Clean Architecture with .NET Core: Getting Started

POSTED ON FEBRUARY 5, 2020 BY JASON TAYLOR

Over the past two years, I've travelled the world teaching programmers how to build enterprise applications using Clean Architecture with .NET Core. I started by providing a sample solution using the iconic Northwind Traders database. Recently, I've developed a new Clean Architecture Solution Template for .NET Core.

This post provides an overview of Clean Architecture and introduces the new Clean Architecture Solution Template, a .NET Core Project template for building applications based on Angular, ASP.NET Core 3.1, and Clean Architecture.

Let's start with an overview of Clean Architecture.

Overview

With Clean Architecture, the **Domain** and **Application** layers are at the centre of the design. This is known as the **Core** of the system.

The **Domain** layer contains enterprise logic and types and the **Application** layer contains business logic and types. The difference is that enterprise logic could be shared across many systems, whereas the business logic will typically only be used within this system.

Core should not be dependent on data access and other infrastructure concerns so those dependencies are inverted. This is achieved by adding interfaces or abstractions within **Core** that are implemented by layers outside of **Core**. For example, if you wanted to implement the Repository pattern you would do so by adding an interface within **Core** and adding the implementation within **Infrastructure**.

All dependencies flow inwards and **Core** has no dependency on any other layer. **Infrastructure** and **Presentation** depend on **Core**, but not on one another.

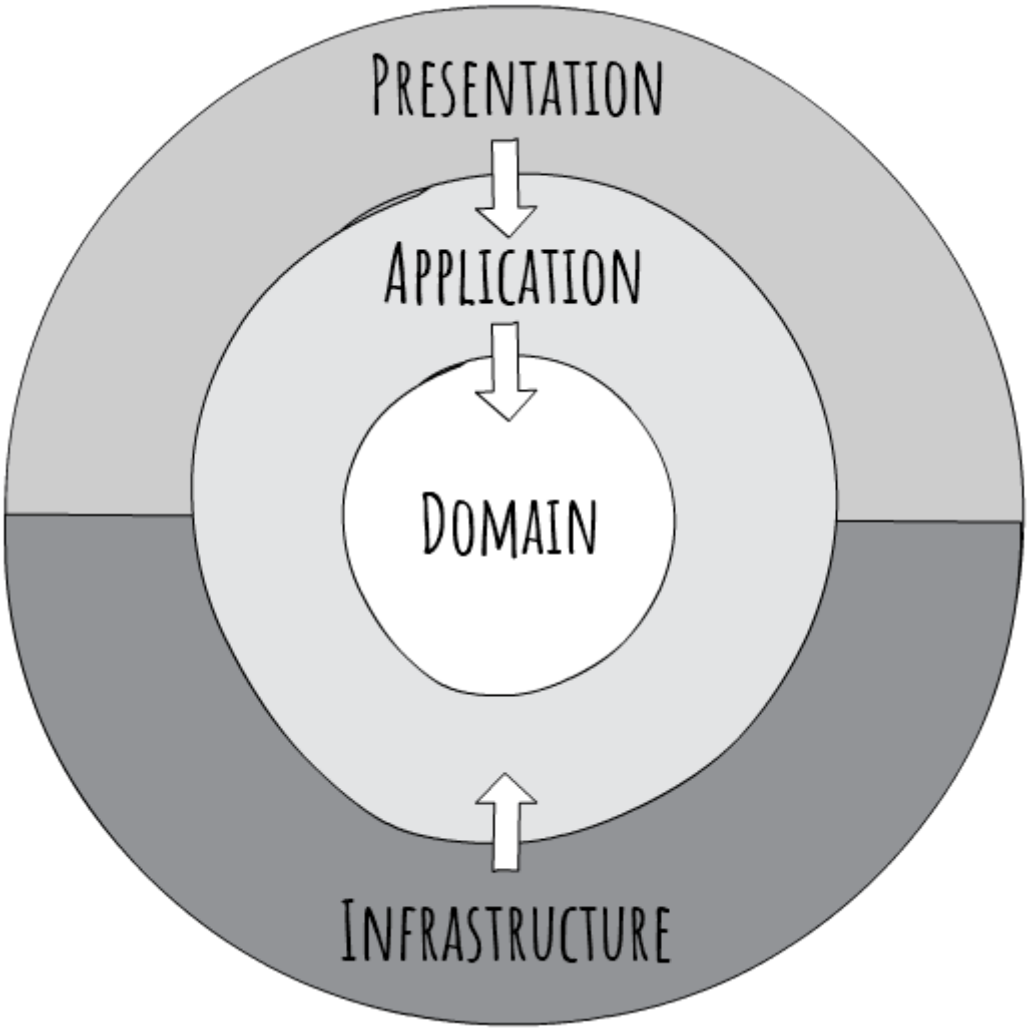


Figure: Clean Architecture Diagram

This results in architecture and design that is:

- **Independent of frameworks** it does not require the existence of some tool or framework
- **Testable** easy to test – **Core** has no dependencies on anything external, so writing automated tests is much easier
- **Independent of UI** logic is kept out of the UI so it is easy to change to another technology – right now you might be using Angular, soon Vue, eventually Blazor!
- **Independent of the database** data-access concerns are cleanly separated so moving from SQL Server to CosmosDB or otherwise is trivial
- **Independent of anything external** in fact, **Core** is completely isolated from the outside world – the difference between a system that will last 3 years, and one that will last 20 years

In the above design, there are only three circles, you may need more. Think of this as a starting point. Just remember to keep all dependencies pointing inwards.

