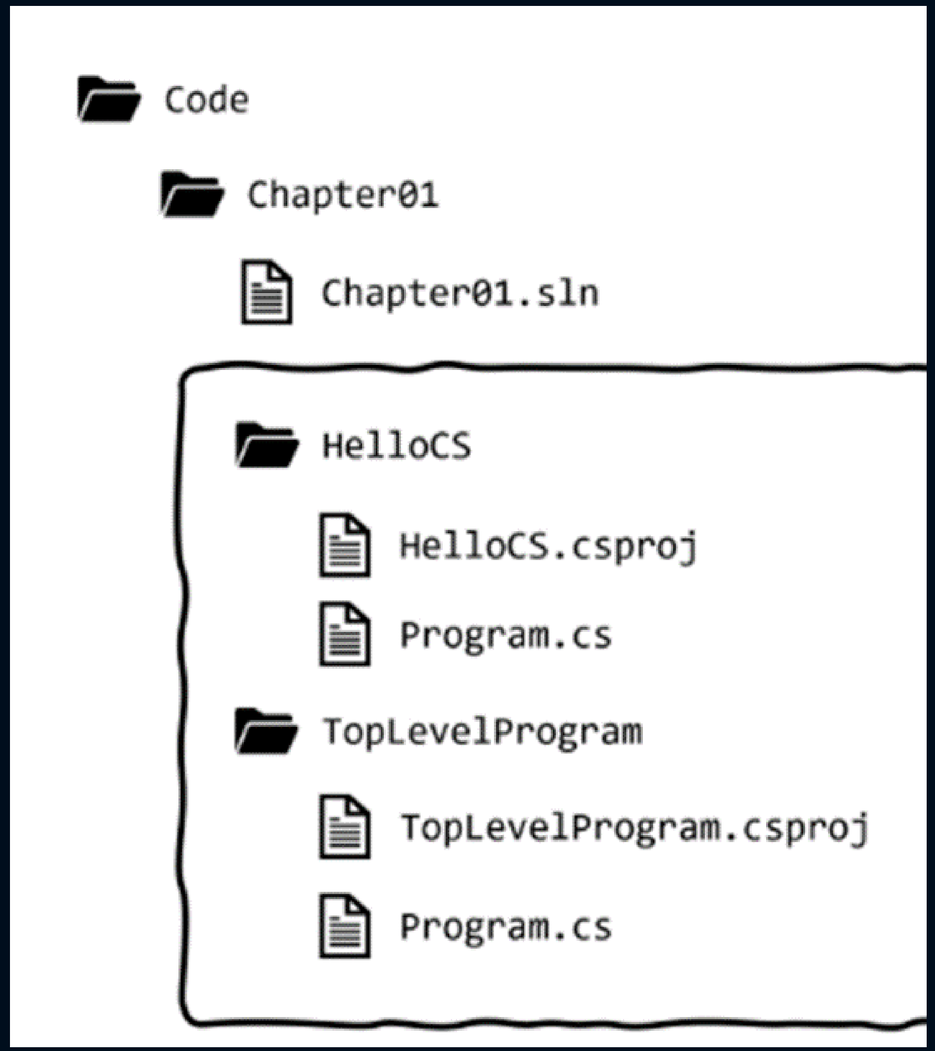
.NET Solution structure

A C# application is organized as follows:
A solution contains one or more projects.
Each project contains one or more namespaces.
The source code of an application is placed in the namespaces of the projects.

A solution is a group of projects. It can contain different types of projects like libraries, desktop , console or web applications, etc.

A project is a set of related files used to create an application, library, or tool. A project can contain source code, configuration files, resource files, and other files.

Projects can be saved in a file with the .csproj extension.
These files are of type XML.
They define the project structure, dependencies and build configurations.



.sln and .csproj File

Solutions are stored in a sln file and are used for Visual Studio to know the structure of your application.

Both solutions and projects can contain configuration files, both are applied in a different context; so configurations at the solution level are applied to all projects and configurations at the project level override other configurations and are applied at that project level.

Projects can be saved in a file with the .csproj extension, these are XML files.

