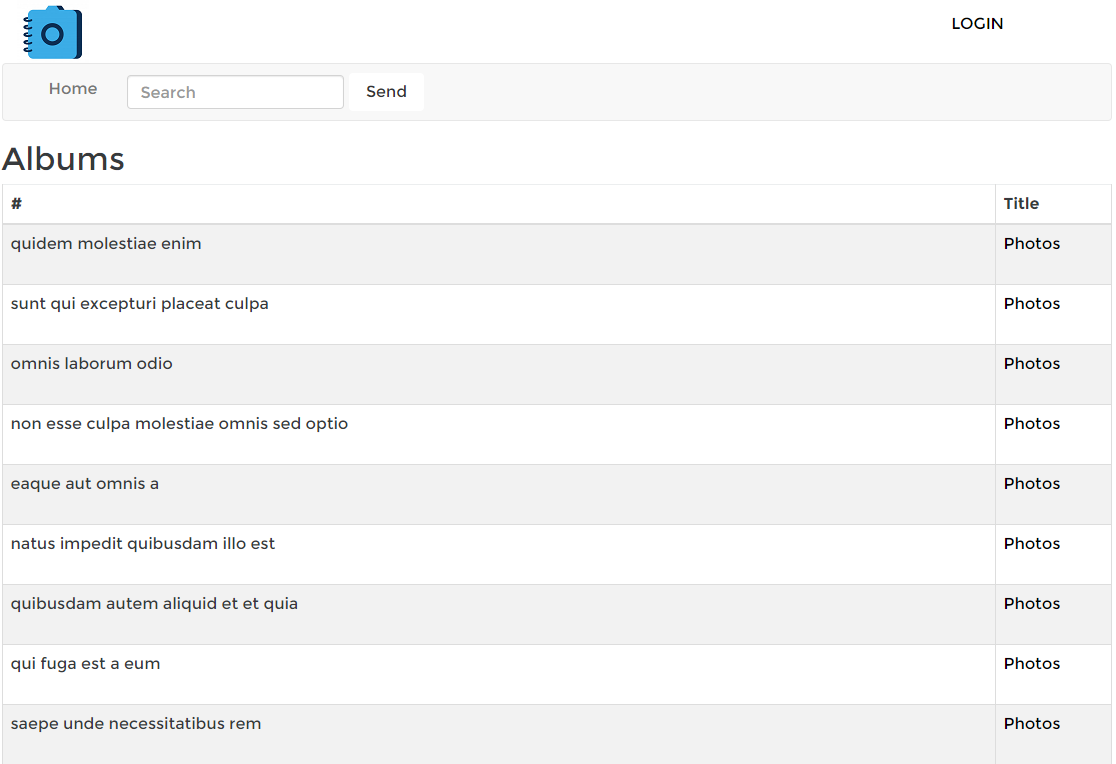
**Bertoni Albums**

**Software architecture Documentation**

**EDGAR VALCARCEL**

**Oct 16 2020**

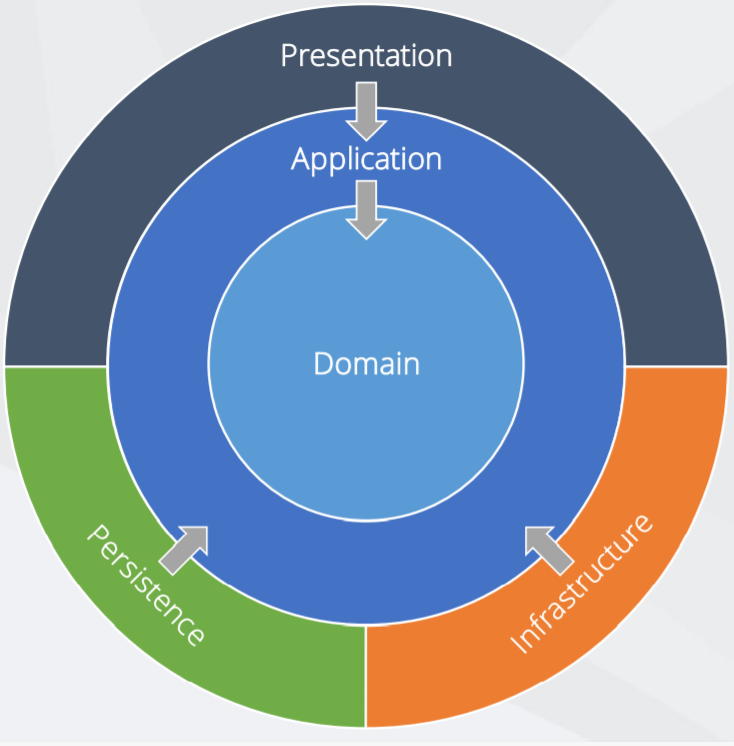


OVERVIEW

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:

