

DDD Onion-Architecture Microservice

Table of Contents

Clean Architecture	2
Solution Projects Organization.....	3
High Level Functional Design.....	5
Functional Design Folders Example	6
Customers API Onion Architecture	7
The Core	8
Core Specs.....	9
Core Domain.....	10
Core Application	11
Outer Layers.....	12
Outer Layer Persistence.....	13
Outer Layer Infrastructure	14
Outer Layer Presentation	15
Outer Client – Extends Presentation Layer.....	16
Outer Specifications test Presentation though Client.....	16
Outer Common for Cross Cutting Concerns	17
Application High Level Composition Diagram	19
Appendix.....	20
Technologies	20
Resources.....	21

Clean Architecture

What is Clean Architecture?

Architecture that is designed for the inhabitants of the architecture...

*Not for the architect... Or the machine! **What***

is clean code?

Focus on the primary needs of the users

Attempt to build a system that mirrors the mental models of the users in code

Build only what is necessary

Build only what is necessary to solve the immediate needs of the users to achieve business value.

This helps to reduce the cost of creating the system.

Optimizes for maintainability

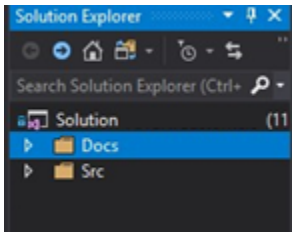
For typical applications, it takes more time to maintain it, than creating it

60 – 80% of a typical application cost comes from maintenance

So, if we optimize for maintainability, we theoretically reduce the cost to maintain the system.

This helps the company get a maximum return on investment.

Solution Projects Organization

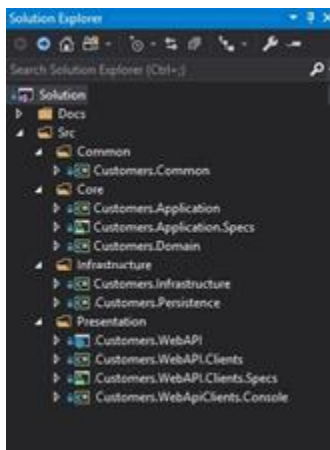


Docs

Contains supporting documents and links

1. **Readme.md** contains notes on how to get started, technologies involved, and additional technology references
2. **Any supporting documentation can be added here.**
3. **There is a single Readme.md** in each project you can use as guidance.

The **src folder** contains actual source files for various layers organized functionally.



Root

MYCO.Customers

The service fabric implementation and configuration files

Common

src/Common /MYCO.Customers.Common

Provides an interface into cross cutting concerns that is aspects of the project that all components depend on.

Core **src /Core/MYCO.Customers.Application**

Contains abstractions in correspondence to the use cases. **src**

/Core/MYCO.Customers.Domain

Contains abstractions corresponding to the business domain.

src /Core/MYCO.Customers.Application.Specs

Contains specifications and tests corresponding to the business domain.

Infrastructure **src /Infrastructure/MYCO.Customers.Persistence**

Provides an interface into the persistence storage medium which is the database.

src /Infrastructure/MYCO.Customers.Infrastructure

Provides an interface into the operating system and 3rd party dependencies.

Presentation **src /Presentation/MYCO.Customers.WebApi**

Contains the user interfaces for the application.

src /Presentation/MYCO.Customers.WebApi.Client.Clients

Contains files for use by a CSharp or typescript client generated from the Web API's outer contract.

src /Presentation/MYCO.Customers.WebApi.Client.Specs

Contains the specifications and steps testing the use cases through the API.Clients.

High Level Functional Design

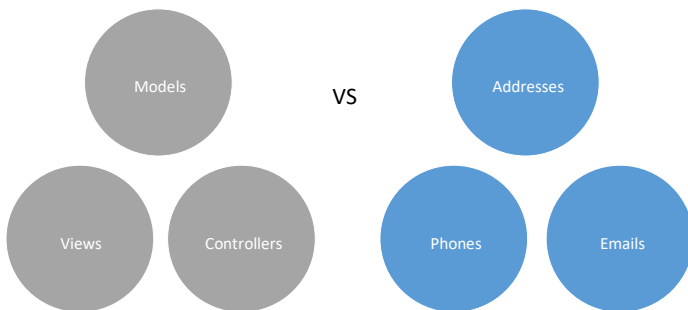
Screaming Architecture

- “The Architecture should scream the intent of the system!” -Uncle Bob

Functional Cohesion:

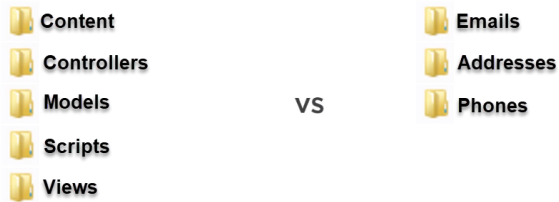
- Code is organized around the use cases of the system.
- Use cases are representations of user’s interactions within the system:
 - o GetAddress
 - o GetAddresses
 - o GetPrimaryAddress