Let's take a look at a simple approach to getting started with the new Clean Architecture Solution Template.

Solution template

This template provides an awesome approach to building solutions based on ASP.NET Core 3.1 and Angular 8 that follow the principles of Clean Architecture. If Angular is not your thing, worry not, you can remove it with ease. In this section, you will install the template, create a new solution, and review the generated code.

Prerequisites

The first step is to ensure you meet the following prerequisites:

- .NET Core SDK (3.1 or later)
- Node.js (6 or later)

Check the .NET Core version by running this command:

```
dotnet --list-sdks
```

Check the node version by running this command:

```
node -v
```

Next, install the solution template using this command:

```
dotnet new --install Clean.Architecture.Solution.Template
```

Create a new solution

Creating a new solution is easy. Within an empty folder, run the following command:

```
dotnet new ca-sln
```

The following message will be displayed:

```
The template "Clean Architecture Solution" was created successfully.
```

This command will create a new solution, automatically namespaced using the name of the parent folder. For example, if the parent folder is named **Northwind**, then the solution will be named **Northwind.sln**, and the default namespace will be `Northwind`.

The solution is built using the Angular project template with ASP.NET Core. The ASP.NET Core project provides an API back end and the Angular CLI project provides the UI.

> **Note**
> Read Use the Angular project template with ASP.NET Core to learn more about this approach.

Launching the solution from Visual Studio 2019 is trivial, just press **F5**.

In order to launch the solution using the .NET Core CLI, a few more steps are required. You can learn more by visiting the above link, but I'll include the information here for completeness.

First, you will need an environment variable named `ASPNETCORE_Environment` with a value of `Development`. On Windows, run `SET ASPNETCORE_Environment=Development`. On Linux or macOS, run `export ASPNETCORE_Environment=Development`.

Next, run the following command from the solution folder:

```
cd src/WebUI
dotnet build
```

> **Note**
> The initial build will take a few minutes, as it will also install required client-side packages. Subsequent builds will be much quicker.

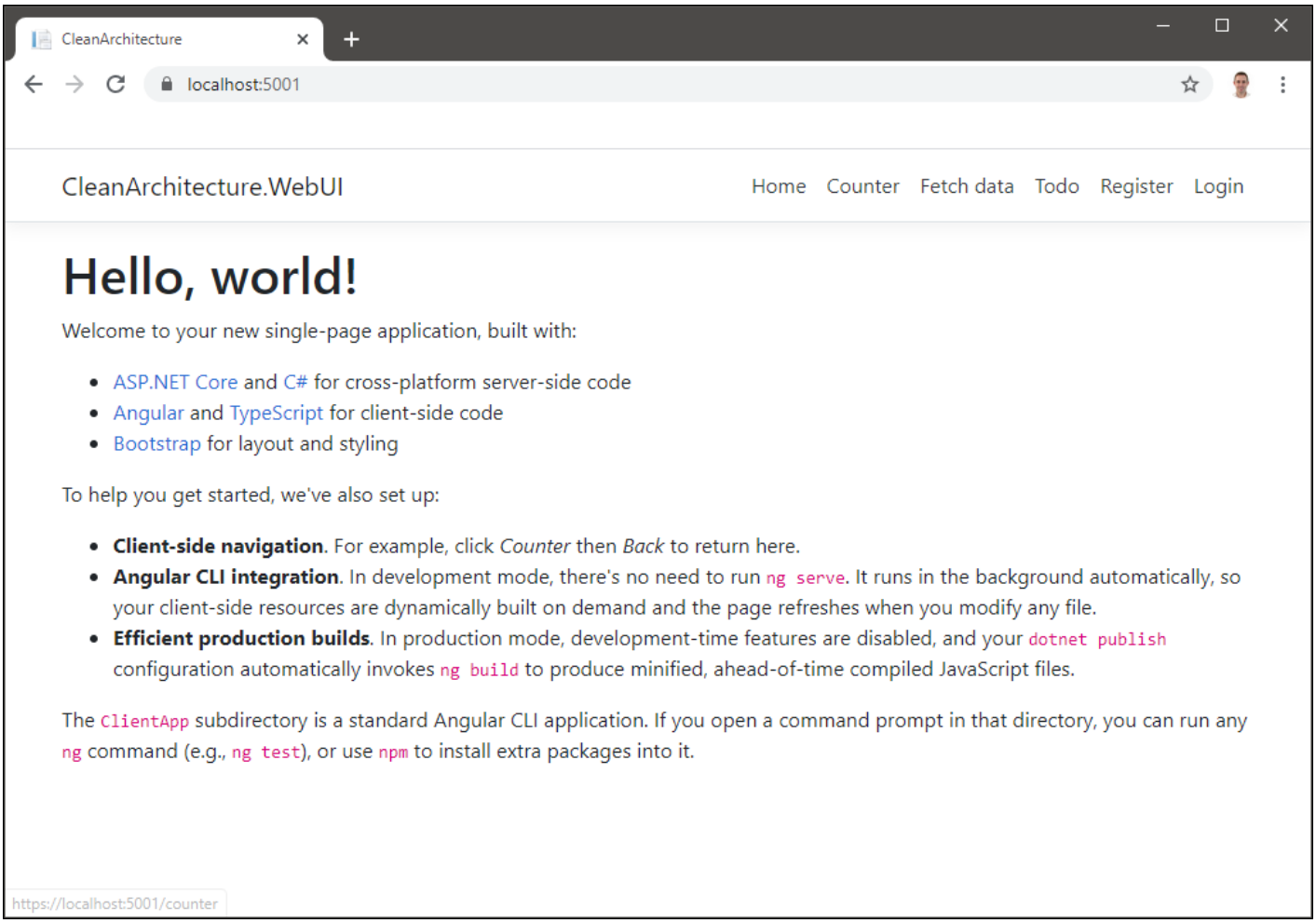
Then run `dotnet run` to start the application. The following message will be displayed:

```
Now listening on: https://localhost:port
```