They define the project structure, dependencies and build configurations

```
Microsoft Visual Studio Solution File, Format Version 12.00
# Visual Studio 2019

Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") = "MyApp", "MyApp\MyApp.csproj", "{E74F0BFD-96AB-4F3E-AE95-2D3C5A47C3EF}"
EndProject

Global
    GlobalSection(SolutionProperties) = preSolution
        HideSolutionNode = FALSE
    EndGlobalSection
    GlobalSection(ProjectConfigurationPlatforms) = postSolution
        {E74F0BFD-96AB-4F3E-AE95-2D3C5A47C3EF}.Debug|Any CPU.ActiveCfg = Debug|Any CPU
        {E74F0BFD-96AB-4F3E-AE95-2D3C5A47C3EF}.Debug|Any CPU.Build.0 = Debug|Any CPU
        {E74F0BFD-96AB-4F3E-AE95-2D3C5A47C3EF}.Release|Any CPU.ActiveCfg = Release|Any CPU
        {E74F0BFD-96AB-4F3E-AE95-2D3C5A47C3EF}.Release|Any CPU.Build.0 = Release|Any CPU
    EndGlobalSection
EndGlobal
```

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Newtonsoft.Json" Version="12.0.3" />
  </ItemGroup>
  <ItemGroup>
    <Compile Include="Program.cs" />
    <Compile Include="Utils.cs" />
  </ItemGroup>
</Project>
```

Namespaces

A Namespaces is like the specific location to find data types.

For example, ”System.Console.WriteLine” indicates the compiler where to find the ”WriteLine” method in the Namespace ”System” and the class “Console”.

Importing a namespace simplifies the code allowing access to other data types in different namespaces without needed to specify the full Namespace and allows tools like Intellisense to recognize members of the classes of these namespaces.

The image shows the using statement which helps to import classes of a namespaces.

The global using statement that works at a project level.

And the implicit global using. which in new C# template projects comes out of the box. In summary these are common classes that we use in an standard application so they are imported by default for all the project.

```
using System; // import the System namespace
```

```
global using System;
global using System.Linq;
global using System.Collections.Generic;
```