In the solution code is **organized around the use cases of the system.**



Functional Design Folders Example

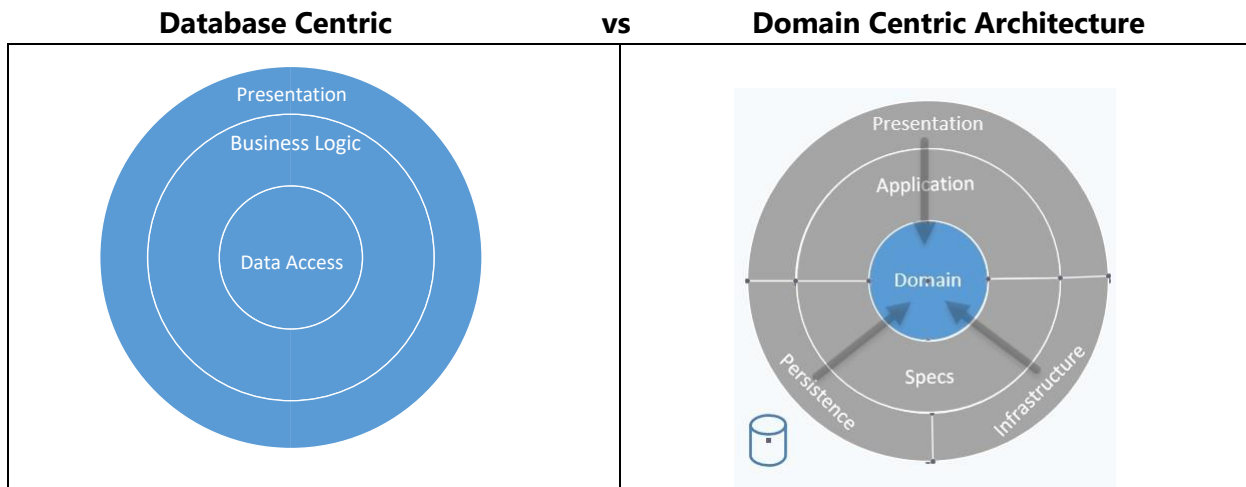
Folder Example:



- It's very difficult to understand the intent of the software on the left.
- It's much easy to understand the intent of the software on the right.
- The software on the right better models how we
 - maintain
 - navigate
 - and reason about software

Customers API Onion Architecture

The following is an overview of the Onion Architecture and how it applies to Customers API Code



Domain Centric Architecture

PROS

Focus on the Domain

Less coupling

Allows for DDD

Key Points:

- ✓ Dependencies point inward
- ✓ Outer layers can reference inner layers (dependencies point inward)
- ✓ Inner layers can't reference outer layers

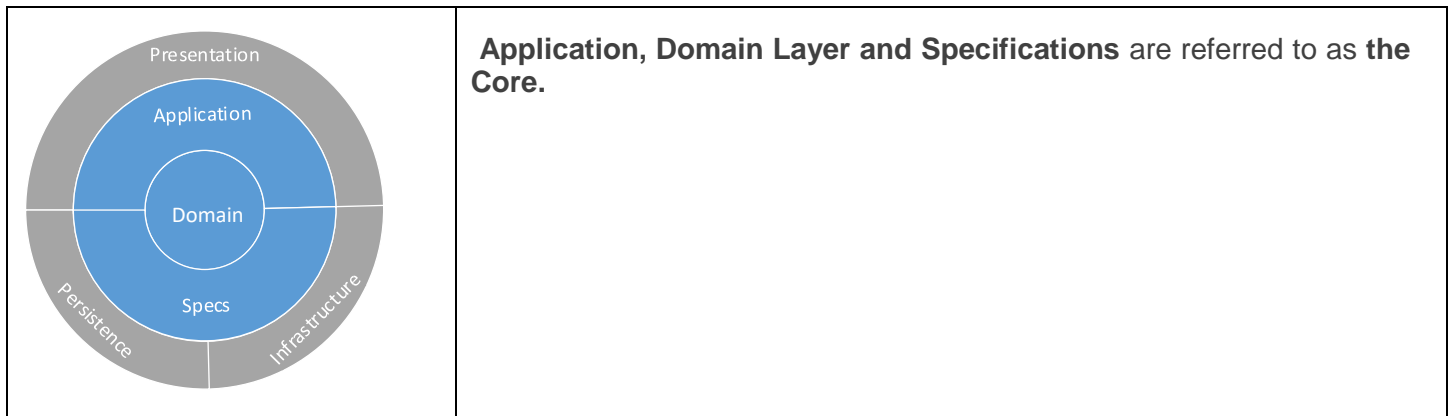
CONS (Why would we not want to use)

