Brandon Atkinson

Hexagonal & Screaming Architecture in .NET

Applying Consistent Patterns to Our Code to Improve Speed and Efficiency.

Agenda

- Introduction
- What is Screaming Architecture?
- What is Hexagonal Architecture?
- Surprise Onion Architecture!
- Comparing Project Structures
- Let’s Build Something! DEMO
- How This Helps With Planning
- Q&A

Introduction

Brandon Atkinson - Senior Manager, MCG Health

I've been writing code for 20+ years, maintain a blog, written books, have a patent, and just love tech.

My love for Hexagonal and Screaming architecture was born at Capital One and honed at MCG Health. My teams solved a lot of challenging problems, using many different languages. What we discovered was Hex + Screaming worked amazingly well across them all.

I started at Capital One on C# .NET Core, building APIs, breaking down a monolith into SOA. We used traditional N-Tier architecture, and it worked fine....but we made **a lot** of mistakes (not related to architecture!).

I had the opportunity to lead a team building a generic CI/CD pipeline for all teams to use. This was my introduction to Screaming architecture. We architecture the code around each stage (use case) of the pipeline, which gave us a really clean project structure that screamed what each stage did.

I was looking for a change so I moved over to lead a team provisioning AWS accounts for teams. This is when we started applying Hexagonal with Screaming architecture. Hexagonal works really well in Golang as its implementation of interfaces is "everything is an interface". Golang is really well suited for Onion Architecture (we'll talk about this later) due to its lack of native DI implementations.

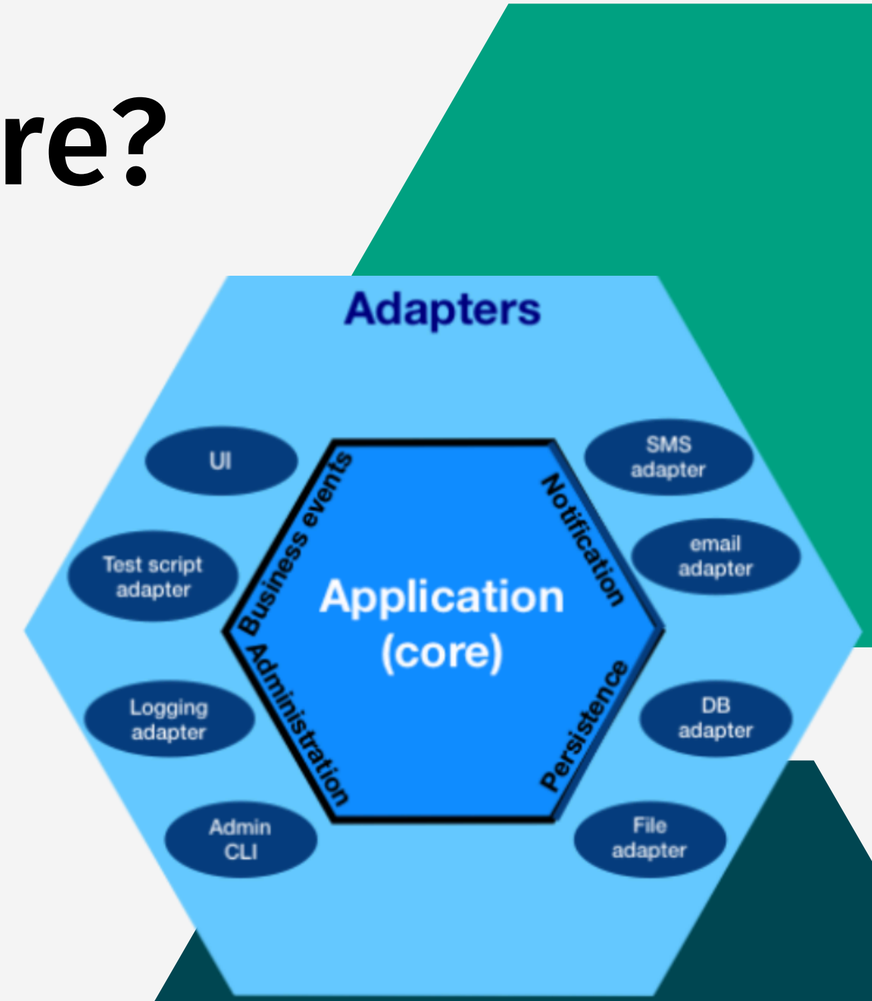
At MCG I came around full circle and back to the teams breaking down monoliths, and this is where we fully embraced Hex + Screaming + Onion and created the "guardrails" for all engineering teams to use when building their APIs.