The port is usually 5001. Open the web site by navigating to https://localhost:port.

> **Note**
> You will also see a message similar to the following:
> *NG Live Development Server is listening on localhost:port, open your browser on http://localhost:port*
> Ignore this message, it's **not** the URL for the combined ASP.NET Core and Angular CLI application

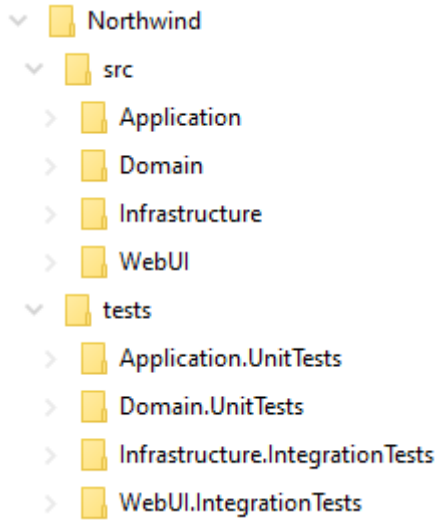
If everything was successful you will see the following:



Let's take a look at the structure of the newly generated solution.

Solution structure

The solution template generates a multi-project solution. For a solution named **Northwind**, the following folder structure is created:



The project names within **src** align closely to the layers of the Clean Architecture diagram, the only exception being **WebUI**, representing the **Presentation** layer.

The **Domain** project represents the Domain layer and contains enterprise or domain logic and includes entities, enums, exceptions, interfaces, types and logic specific to the domain layer. This layer has no dependencies on anything external.

The **Application** project represents the Application layer and contains all business logic. This project implements CQRS (Command Query Responsibility Segregation), with each business use case represented by a single command or query. This layer is dependent on the Domain layer but has no dependencies on any other layer or project. This layer defines interfaces that are implemented by outside layers. For example, if the application needs to access a notification service, a new interface would be added to the Application and the implementation would be created within **Infrastructure**.

The **Infrastructure** project represents the Infrastructure layer and contains classes for accessing external resources such as file systems, web services, SMTP, and so on. These classes should be based on interfaces defined within the Application layer.

The **WebUI** project represents the Presentation layer. This project is a SPA (single page app) based on Angular 8 and ASP.NET Core. This layer depends on both the Application and Infrastructure layers. Please note the dependency on Infrastructure is only to support dependency injection. Therefore **Startup.cs** should include the only reference to Infrastructure.

Tests

The tests folder contains numerous unit and integration tests projects to help get you up and running quickly. The details of these projects will be explored in a follow-up post. In the meantime, feel free to explore and ask any questions below.

Technologies

Aside from .NET Core, numerous technologies are used within this solution including:

- CQRS with MediatR
- Validation with FluentValidation
- Object-Object Mapping with AutoMapper
- Data access with Entity Framework Core
- Web API using ASP.NET Core
- UI using Angular 8
- Open API with NSwag
- Security using ASP.NET Core Identity + IdentityServer
- Automated testing with xUnit.net, Moq, and Shouldly

In follow-up posts, I'll include additional details on how the above technologies are used within the solution.

Additional resources

In this post, I have provided an overview of Clean Architecture and the new solution template. If you would like to learn more about any of these topics, take a look at the following resources:

- Clean Architecture Solution Template
- Clean Architecture with ASP.NET Core 3.0 (NDC Sydney 2019)
- Rules to Better Clean Architecture
- Clean Architecture Dev Superpowers Tour
- Clean Architecture 2-day Workshop

Thanks for reading. Please post any questions or comments below.