Sometimes change is difficult

Requires more thought

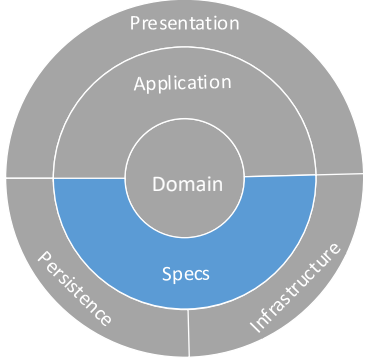
Initial higher cost

The Core



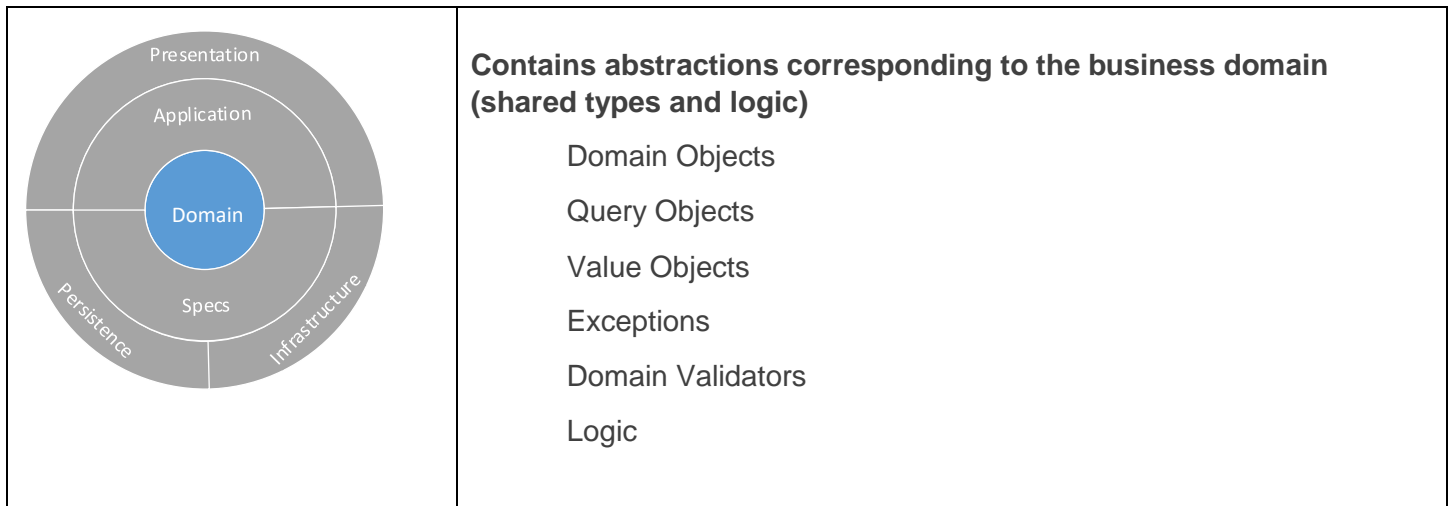
Projects: `src /Core/MYCO.Customers.Domain` and `src /Core/MYCO.Customers.Application` and `src/Core/MYCO.Customers.Application.Specs`

Core Specs

	<p>The Core Specifications contains specifications and steps corresponding to the business domain. Domain Specific Use Cases are proven here.</p> <p>Living documentations using Specflow</p>
---	--

Project:s: src/Core/MYCO.Customers.Application.Specs

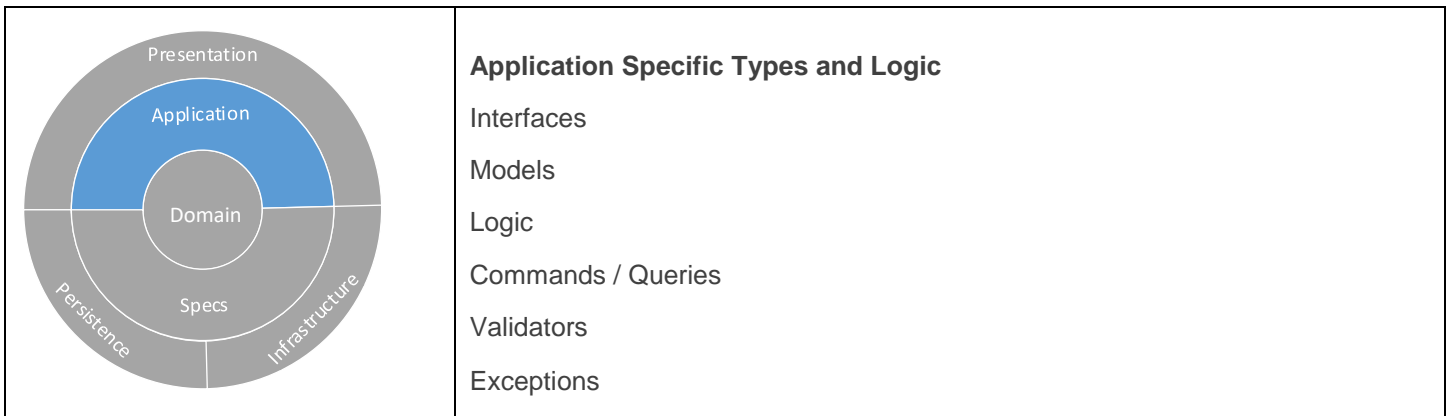
Core Domain



Note: We don't want to see a lot of data annotations in this project (show approach)

Project: `src /Core/MYCO.Customers.Domain`

Core Application



Projects: src /Core/MYCO.Customers.Domain and src /Core/MYCO.Customers.Application

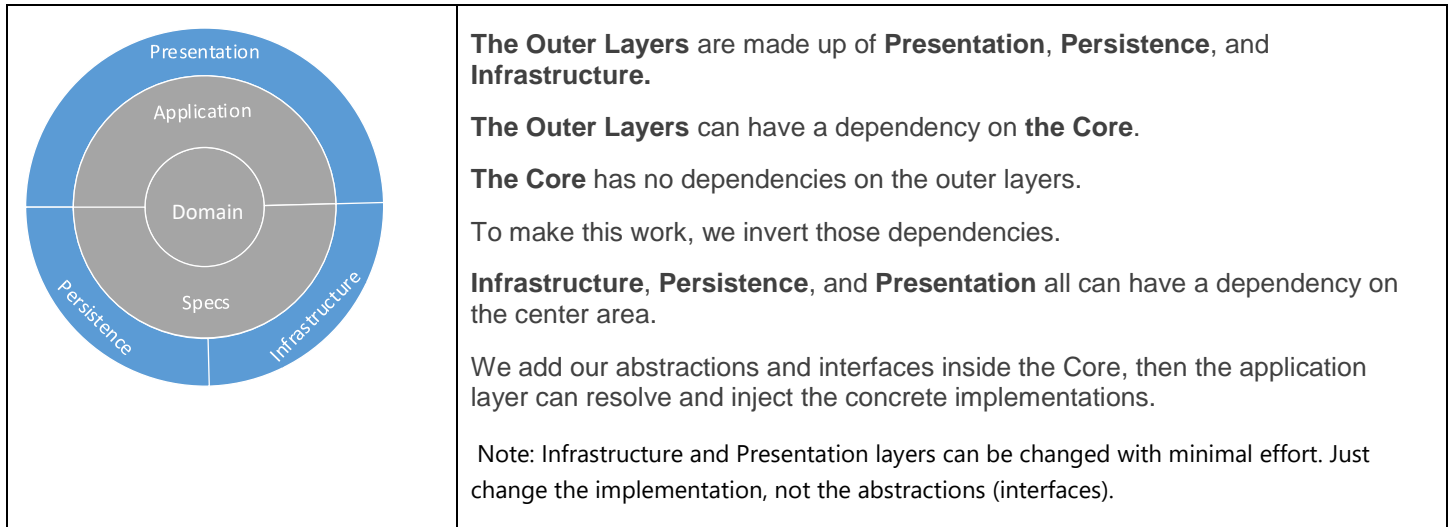
Application Layer is the most outer layer of The Core.