What is Hexagonal Architecture?

Hexagonal architecture is an architectural pattern, introduced by Alistair Cockburn, designed to create loosely coupled application components that can seamlessly connect to their software environment using “ports” and “adapters”, sometimes called “driving” and “driven”:

- **Ports/Driving**: Incoming requests (UIs, APIs, CLIs, etc)
- **Adapters/Driven**: Calls out to support business logic (downstream APIs, databases, etc)

This approach makes components interchangeable at any level and facilitates test automation, offering a robust alternative to the traditional layered architecture. In hexagonal architecture, each component communicates with others through well-defined "ports", following a specified interface to ensure loose coupling and flexibility.

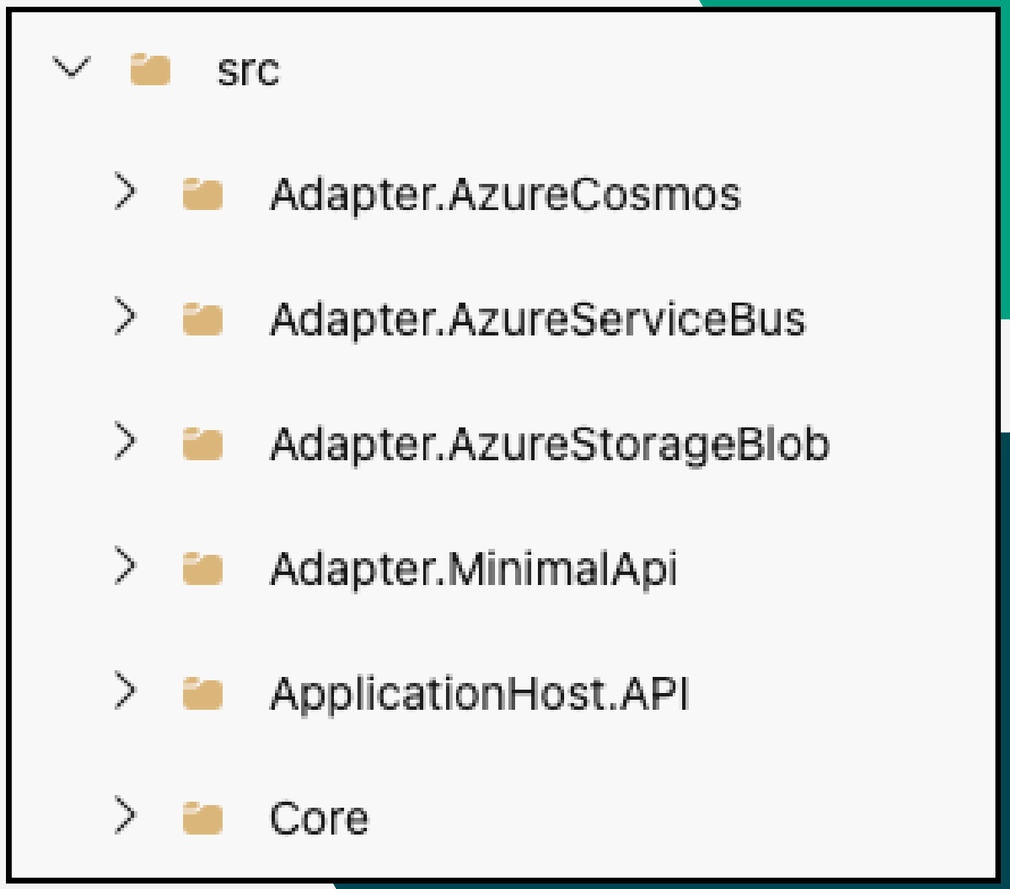


Hexagonal Architecture in Practice

In practice, Hexagonal Architecture will look similar to this structure. We have our “Core” project that houses all of our business logic and interfaces.

We then have as many “Adapter” projects as needed to support our Core business use cases. Adapters are clearly and explicitly named to support Screaming Architecture.

