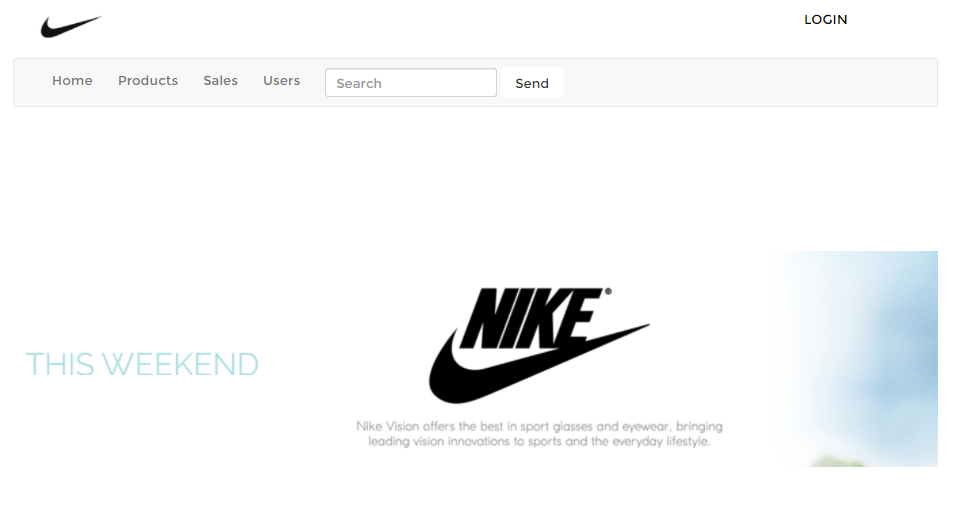
**Sales Store**

**Software architecture Documentation**

**EDGAR VALCARCEL**

**Sep 25 2020**

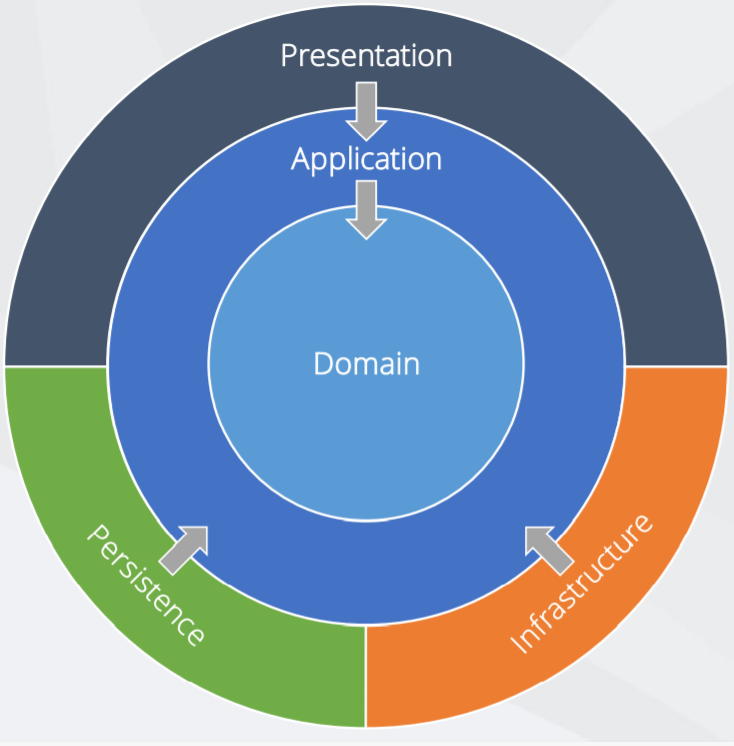


OVERVIEW

I had written a clean architecture on C# with the following benefits:

* Independent of frameworks
* Testable
* Independent of UI
* Independent of database
* Independent anything external

Diagram of the Architecture:



Using .NET Core the app will be Cross Platform, I had used:

Code First

Data Seeding

**Prerequisites**

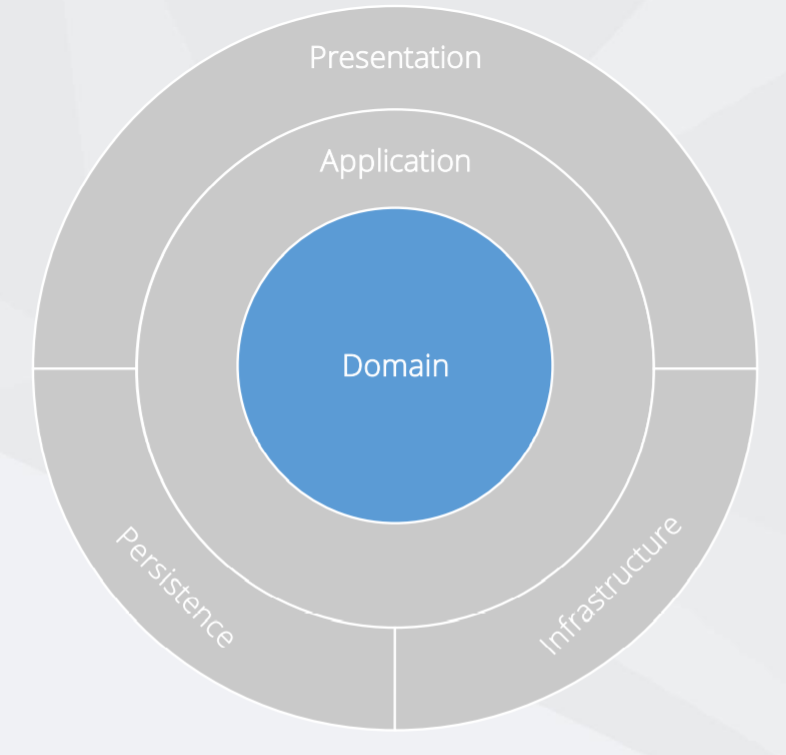
Visual Studio

.NET Core SDK (make sure to look for the SDK that supports the Visual Studio version you’re using, I’m on Visual Studio 2019 and .NET Core 3.1)

**Explanation of each layer**

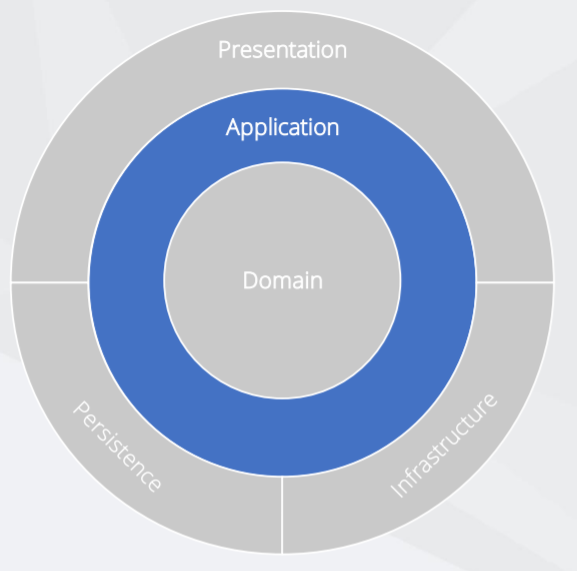
* **Domain**: contains enterprise-wide logic and types
* **Application**: contains business-logic and types
* **Infrastructure**:(including Persistence) contains all external concerns
* **Presentation** and **Infrastructure**: depend only on Application
* **Infrastructure** and Presentation components can be replaced with minimal effort

**Domain:**



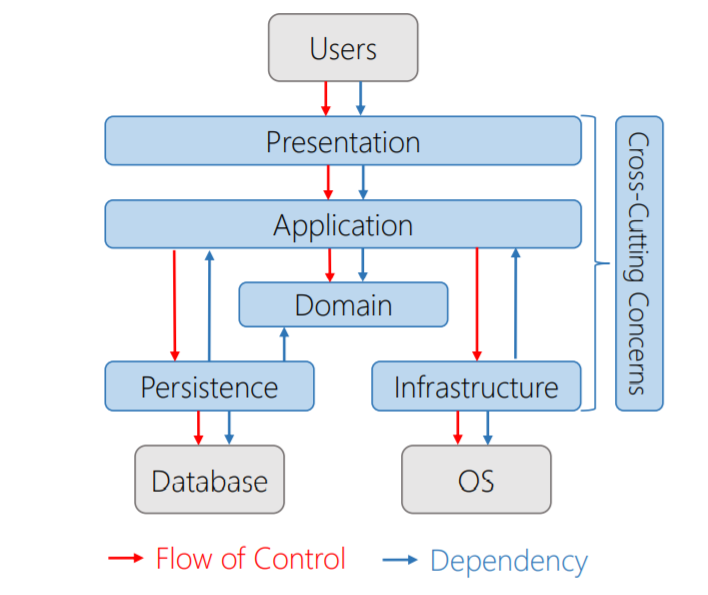
Has the Entities, Value Objects, Enumerations and Logic Exceptions, Infrastructure.Data knows about how to access our data, IoC (Inversion of Control) will help us to dependency injection.

**Application:**



This layer has the Interfaces, Models, Logic Commands and Queries, Validators and Exceptions.

I have considered Levels of abstraction, Single-Responsibility Principle, Multiple implementations, Varying rates of change.



Use case implementation on the layer:

The request start on the presentation layer, then the application layer receives the request, the flow continues with the persistence layer and the domain. All of the with the High-level application logic.