- We want to make sure this core is there many years from now
- Anything outside the core is exchangeable.
- Contains abstractions in correspondence to the use cases.

Commands and Queries (CQRS): Command Query Responsibility Segregation separates read (queries) from writes (commands)

- Can maximize performance and scalability
- Easy to maintain, changes only affect one command or query
- Maximizes Simplicity by encapsulating a read or a write

Outer Layers



Presentation: Contains the user interfaces for the application

- Contains references to Core, Persistence and Infrastructure
- Implements Application Layer
- Injects any external dependencies into Application Layer

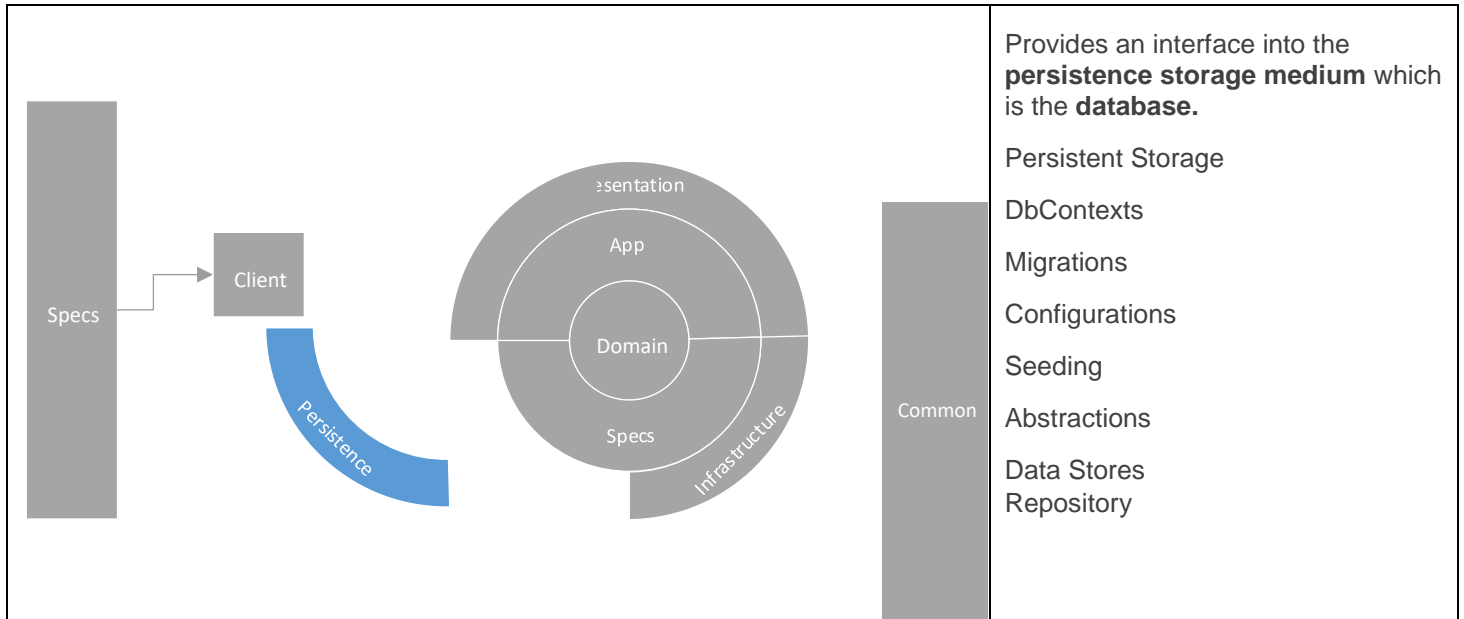
Persistence: Provides an interface into the persistence storage medium which is the database or data stores

- Persistent Storage
- Repositories
- Persistence Configurations
- DbContext(s)

Infrastructure: Provides an interface into the operating system and 3rd party dependencies