It’s all tied together via the “ApplicationHost” project, which is the entry point for our application and is also responsible for dependency injection.

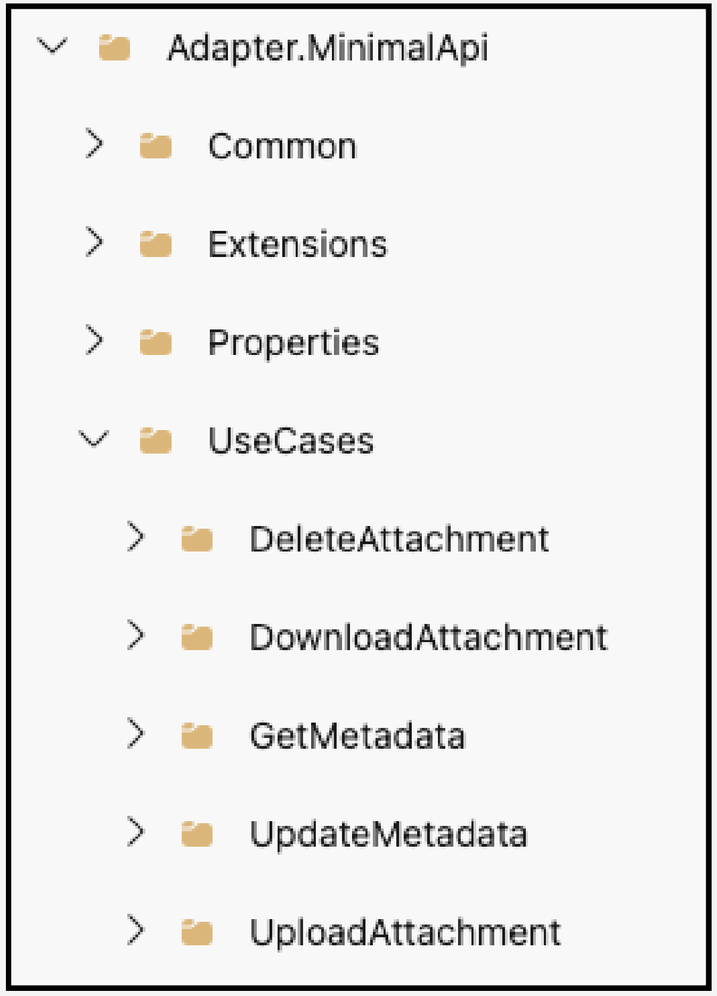


What is Screaming Architecture?

Screaming architecture, a concept introduced by Robert Martin (Uncle Bob), emphasizes that a software system’s structure should clearly communicate its purpose. Much like a building blueprint reveals the function of a structure, a well-designed software architecture should make its intent immediately apparent.

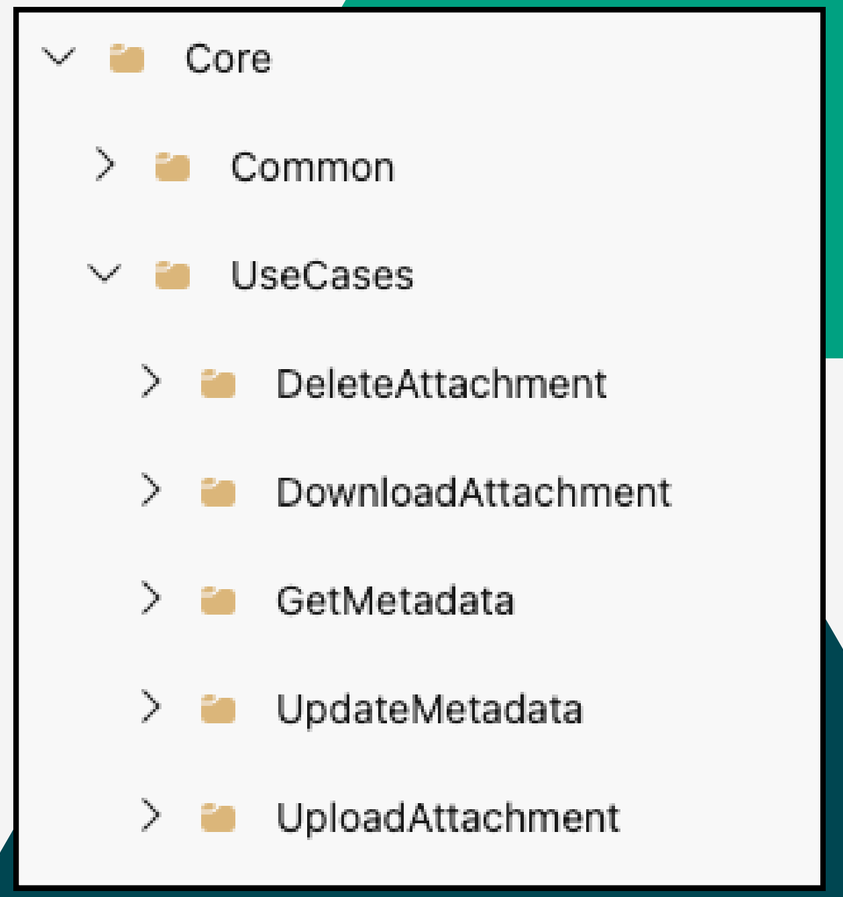
An application properly implementing Screaming Architecture allows for easily finding the relevant code you care about it, as the code itself screams what it does. The goal is to cut down on abstraction through directories and naming conventions, making it extremely clear what code does.

