
- AutoMapper
- Razor Components
- ASP.NET Core MVC
- GuardClauses
- xUnit
- Moq
- Fluent Assertions
- FakeItEasy
- Docker

The features of this particular solution are summarized briefly below, in no particular order:

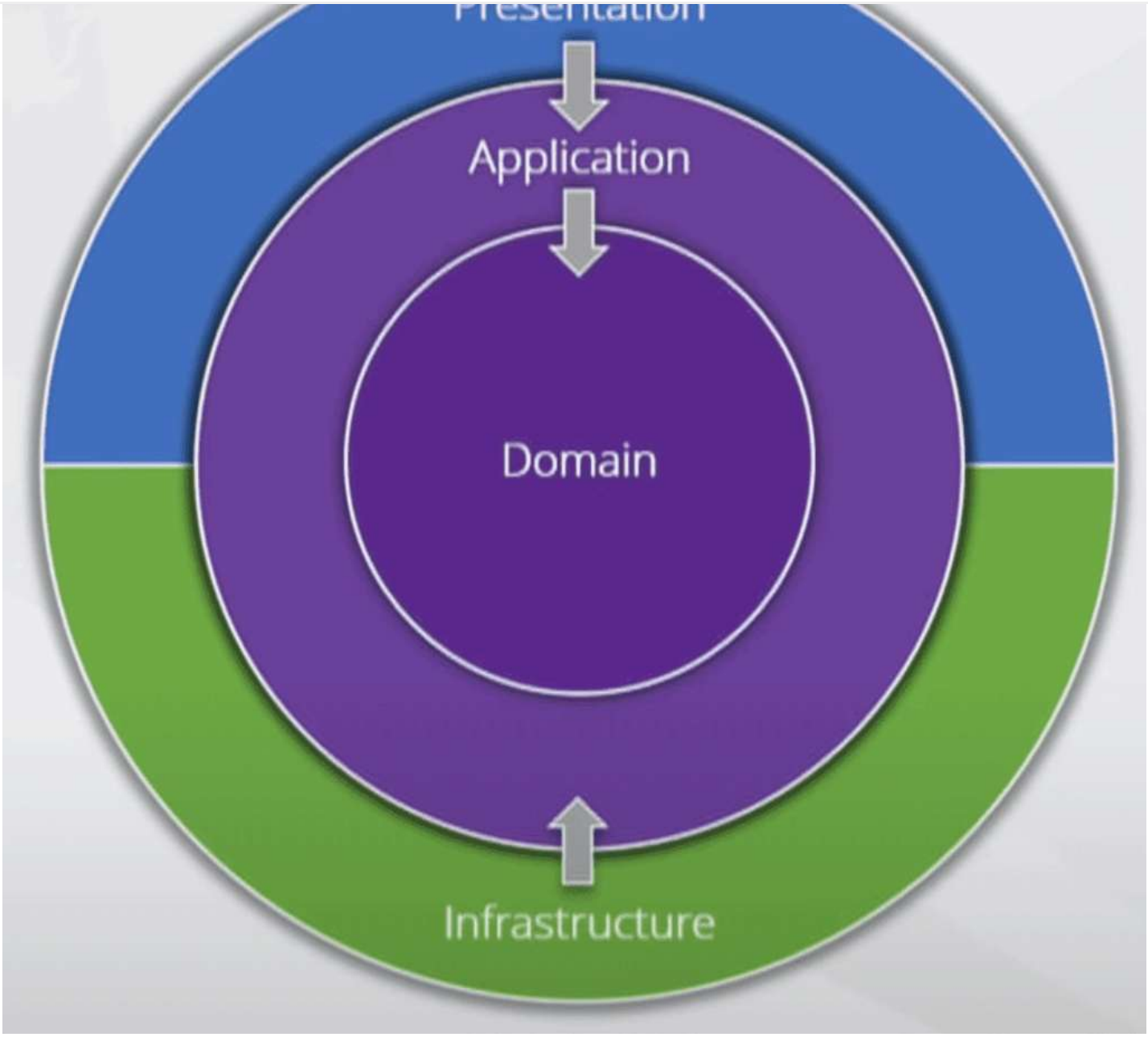
- Localization for multiple language support
- Event sourcing using Entity Framework Core and SQL Server as persistent storage, including snapshots and retroactive events
- EventStore repository and DataEntity generic repository. Persistence can be swapped between them, fine-grained to individual entities
- Persistent application configurations with optional encryption
- Data operation auditing built-in (for entities which are not using the EventStore)
- Local user management with ASP.NET Core Identity
- Clean separation of data entities and domain objects and mapping between them for persistence/retrieval using AutoMapper
- ASP.NET Core MVC with Razor Components used for presentation
- CQRS using handler abstractions to support MassTransit or MediatR with very little change
- Service bus abstractions to support message-broker solutions like MassTransit or MediatR (default implementation uses MassTransit's mediator)
- Unforcefully promoting Domain-Driven Design with aggregates, entities and domain event abstractions.
- Lightweight authorization framework using ASP.NET Core AuthorizationHandler
- Docker containerization support for SQL Server and Web app

