How To Approach Clean Architecture Folder Structure

SEPTEMBER 24, 2022

**READ TIME - 3 MINUTES**

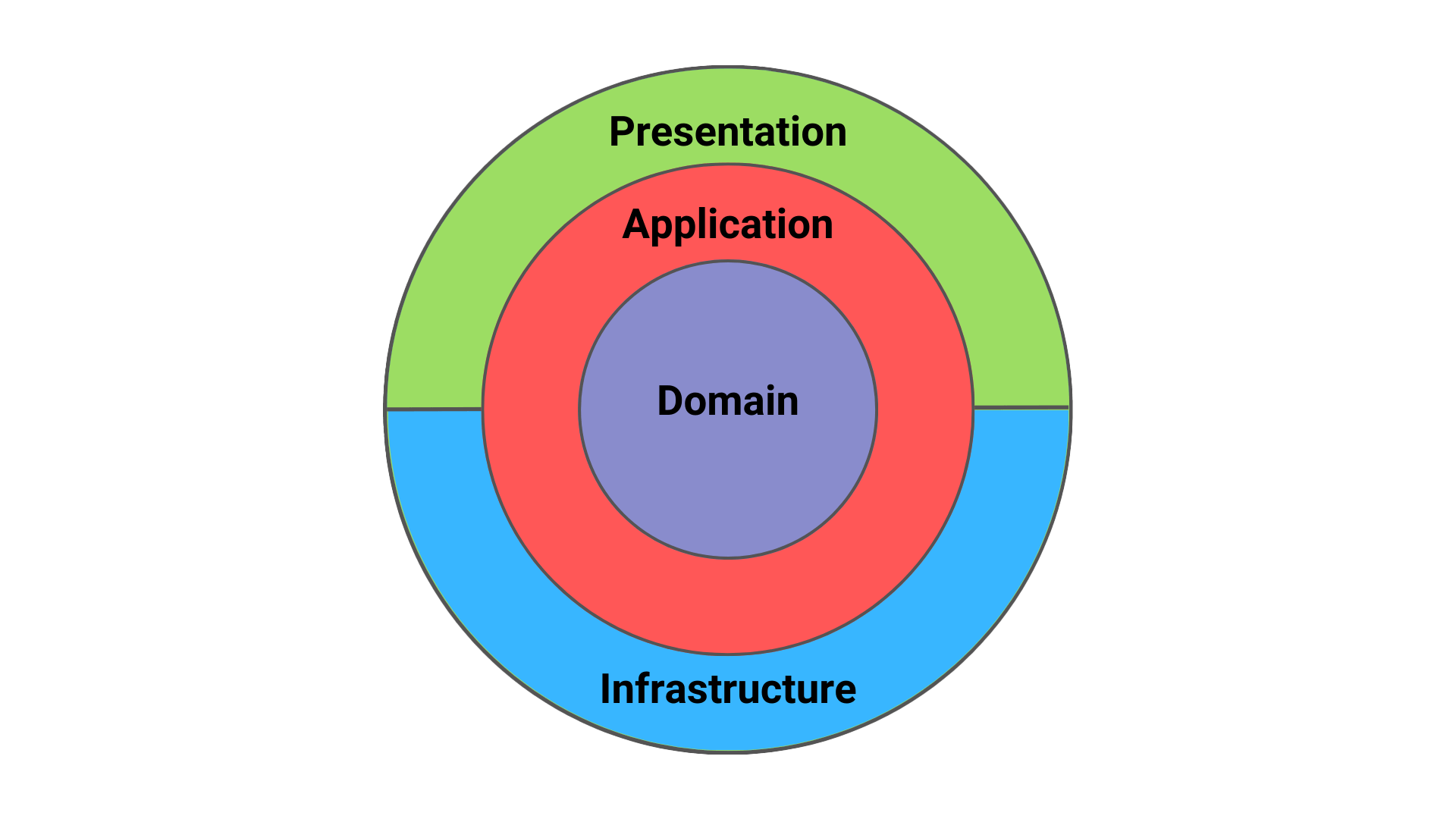
**Clean Architecture** is a popular approach to structuring your application.