- File System
- Email / SMS
- System Clock
- Messaging Service

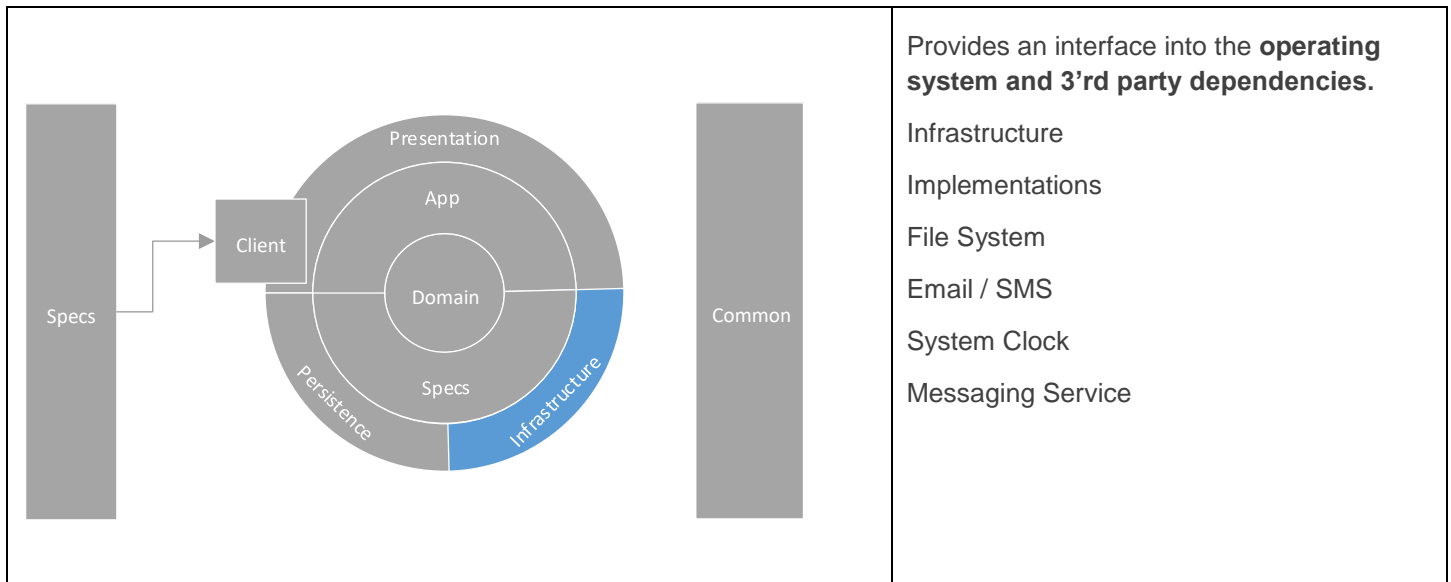
Outer Layer Persistence



Project: src /Infrastructure/MYCO.Customers.Persistence

- The Core is independent of the database
- Prefer fluent configuration over data annotations
- Keep your entities nice and clean
- Example of entity configuration

Outer Layer Infrastructure



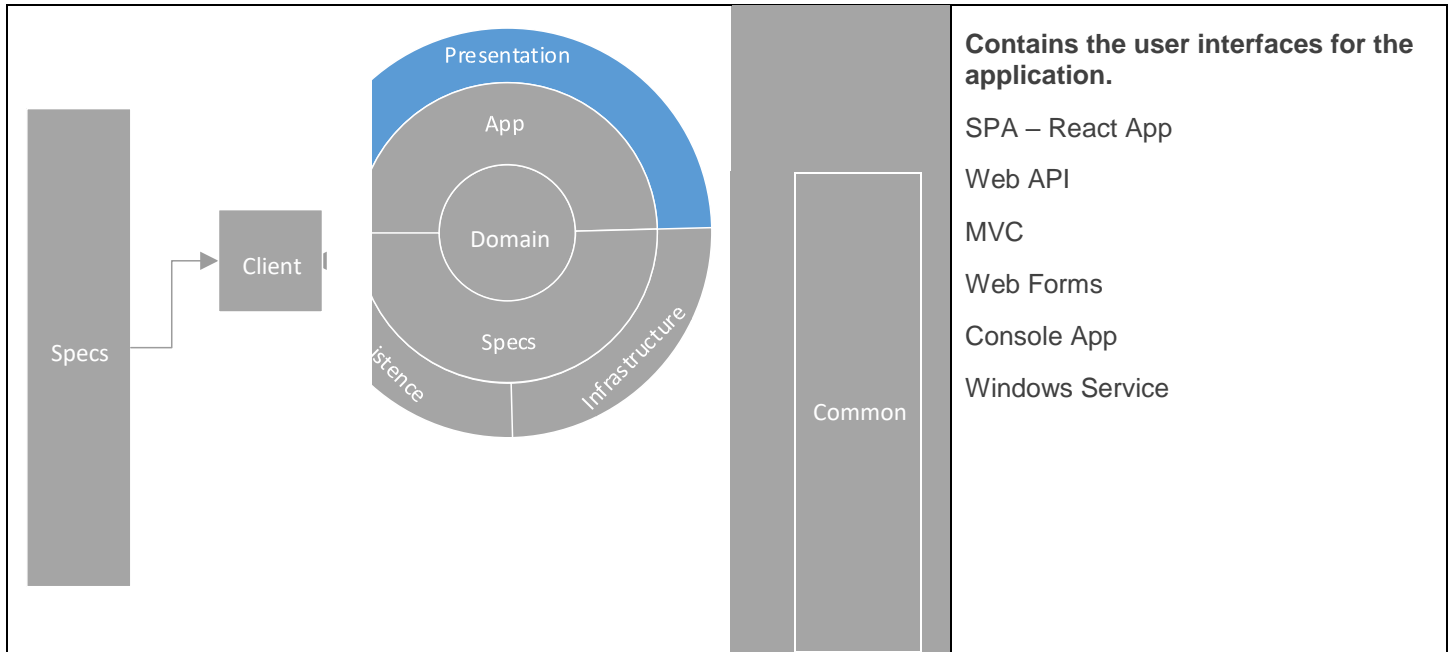
Example: If we're using SendGrid, we can switch it out with email.