Visual Studio

.NET Core SDK

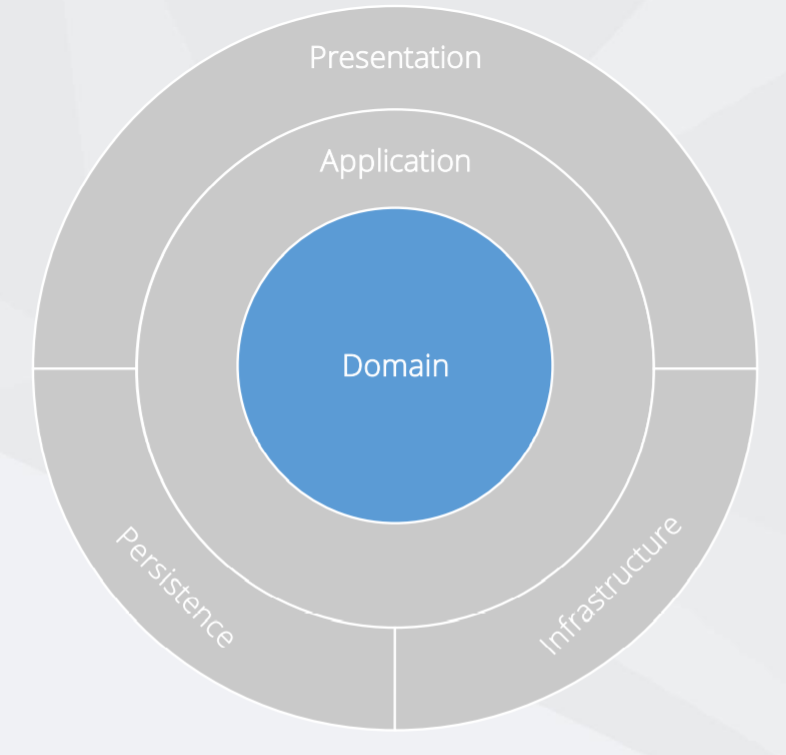
Visual Studio 2019

.NET Core 3.1

**Explanation of each layer**

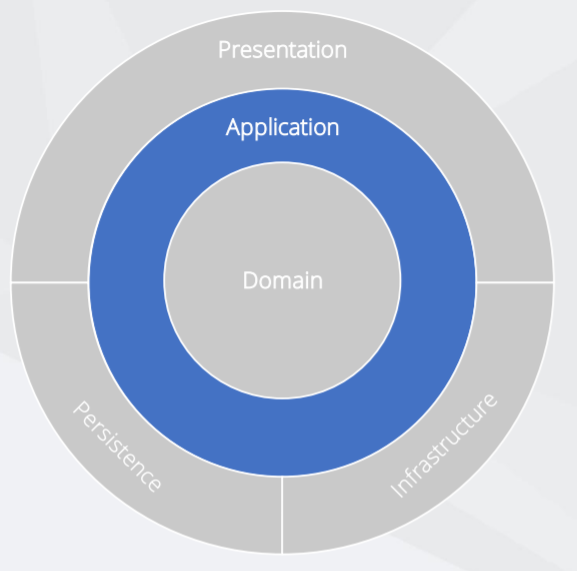
* **Domain**: contains enterprise-wide logic and types
* **Application**: contains business-logic and types
* **Infrastructure**:(including Persistence) contains all external concerns
* **Presentation** and **Infrastructure**: depend only on Application
* **Infrastructure** and Presentation components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

**Domain:**



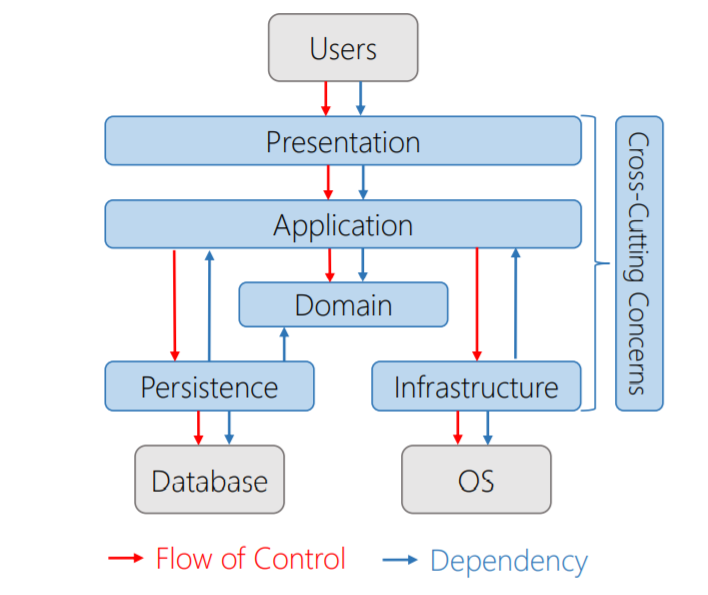
Has the Entities, Value Objects, Enumerations and Logic Exceptions, Infrastructure.Data knows about how to access our data, IoC (Inversion of Control) will help us to dependency injection.