Screaming Architecture in Practice



In practice, Screaming Architecture will look similar to this structure. We have our API Adapter project that houses all of our endpoints to support our business use cases. A directory for each use case.

You'll see the exact same structure in the Core project, with corresponding use case directories.



Naming in Hexagonal and how to make it Scream!

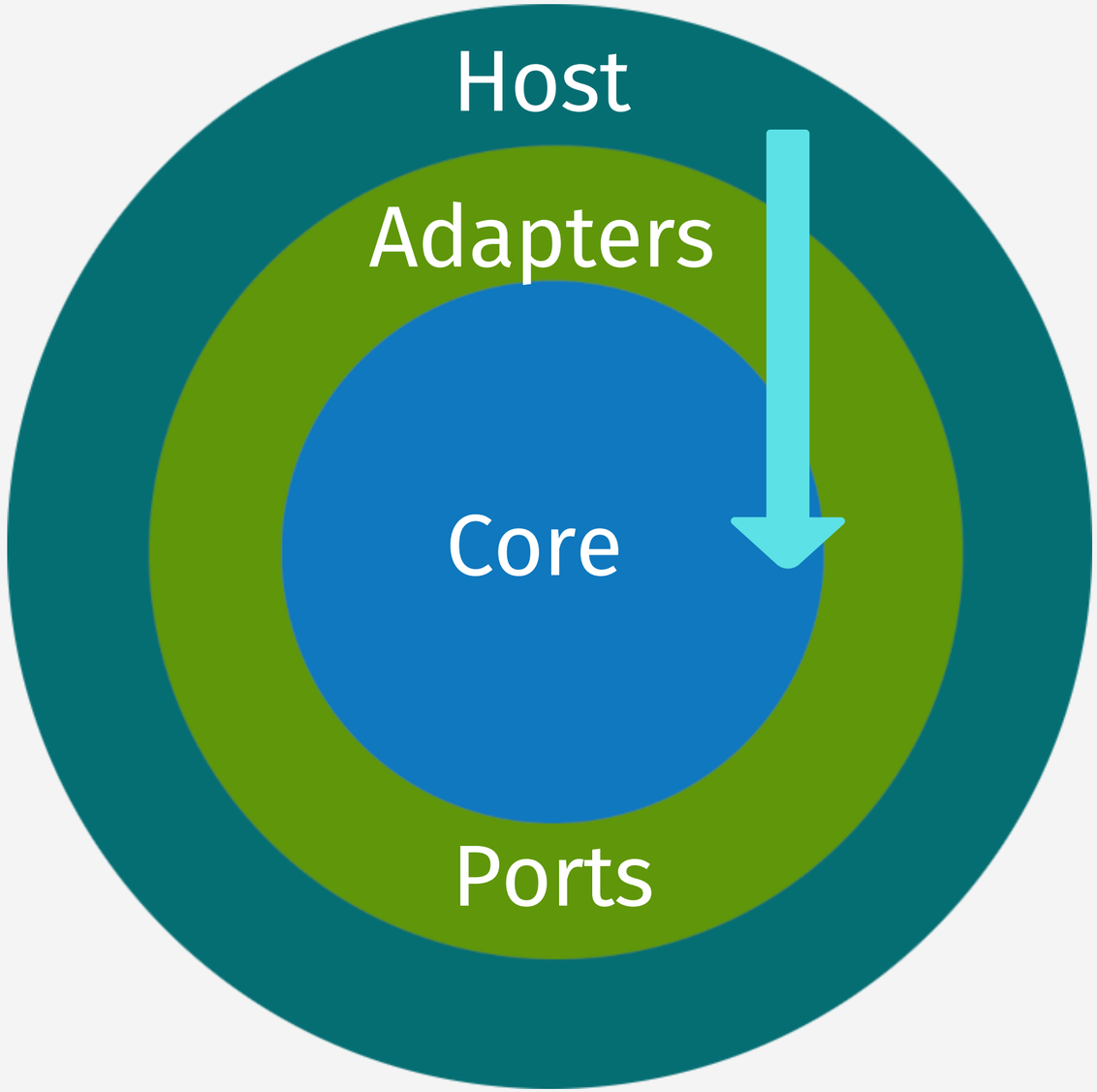
Hexagonal may go by different names: **Ports & Adapters**, **Driving & Driven**, but I prefer ***Hexagonal*** and will only use this name. However, the other names are here for a reason as they help support Screaming in our project.