Project: `src /Infrastructure/MYCO.Customers.Infrastructure`

The result is:

- They are just abstractions
- The implementations are contained in the Core
- There are no direct implementations except at runtime when we wire it up with DI ○ And this means we can switch it out (example Notification Service)
- No layers depend on Infrastructure
- Contains classes for accessing external resources ○ Such as file system, web servers, SMTP, etc.
- Implements abstractions / the interfaced defined in the application layer

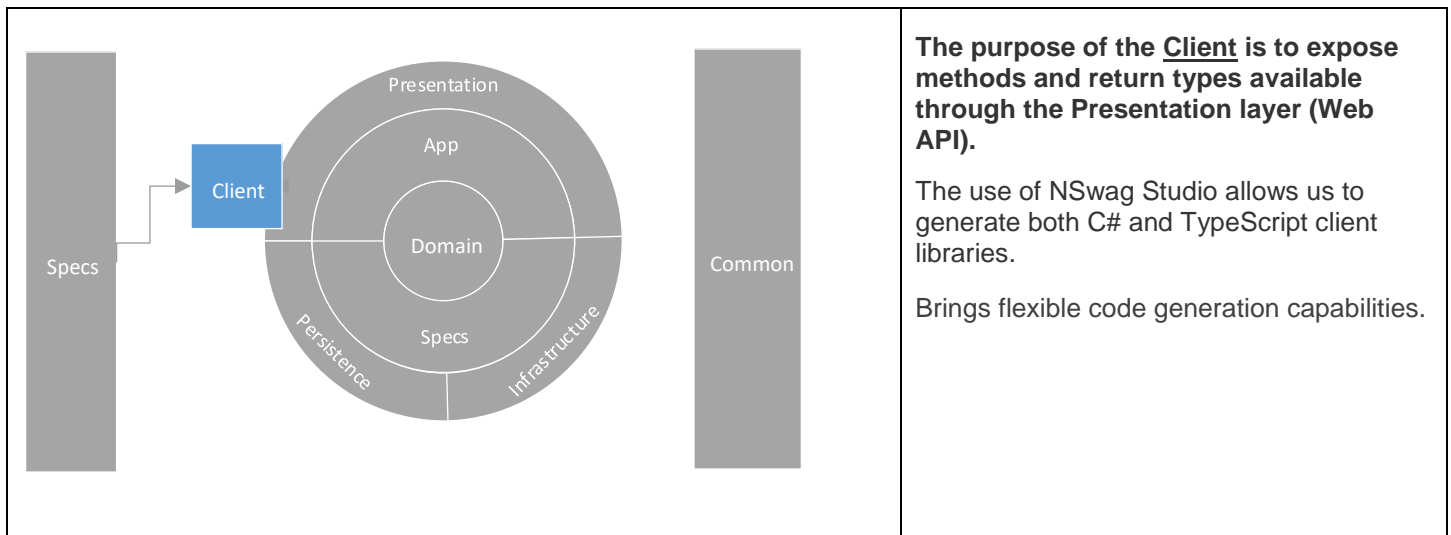
Outer Layer Presentation



Project: src /Presentation/MYCO.Customers.WebApi

- Is a RESTFUL WebApi exposing inputs and outputs for client consumption.
- Is dependent on the Application Layer to conduct Domain logic.
- Swashbuckler is used to expose the Swagger Specification (used by NSwag Studio to scaffold Client(s))
- Is responsible for injecting dependencies into The Core.

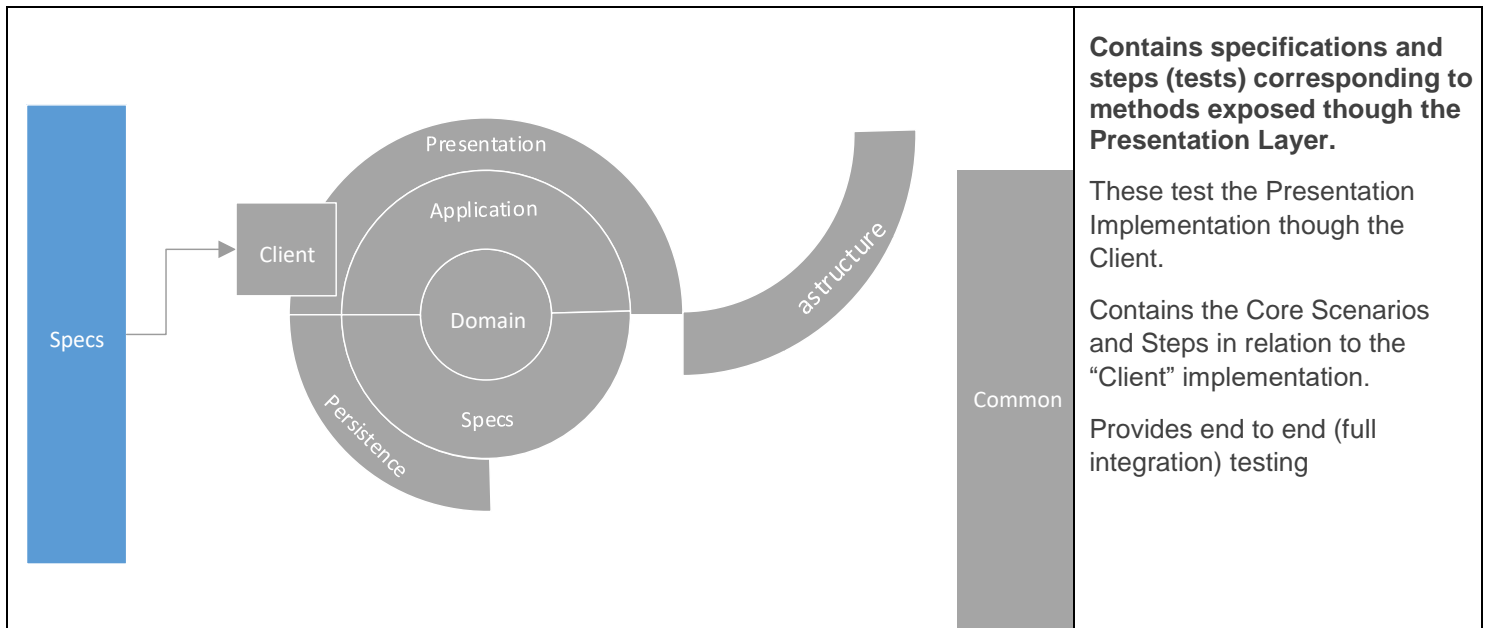
Outer Client – Extends Presentation Layer