**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

DatabaseCont

ext

Presentation Layer

Users

HomeController

Infrastructure IoC Layer

Application Layer

Dependency

Container

IAlbumService

AlbumInvoiceService

Infrastructure Data Layer

IAlbumRepository

AlbumRepository

Infrastructure Layer

Domain Layer

Models: Album,Comments

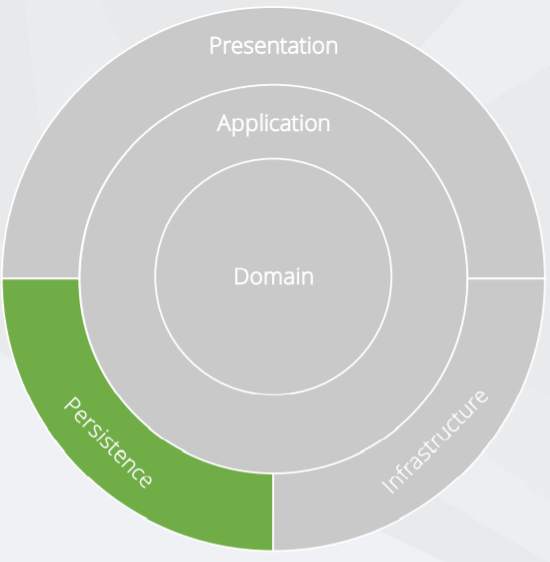
AlbumAPIConfig

https://jsonplaceholder.typicode.com

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.

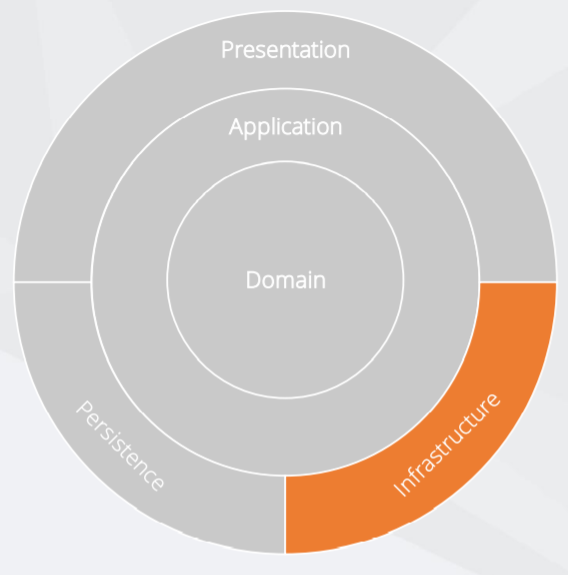
**IAlbumService:** l used the Repository pattern to decouple the business logic and the data access layers in our application. The Repository Design Pattern in C# Mediates between the domain and the data mapping layers using a collection-like interface for accessing the domain objects. Repository Design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

**Persistence:**



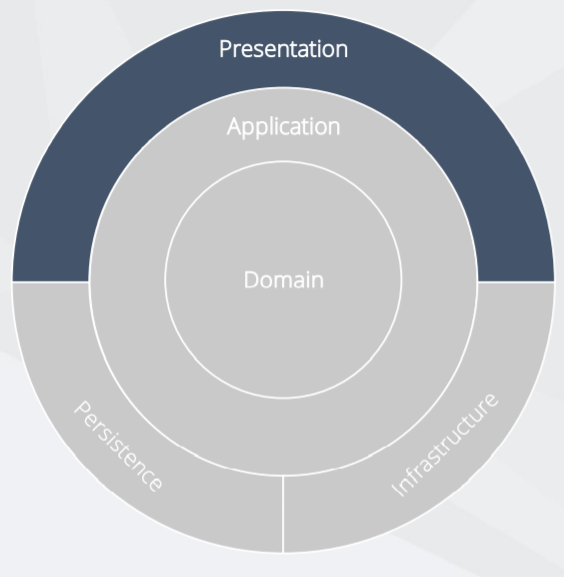
This layer has the Repositories, Migrations, Configurations, Seeding and the Abstractions, but in this project I had put the AlbumAPI Configuration and repositories classes to mediate with the API <https://jsonplaceholder.typicode.com> to retrieve data using the repositories created

**Infrastructure:**



This layer has the Inversion of Control, Implementations as a API Clients, File System, Email / SMS, System Clock, Anything external concern as the Configuration in a class to mediate with the API <https://jsonplaceholder.typicode.com>

**Presentation:**



This layer has presentation layer using MVC .NET CORE

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 Albums) needs a list of ALBUMS, it needs to talk I.E. to the *AlbumService* (using *IAlbumService*), and *AlbumService* need to get it from *AlbumRepository* (using *IAlbumRepository*).