- **Port (Driving) Projects**: These are projects that "drive" traffic into the application. These are often APIs, GraphQL, CLIs, etc. A Port project will always be incoming calls that your Core business logic project will respond to.
- **Adapter (Driven) Projects**: These are projects that are "driven" by your Core project. These are projects that interact with databases, other downstream APIs, etc. These are called by Core to support some functionality in the business logic.

We follow these naming conventions to help support Screaming architecture. If you look a project and its named Port.XXX, you know its an entry point into the application, and vice versa, Adapter.XXX screams it supports Core functionality.

NOTE: *The screenshots in this presentation show "Adapter.MinimalAPI" where the GitHub code shows the proper name of "Port.MinimalAPI"*

Onion Architecture



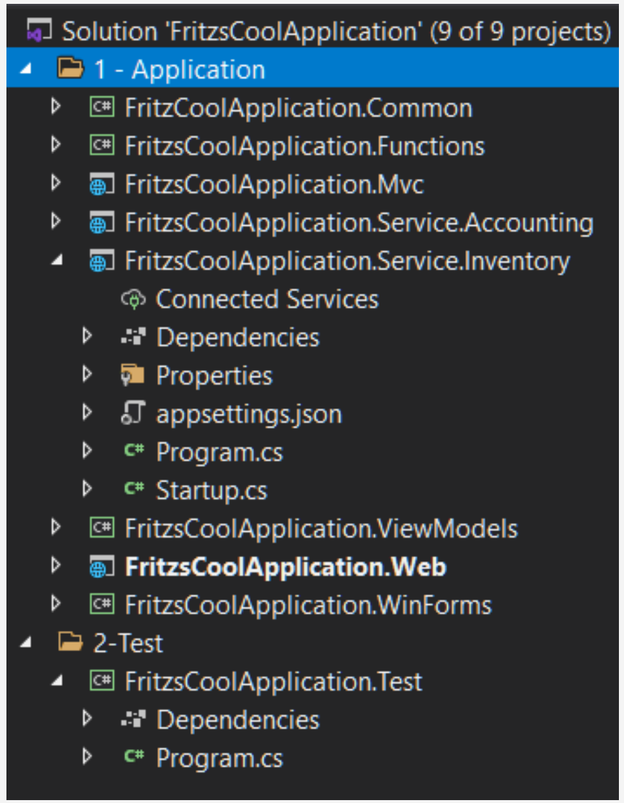
The bonus architecture pattern!!