```
// <autogenerated />
global using global::System;
global using global::System.Collections.Generic;
global using global::System.IO;
global using global::System.Linq;
global using global::System.Net.Http;
global using global::System.Threading;
global using global::System.Threading.Tasks;
```

Common Namespaces

SDK	Implicitly imported namespaces

`Microsoft.NET.Sdk`	`System`
	`System.Collections.Generic`
	`System.IO`
	`System.Linq`
	`System.Net.Http`
	`System.Threading`
	`System.Threading.Tasks`

`Microsoft.NET.Sdk.Web`	Same as `Microsoft.NET.Sdk` and:
	`System.Net.Http.Json`
	`Microsoft.AspNetCore.Builder`
	`Microsoft.AspNetCore.Hosting`
	`Microsoft.AspNetCore.Http`
	`Microsoft.AspNetCore.Routing`
	`Microsoft.Extensions.Configuration`
	`Microsoft.Extensions.DependencyInjection`
	`Microsoft.Extensions.Hosting`
	`Microsoft.Extensions.Logging`

`Microsoft.NET.Sdk.Worker`	Same as `Microsoft.NET.Sdk` and:
	`Microsoft.Extensions.Configuration`
	`Microsoft.Extensions.DependencyInjection`
	`Microsoft.Extensions.Hosting`
	`Microsoft.Extensions.Logging`

References

C# project can have two types of reference: Packages and Project to Project

Dependencies managed through NuGet packages	Dependencies managed within the solution
External libraries or frameworks Managed independently from the project	References to other projects within the same solution Directly access and use types from the referenced projects
Packages provide additional functionality and features	Access classes, interfaces, and other types directly