This layer knows about domain, but no knowledge of other layers. Contains interfaces for all the details, considering Dependency inversion, Inversion of control, Independent deployment, Flexibility and maintainability

Presentation Layer

Users

InvoiceController

Infrastructure IoC Layer

Application Layer

Dependency

Container

IInvoiceService

InvoiceService

Infrastructure Data Layer

IInvoiceRepository

InvoiceRepository

DB

SalesStoreContext

Infrastructure Layer

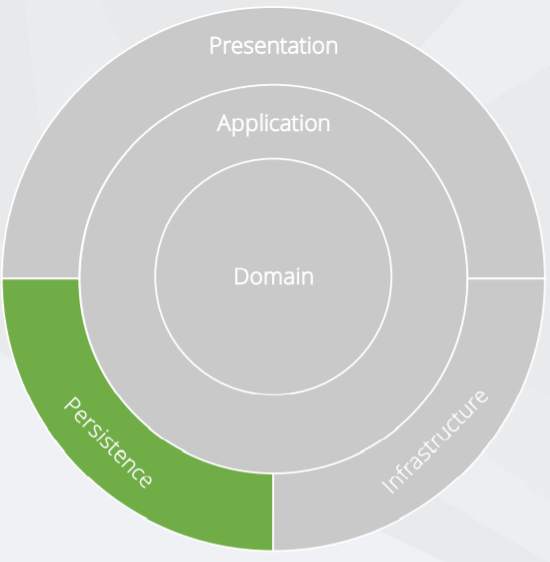


Domain Layer

The Application layer has the following pros: Focus is on use cases, Easy to understand, Follows DIP and the cons: Additional cost to maintain and a Requires extra thought.

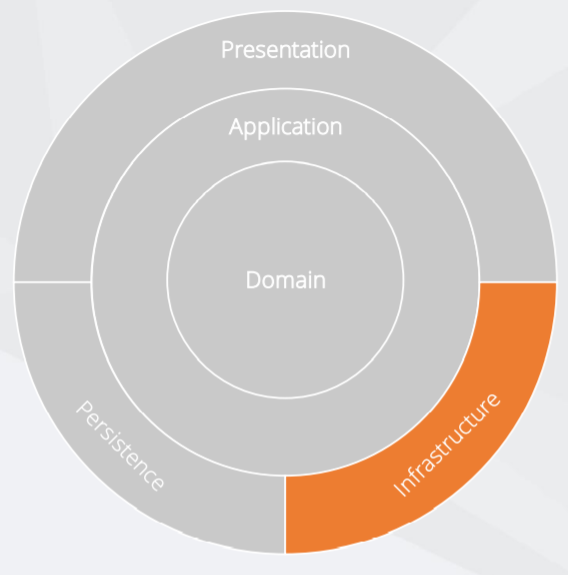
**IInvoiceService:** l used the Repository pattern to decouple the business logic and the data access layers in our application. The Repository Design Pattern in C# Mediates between the domain and the data mapping layers using a collection-like interface for accessing the domain objects. Repository Design Pattern acts as a middleman or middle layer between the rest of the application and the data access logic.

**Persistence:**



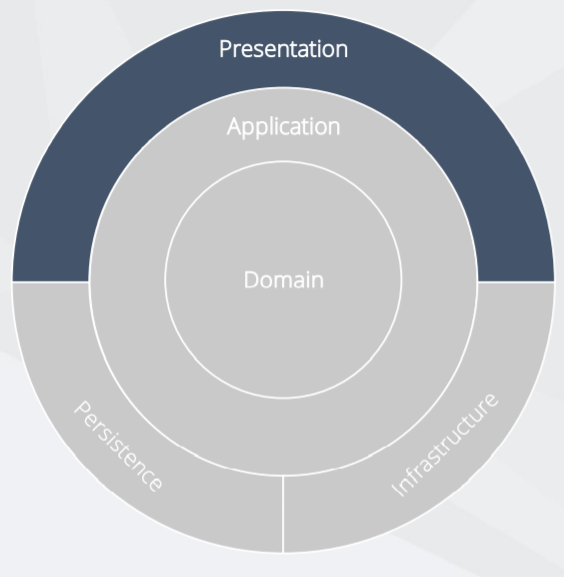
This layer has the DbContext, Migrations, Configurations, Seeding and the Abstractions

**Infrastructure:**



This layer has the Inversion of Control, Implementations as a API Clients, File System, Email / SMS, System Clock, Anything external concern.

**Presentation:**



This layer has any presentation layer as MVC .NET CORE, SPA: Angular or React, Web API, Razor Pages, MVC, Web Forms.

The Controllers should not contain any application logic, Create and consume well defined view models

Open API bridges the gap between the front end and back end.

At this point, if the MVC project, or the Presentation Layer (which has no idea about the domain entity Invoice) needs a list of invoices, it needs to talk to the *InvoiceService* (using *IInvoiceService*), and *InvoiceService* need to get it from *InvoiceRepository* (using *IInvoiceRepository*).

**Thanks to:**

* Greg Trevellick
  + https://marketplace.visualstudio.com/publishers/GregTrevellick
* Jason Taylor, SSW Solution Architect
  + https://github.com/jasontaylordev
* Steve Smith
  + https://github.com/ardalis/CleanArchitecture
* https://docs.microsoft.com/en-us/ef/core/miscellaneous/cli/dbcontext-creation