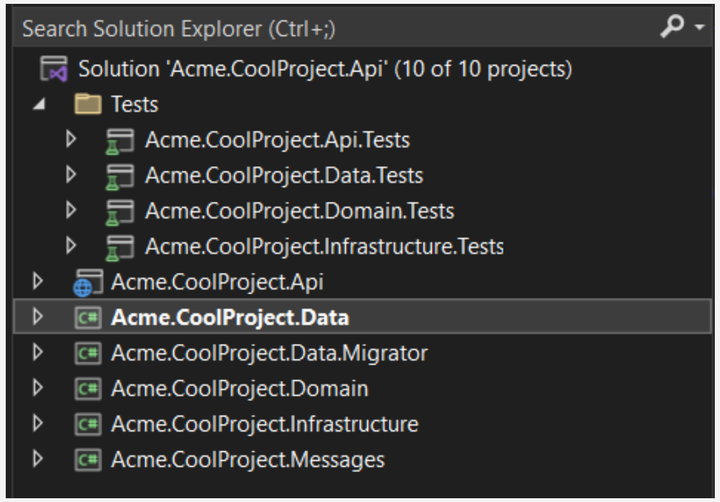
Onion Architecture represents a flow of dependencies in your application. Basically, dependencies should be inward and never outward. Outer layers may know about the inner layers, but inner layers should NEVER know about the upper layers.

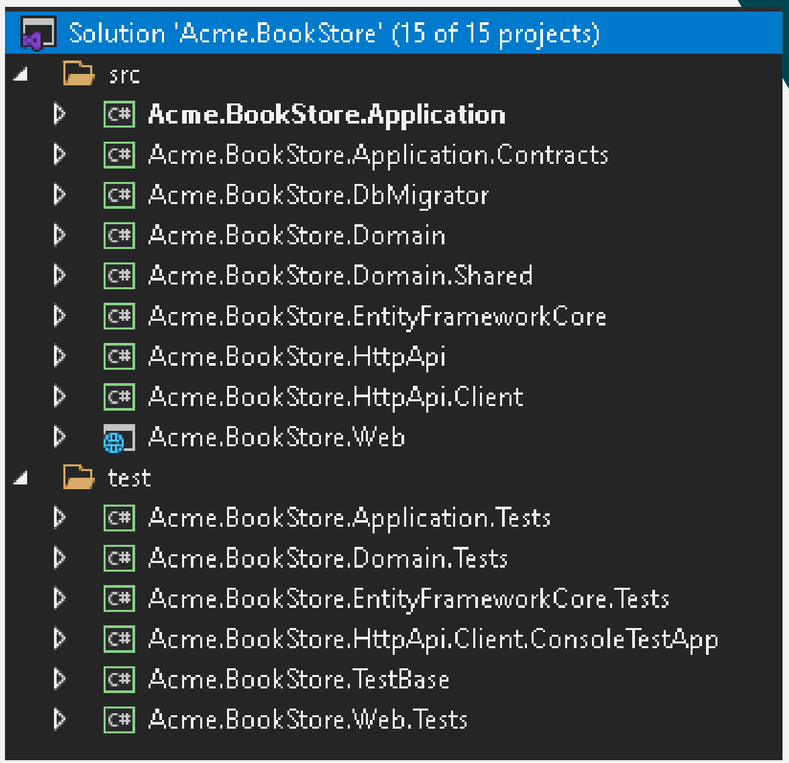
In this scenario, Core will never understand Adapters. Core defines the Interfaces that the Adapters implement, but it gets its dependencies from the Host, so it never understands a concrete implementation, only an Interface.

Comparing Project Structures

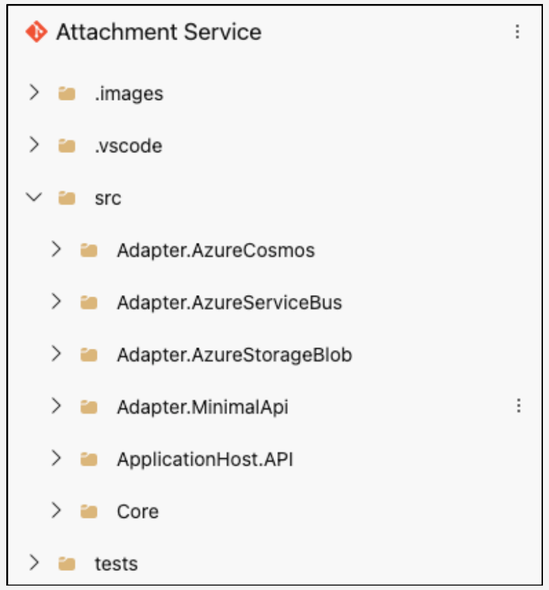
.NET projects historically take a more technology or layered based approach to architecting an application. Without a guiding style, applications tend to group functionality based on the technology being used. Coupled with nonstandard patterns across teams, you generally end up with code bases that don't share a common look and feel.