It's a layered architecture that splits the project into four layers:

* [Domain](https://www.milanjovanovic.tech/blog/clean-architecture-folder-structure#domain-layer)
* [Application](https://www.milanjovanovic.tech/blog/clean-architecture-folder-structure#application-layer)
* [Infrastructure](https://www.milanjovanovic.tech/blog/clean-architecture-folder-structure#infrastructure-layer)
* [Presentation](https://www.milanjovanovic.tech/blog/clean-architecture-folder-structure#presentation-layer)

Each of the layers is typically one project in your solution.

Here's a visual representation of the **Clean Architecture**:



How do we create this in our .NET solutions?

[Domain Layer](https://www.milanjovanovic.tech/blog/clean-architecture-folder-structure#domain-layer)

The **Domain layer** sits at the core of the **Clean Architecture**. Here we define things like: entities, value objects, aggregates, domain events, exceptions, repository interfaces, etc.

Here is the folder structure I like to use:

📁 Domain

|\_\_ 📁 DomainEvents

|\_\_ 📁 Entities

|\_\_ 📁 Exceptions

|\_\_ 📁 Repositories

|\_\_ 📁 Shared

|\_\_ 📁 ValueObjects

You can introduce more things here if you think it's required.

One thing to note is that the **Domain layer** is not allowed to reference other projects in your solution.

[Application Layer](https://www.milanjovanovic.tech/blog/clean-architecture-folder-structure#application-layer)

The **Application layer** sits right above the **Domain layer**. It acts as an orchestrator for the **Domain layer**, containing the most important use cases in your application.

You can structure your use cases using services or using commands and queries.

I'm a big fan of the **CQRS** pattern, so I like to use the command and query approach.

Here is the folder structure I like to use:

📁 Application

|\_\_ 📁 Abstractions

|\_\_ 📁 Data

|\_\_ 📁 Email

|\_\_ 📁 Messaging

|\_\_ 📁 Behaviors

|\_\_ 📁 Contracts

|\_\_ 📁 Entity1

|\_\_ 📁 Commands

|\_\_ 📁 Events

|\_\_ 📁 Queries

|\_\_ 📁 Entity2

|\_\_ 📁 Commands

|\_\_ 📁 Events

|\_\_ 📁 Queries

In the **Abstractions** folder, I define the interfaces required for the **Application layer**. The implementations for these interfaces are in one of the upper layers.

For every entity in the **Domain layer**, I create one folder with the commands, queries, and events definitions.

[Infrastructure Layer](https://www.milanjovanovic.tech/blog/clean-architecture-folder-structure#infrastructure-layer)

The **Infrastructure layer** contains implementations for external-facing services.

What would fall into this category?

* Databases - PostgreSQL, MongoDB
* Identity providers - Auth0, Keycloak
* Emails providers
* Storage services - AWS S3, Azure Blob Storage
* Message queues - Rabbit MQ

Here is the folder structure I like to use:

📁 Infrastructure

|\_\_ 📁 BackgroundJobs

|\_\_ 📁 Services

|\_\_ 📁 Email

|\_\_ 📁 Messaging

|\_\_ 📁 Persistence

|\_\_ 📁 EntityConfigurations

|\_\_ 📁 Migrations

|\_\_ 📁 Repositories

|\_\_ #️⃣ ApplicationDbContext.cs

|\_\_ 📁 ...

I place my **DbContext** implementation here if I'm using **EF Core**.

It's not uncommon to make the Persistence folder its project. I frequently do this to have all database facing-code inside of one project.

[Presentation Layer](https://www.milanjovanovic.tech/blog/clean-architecture-folder-structure#presentation-layer)

The **Presentation layer** is the entry point to our system. Typically, you would implement this as a Web API project.

The most important part of the **Presentation layer** is the **Controllers**, which define the API endpoints in our system.

Here is the folder structure I like to use:

📁 Presentation

|\_\_ 📁 Controllers

|\_\_ 📁 Middlewares

|\_\_ 📁 ViewModels

|\_\_ 📁 ...

|\_\_ #️⃣ Program.cs

Sometimes, I will move the **Presentation layer** away from the actual Web API project. I do this to isolate the **Controllers** and enforce stricter constraints. You don't have to do this if it is too complicated for you.