Some other goodies:

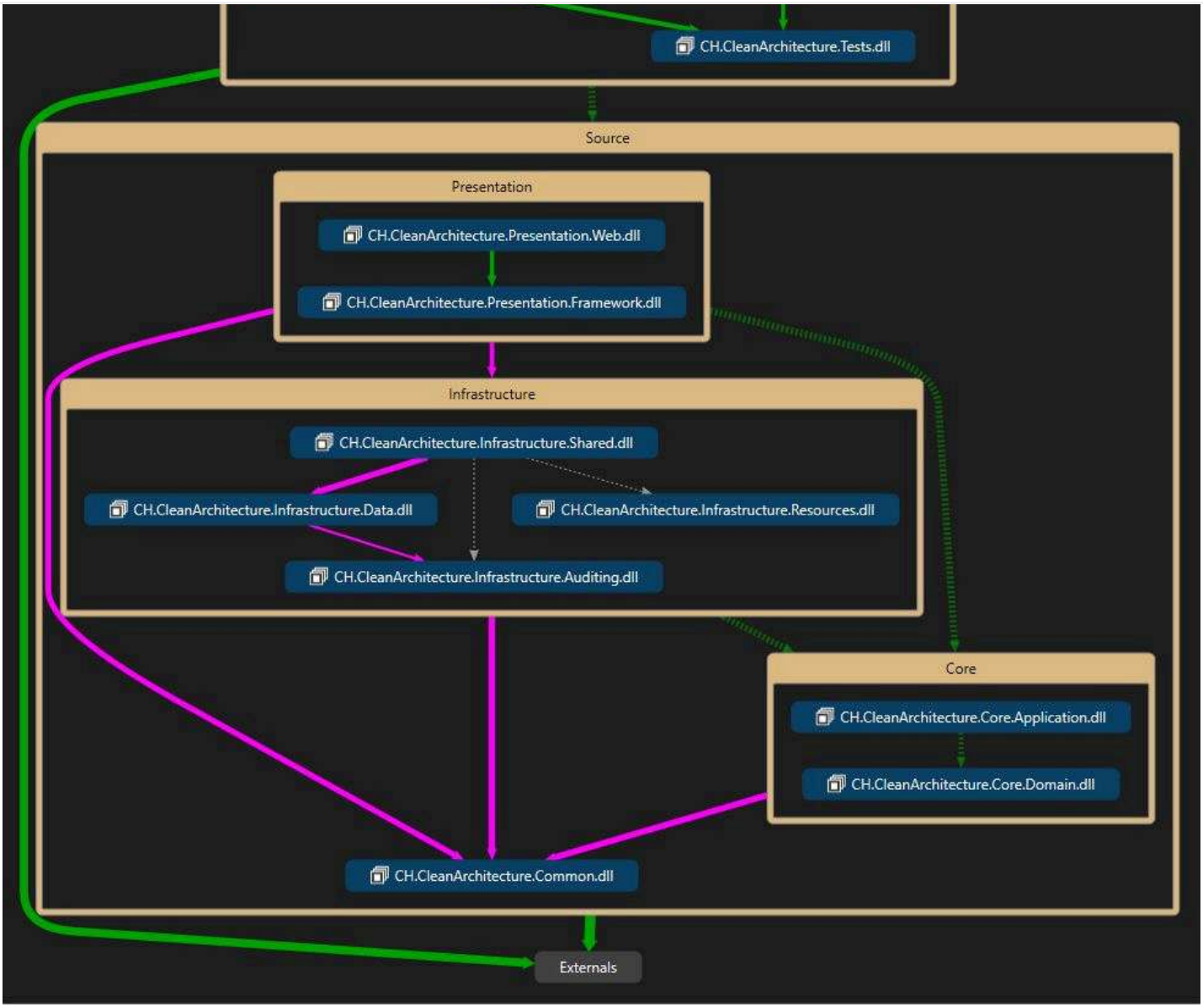
- Password generator implementation based on ASP.NET Core Identity password configuration
- Razor Class Library containing ready-made Blazor components for commonly used features such as CRUD buttons, toast functionality, modal components, Blazor Select2, DataTables integration and page loader
- Common library with various type extensions, result wrapper objects, paged result data structures, date format converter and more

Further below is a figure depicting the overall architecture in terms of layers and their components. The figure is meant to be a representation of how the actual solution is layered and show the gradual interrelationships between the layers, as opposed to simply showing the logical structure of a clean architecture.

In any case, the below diagram taken from Jason Taylor's talk at NDC Sydney (2019), broadly depicts the logical structure of the architecture:



The actual solution consists of several projects, separated in folders representing the layers for convenience. The below figure is a code map generated from the solution. It shows the different layers and projects, along with their inter-dependencies.



Code map generated from Visual Studio 2019 for the Clean Architecture solution.

As you can see from the arrows, all dependencies point downwards (or inwards if this was a circle diagram). there are no arrows pointing upwards between the Core, Infrastructure and Presentation layers.