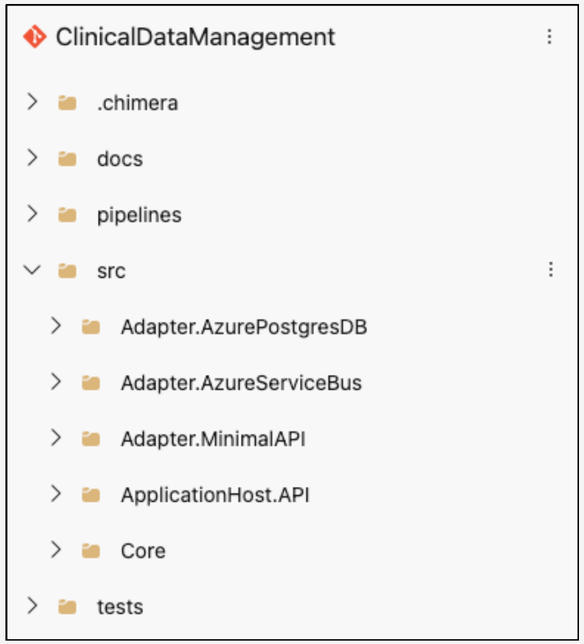


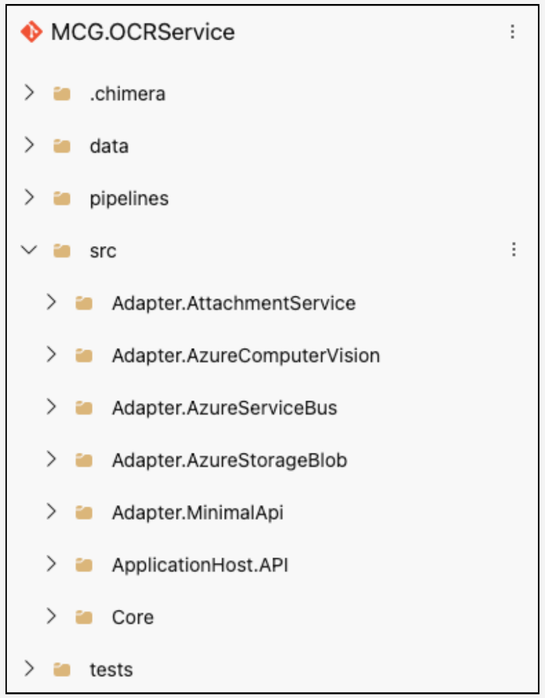


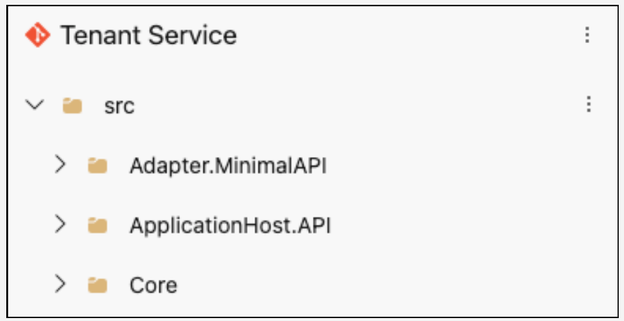


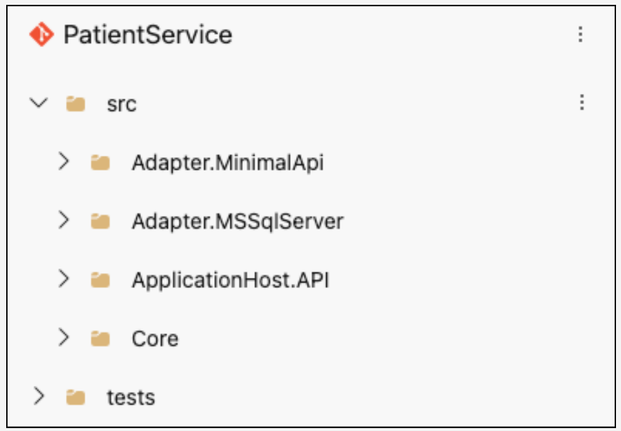
Comparing Project Structures











These projects all implement Hexagonal and Screaming architecture. They all share a common look and feel, making it easy to navigate, and familiar to work in.