Project: src /Presentation/MYCO.Customers.WebApi.Client

NSwag is a Swagger/OpenAPI 2.0 and 3.0 toolchain for .NET, .NET Core, Web API, ASP.NET Core, TypeScript (jQuery, AngularJS, Angular 2+, Aurelia, KnockoutJS and more) and other platforms, written in C#. The Swagger specification uses JSON and JSON Schema to describe a RESTful Web API. The NSwag project provides tools to generate Swagger specifications from existing ASP.NET Web API controllers and client code from these Swagger specifications.

Outer Specifications test Presentation though Client

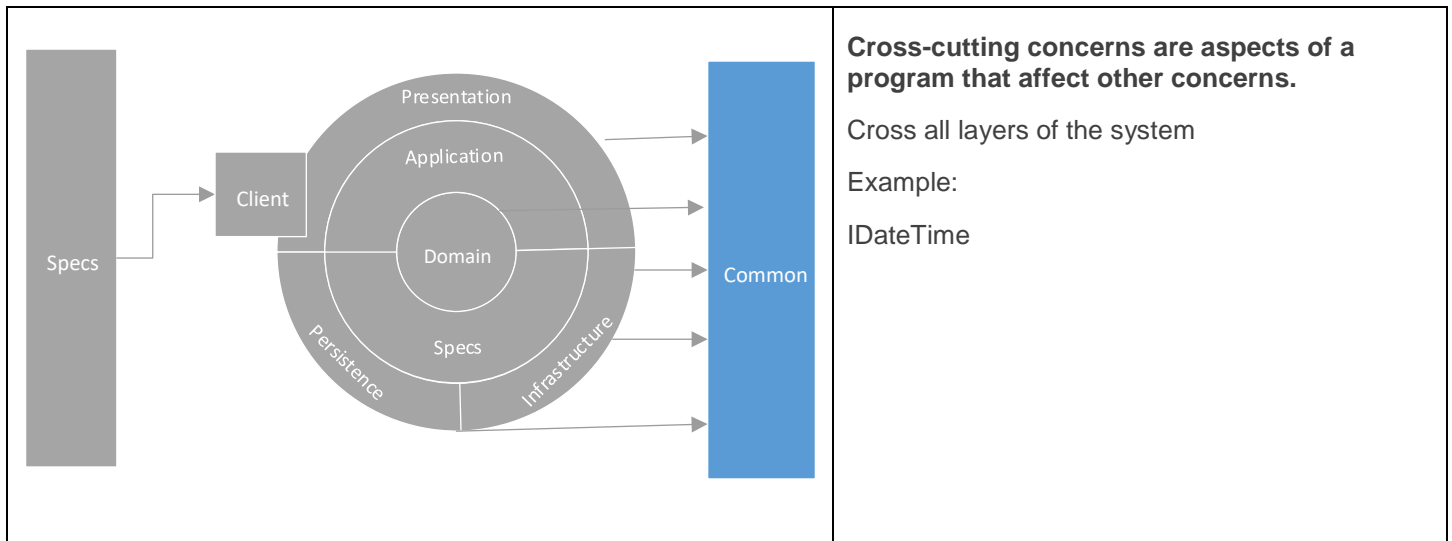


Project: src /Presentation/MYCO.Customers.WebApi.Client.Specs

We are developing a testable architecture using TDD and BDD:

- Specifications define and execute human-readable acceptance tests (use cases).
- Specifications are living documentation.
- Specs are written in an agreed upon language of the developers and stakeholders.
- We write these specs to test our WebApi Client implementation
- Comprehensive suite of tests
- Drives testable design
- Is more maintainable
- Eliminates Fear
- Acceptance criteria should be written in the language of the business.
- Should describe the functionality the business should provide.