Import namespaces and use types provided by the package	Supports code reuse and modular development within the solution
Packages are resolved and included during build or runtime	Projects are built and compiled together within the solution
Simplified dependency management	Simplified development within the solution
Easy versioning and updating of packages	Easier debugging and collaboration among solution projects
Supports modular and reusable code through external libraries	Enables fine-grained control and customization of dependencies
Enables collaboration and sharing of code through published packages	

Nuget

Nuget docs

Nuget

A package manager for .NET that simplifies the process of adding and managing third-party libraries and dependencies.

It is an essential part of the .NET ecosystem, providing a centralized repository for sharing and consuming packages.

Developers use the NuGet CLI (Command-Line Interface) or the Visual Studio NuGet Package Manager to interact with packages.

Some benefits it brings to the table are Modularity, Versioning, Discoverability, Dependency Managemente.

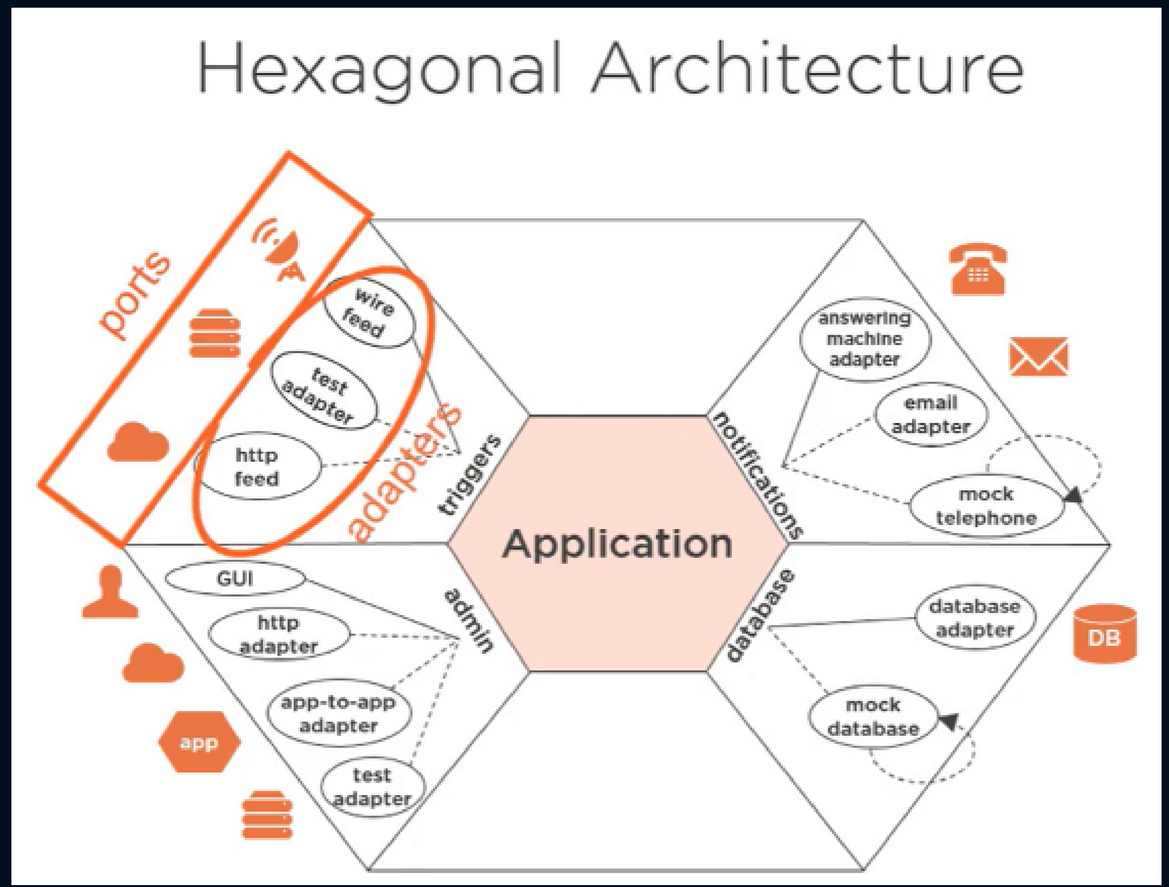
Hexagonal

Is a software architectural pattern that emphasizes the separation of concerns and the independence of the core application logic from external concerns such as databases, user interfaces, and external services.

The core of the application contains the domain model and business rules. It represents the heart of the application and should be independent of any external dependencies.