Engineers can easily move into another code base and more effectively contribute code.

DRY? RUG Instead!

"Don't Repeat Yourself" (DRY) is a software development principle that encourages developers to avoid duplicating code in a system. The main idea behind DRY is to reduce redundancy and promote efficiency by ensuring that a ***particular piece of knowledge or logic exists in only one place*** within a codebase.

In Hexagonal, DRY needs to be relaxed slightly. particularly with DTOs, response/request objects, and models. Adopt RUG instead, Repeat Until Good*.

Adopt DRY heavily with business logic, loosen the requirements with everything else.

*FYI, I hate this acronym 😉

Let’s Build Something!

A step by step demo building an API using these patterns.

Best Practices

- **It’s OK to repeat some code in Hexagonal.**
  - Don’t repeat your business logic!
  - Models are OK to be in multiple projects, especially if they serve a purpose: JSON responses, DTOs, etc.
- **Name your projects appropriately!**
  - Port.ProjectName for “driven” projects, incoming traffic: API, Graph, CLI, etc.
  - Adapter.ProjectName for “driving” projects, outgoing traffic: DB calls, API calls, etc.
  - Project names should be clear about the implementation, IE: Port.MinimalAPI, Adapter.MySQL, etc.
- **Name your interfaces appropriately!**
  - Interface names should be about the action only: IPersistData, IFetchData, IUpdateData, etc.
  - They **should not** include implementation or infrastrcuture names: ISaveToPostgres, IAzureBlobSave, etc.
- **Slow down and think about where you are putting your code!**
  - If your model has annotations, it does not belong in Core!
  - Avoid projects that are not a Port, an Adapter, or a Host
  - Put code as close to where it serves it purpose

How This Helps With Planning

Use Cases Become Milestones

Each use case can be directly tied to a milestone. This helps organize your work and provide accurate reporting on the completion of a service. It also helps to prioritize work. Each service can be tied to an Epic.

Use Cases Drive Story Creation

Once your use cases are defined, you can break those down into individual stories: Upload Attachment => Scaffold Endpoint, Create Interfaces (which inform what Adapters you need), Implement Interfaces via Adapaters, etc.

Estimating Is Easier, and More Accurate!

You can now estimate easier, as you have a clearer understanding of how all your code integrates together. “We need to add a new use case with X requirements” => define the endpoint & request/response, define the interfaces & models needed in Core, define the Adapters you will need, estimate the LOE.

Planning Examples

We need to move from a relational database to NoSQL

- Create a new Adapter that implements the interfaces
- Write tests against the new Adapter code
- Change DI in Application Host to use the new Adapter

We need to call a new API to enrich data in our responses

- Create a new interface for calling the new API
- Create a new Adapter that implements the interface
- Update the Core logic to call the new Adapter for enrichment
- Update the Core models for new fields
- Update the response for new fields

We need to move from a REST API to a Graph API

- Create a new Adapter to host the graph server
- Update the Application Host to serve both the new graph and current API
- Update Core to support DataLoaders, field level resolvers, etc.

Q&A

Resources

Brandon Atkinson

https://www.linkedin.com/in/brandongatkinson/ - LinkedIn
https://github.com/atkinsonbg - GitHub
Hhttps://atkinsonbg.medium.com/ - Medium

Hexagonal Architecture

Hexagonal Architecture, there are always two sides to every story - Blog
Hexagonal Architecture for .NET Developers - YouTube
How to apply Hexagonal Architecture with .NET - YouTube
The missing Project that fixes everything in .NET - YouTube

Source Code

hexagonal-screaming-architecture-dotnet
Note: Each section of the talk is in its own branch so you can move along and see the progress in stages.

Onion Architecture

Onion Architecture - Let's slice it like a Pro - Blog

.NET Minimal APIs

Minimal APIs overview - Blog
Goodbye controllers, hello Minimal APIs - YouTube

Screaming Architecture

Screaming Architecture – Clean Code Blog - Blog
The BIGGEST Folder Structure MISTAKE on .NET - YouTube
The Magical Pattern to Organize .NET Minimal APIs - YouTube