Outer Common for Cross Cutting Concerns



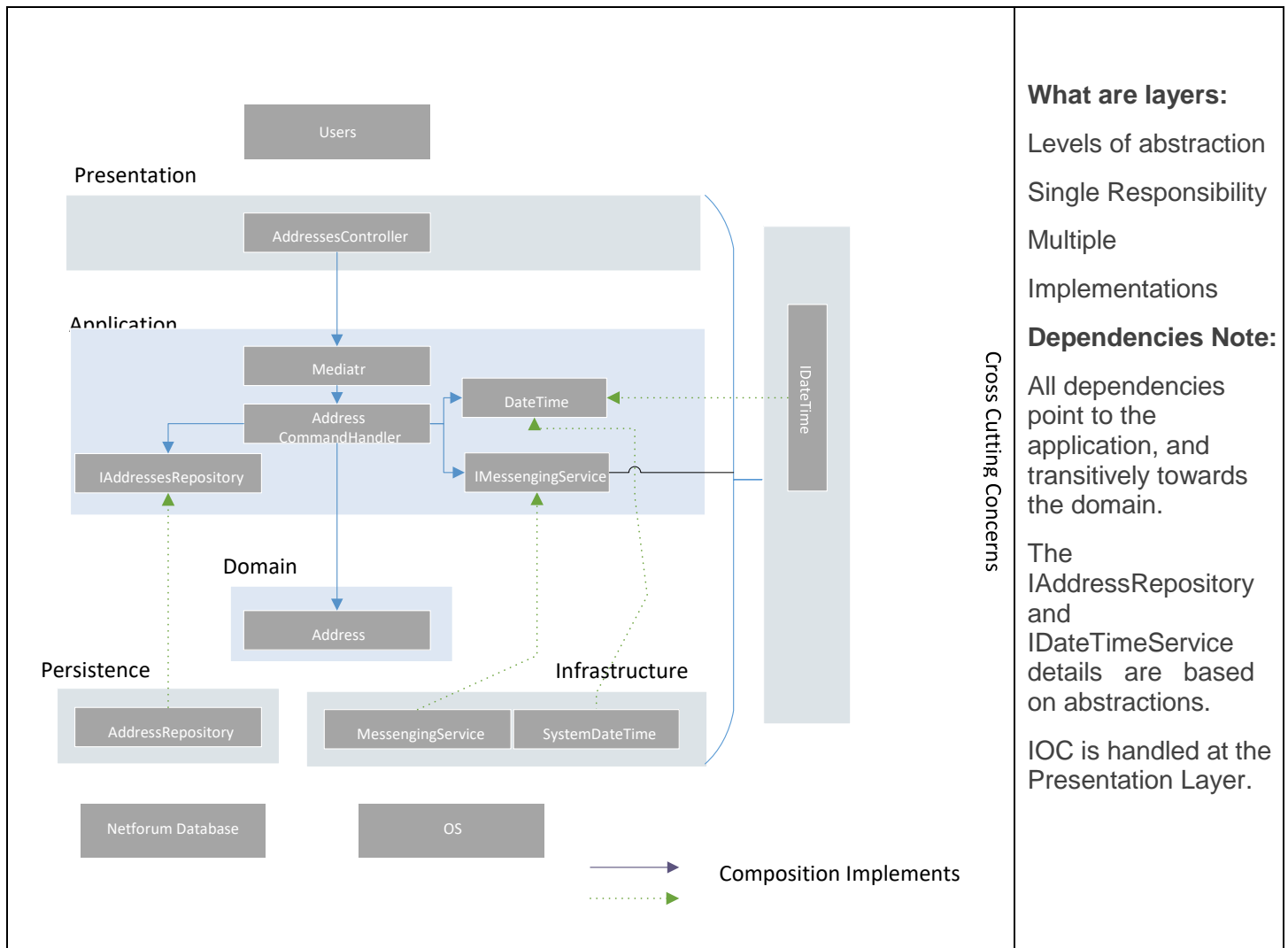
Project: src/Common /MYCO.Customers.Common

Key Points:

- All layers can have a dependency on Common
- Contains all cross-cutting concerns

Example: DateTime, Logger

Application High Level Composition Diagram



Key Points:

- All layers can have a dependency on Common
- Common contains all cross-cutting concerns

Appendix

Technologies

- VS 2017 ○ C#
- # Technologies ○ * .NET Core 2.2 ○ * ASP.NET Core 2.2 ○ #
- Additional Technologies
- References ○ *
- AspNetCore.HealthChecks.UI 2.2.2
- * CSharpGuidelinesAnalyzer 3.0.0 ○ * Entity Framework Core 2.2 ○ *
- FluentValidation.AspNetCore 8.0.101 ○ *
- Microsoft.ServiceFabric.AspNetCore.Kestrel 3.0.45 ○ *
- MicroElements.Swashbuckle.FluentValidation ○ *
- Microsoft.AspNetCore.App ○ *
- Microsoft.AspNetCore.Cors ○ *
- Microsoft.CodeAnalysis.FxCopAnalyzers 2.2.0 ○ *
- Swashbuckle.AspNetCore.SwaggerGen 4.0.1 ○ *

Swashbuckle.AspNetCore.Swagger
rUI 4.0.1 ○ * Mediatr ○ * NUnit ○
* Specflow ○ * AutoMapper ○ *
NSWAG Studio

Resources

Domain-Driven Design in Practice by Vladimir Khorikov

<https://app.pluralsight.com/library/courses/domain-driven-design-in-practice/table-of-contents>

Domain-Driven Design Fundamentals by Julie Lerman and Steve Smith

<https://app.pluralsight.com/library/courses/domain-driven-design-fundamentals/table-of-contents>

Modern Software Architecture: Domain Models, CQRS, and Event Sourcing by Dino Esposito

<https://app.pluralsight.com/library/courses/modern-software-architecture-domain-models-cqrs-event-sourcing/table-of-contents>

Northwind Trader Project By Jason Phillips

<https://github.com/JasonGT/NorthwindTraders>

Clean Architecture: Patterns, Practices, and Principles by Matthew Renze

<https://app.pluralsight.com/library/courses/clean-architecture-patterns-practices-principles/table-of-contents>