The Core Layer

The Core Layer is made up of two parts, the inner core and outer core. The inner core is the domain and the outer core is the application. This consists of two projects in the solution

Inner Core (Domain Layer)

This layer contains application-independent business logic. This is the part of the business logic which would be the same even if we weren't building a software. It is a formulation of the core business rules.

The organization of this project follows Domain-Driven design patterns, although this is a matter of preference and can be handled any way you see fit. This includes:

- Aggregates, entities, value objects, custom domain exceptions, and interfaces for domain services.
- Interfaces for domain-driven design concepts (i.e. *IAggregateRoot*, *IDomainEvent*, *IEntity*).
- Base implementations of aggregate root and domain event. Also contains specific domain events pertaining to the business processes.

Outer Core (Application Layer)

This layer contains application-specific business logic. This contains the "what" the system should do. This includes:

- Interfaces for infrastructure components such as repositories, unit-of-work and event sourcing.
- Command models and handlers
- Query models (handler implementations are contained in the Infrastructure layer)
- Interfaces and DTOs for cross-cutting concerns (i.e. service bus)
- Authorization operations, requirements and handlers implementations
- Interfaces and concrete implementations of application-specific business logic services.
- Mapping profiles between domain entities and CQRS models

This layer contains details, concrete implementations for repositories, unit-of-work, event store, service bus implementations etc. This contains the “how” the system should do what is supposed to do. The decoupling between the application layer and the infrastructure layer is what allows solution structures like this one to change and/or add specific implementations as the project requirements change.

In overview, this layer contains:

- Generic and specific repositories implementations
- EF DbContexts, data models and migrations
- Event sourcing persistence and services implementations
- Implementations for cross-cutting concerns (i.e, application configuration service, localization service etc.)
- Data entity auditing implementation

This consists of 4 projects in the solution under the Infrastructure folder, the Auditing, Data and Shared projects.

The *Infrastructure* project contains domain and generic (CRUD and event-sourcing) repository implementations, DbContexts, EF Core migrations, entity type configurations (if any), event store implementation (including snapshots), data entity to domain object mappings, query handlers, and persistence related services (i.e. a database initializer service).

The *Auditing* project consists of various extensions methods for DbContext, primarily related to SaveChanges. It is responsible for generating auditing records for each tracked entity’s changes.

The *Resources* project contains localized shared resources and resource keys, along with localization services implementations.

The *Shared* project contains service implementations for cross-cutting concerns such as user management and authentication, file storage, service bus, localization, application

The Presentation Layer

This layer essentially contains the I/O components of the system, the GUI, REST APIs, mobile applications or console shells, and anything directly related to them. It is the starting point of our application.

For this starting point solution, it contains the following:

- ASP.NET Core MVC web application using Razor Components
- A shared class library containing common Razor Components, such as toast notifications, modal components, Blazor Select2, DataTablesJS integration and CRUD buttons

The Common Layer

This single class library contains common types used across all other layers. What you would typically include in this library is things that you would wrap up into a Nuget package and use in multiple solutions. For clarification, this does not represent the Shared Kernel strategic pattern in DDD. Some of these include:

- A generic Result wrapper
- Paged result models for query operations
- CLR type extensions
- Notification models
- Custom attributes and converters

Yes, yes, give me the code

The repository for the source code of this solution can be found here. The solution includes an extremely basic domain model for an online shop application, as an example. The UI part for

Update – 2024/07/29: The Blazor Server version of the solution is on a separate repository here**.** Maintenance for the architecture will happen in this repository. The original repository will NO longer be maintained.

Update – 2022/12/05: Some abstractions regarding domain-driven design tactical patterns, event sourcing and data persistence have been extracted into separate libraries. I've also went ahead and nugetized those libraries, so that anyone can easily fit them into their projects. The full set of libraries in Nuget gallery can be found here (with the 'CH' prefix). The source code for the packages can be found here.

Final thoughts

Just like any software solution, this is not a one-size-fits-all architecture. It is simply what worked very well for me in my projects. As mentioned in the introduction of this post, there is still quite a lot to include in this solution, therefore I will be updating the repo on GitHub, as well as this post, whenever something new is added.

Please feel free to leave any feedback or suggestions on this, and if you would like to have anything in particular be included in the project.

Future work

In the near future, I've planned to make the following additions/changes to the solution:

- Add support for caching, with an implementation for Redis
- Replace MVC for presentation with Blazor Server
- Include a RESTful API for domain business logic
- Additional implementation of event-sourcing using EventStoreDB

Here are some additional posts you may find helpful in understanding what clean architecture is all about:

https://www.freecodecamp.org/news/a-quick-introduction-to-clean-architecture-990c014448d2/

https://www.oncehub.com/blog/explaining-clean-architecture

thecodewrapper

Next Post:

EF Core: Effectively decouple the data and domain model

Related Posts:

A consolidated guide to Well-Architected Frameworks

Designing the domain model to support multiple persistence methods

Simple Event-Sourcing with EF Core and SQL Server