Ports: Ports define interfaces through which the core interacts with the external world. They are contracts that the core application exposes but doesn't implement. Examples of ports are repositories interfaces for data access, service interfaces for external services, and UI interfaces for user interaction.

Adapters: Adapters are implementations of the port interfaces. They are responsible for connecting the core with the external dependencies. For example, there can be database adapters, web service adapters, and user interface adapters.



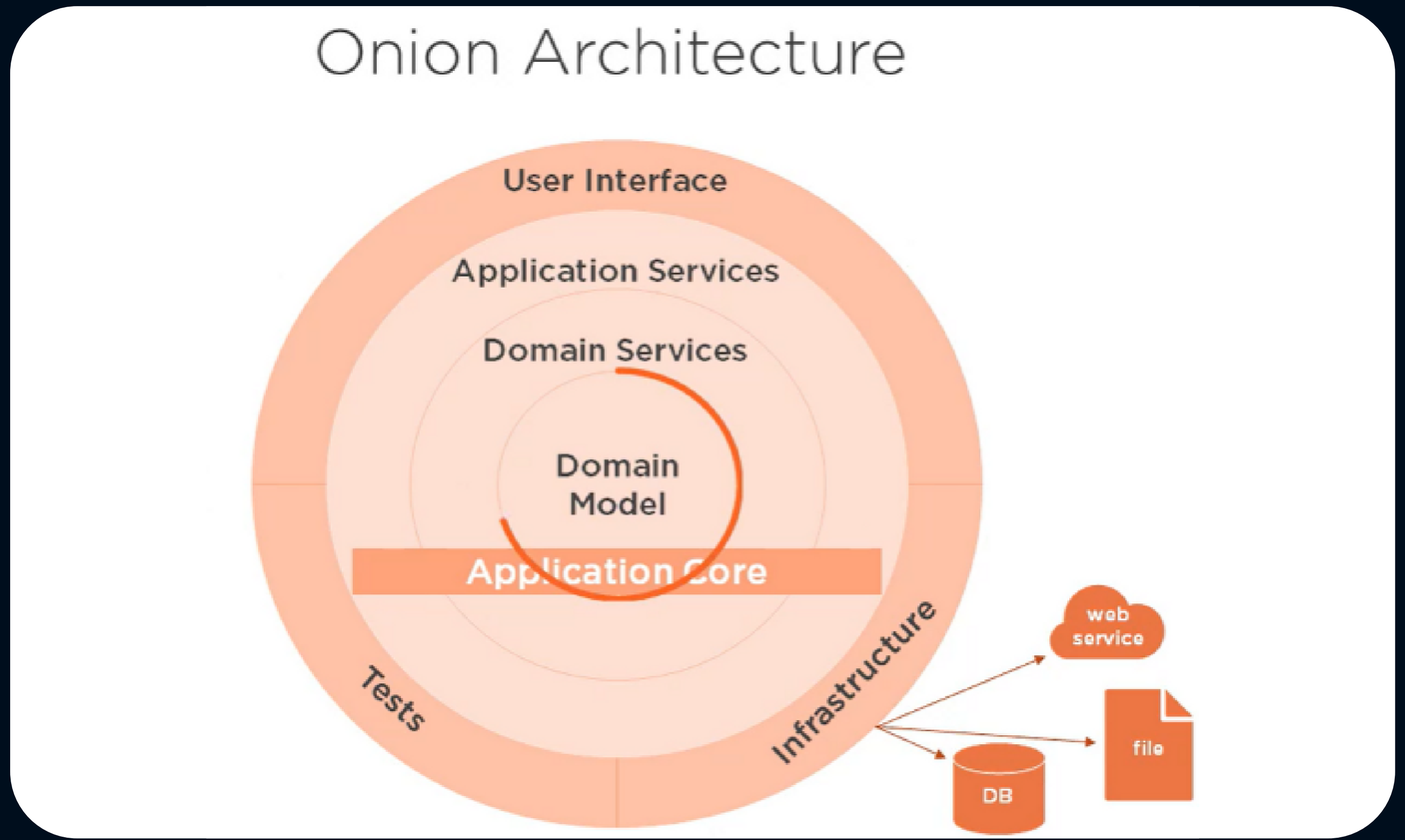
Onion

Is software architectural pattern that emphasizes the separation of concerns and the independence of the core business logic from external concerns such as databases, user interfaces, and frameworks. The main idea behind Onion Architecture is to keep the core business logic at the center, surrounded by layers of increasing abstraction, with each layer having a clear and specific responsibility.

At the center part of the Onion Architecture, the domain layer exists; this layer represents the business and behavior objects. These domain entities don't have any dependencies.
Repository Layer - This layer creates an abstraction between the domain entities and business logic of an application.

Services Layer - The Service layer holds interfaces with common operations, such as Add, Save, Edit, and Delete. Also, this layer is used to communicate between the UI layer and repository layer.

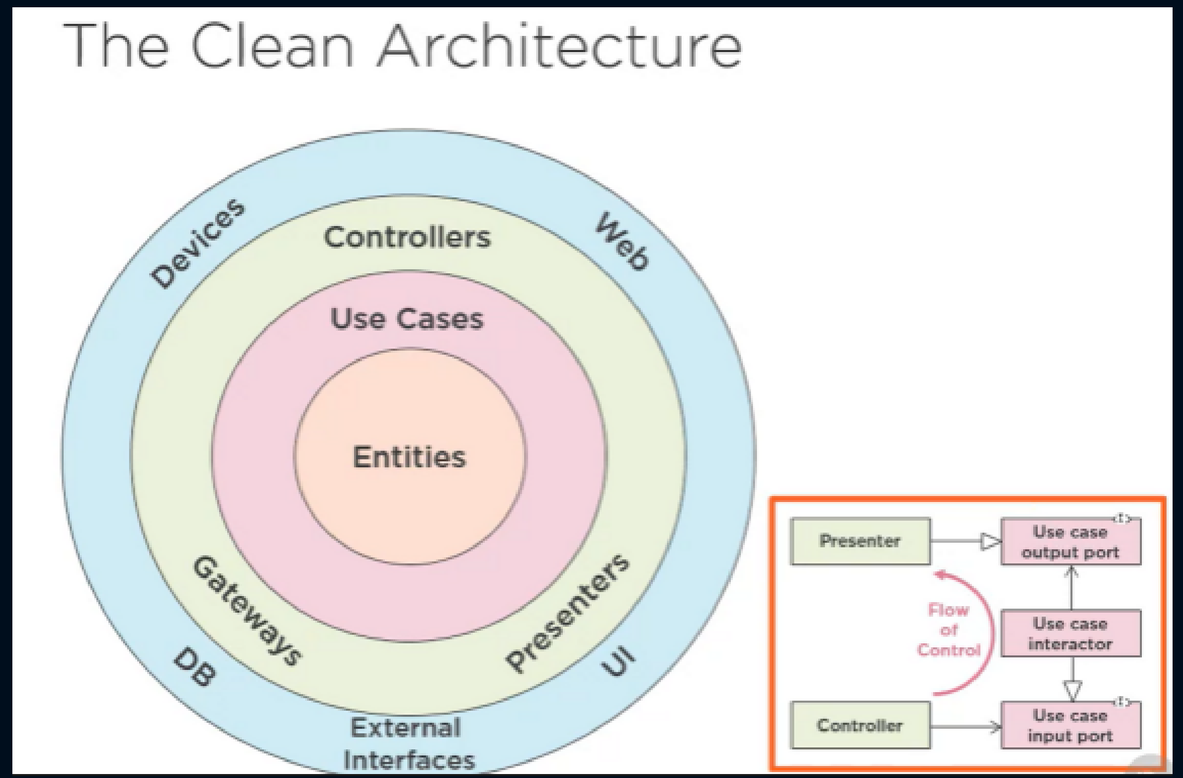
UI Layer - It's the outer-most layer, and keeps peripheral concerns like UI and tests. For a Web application, it represents the Web API or Unit Test project.



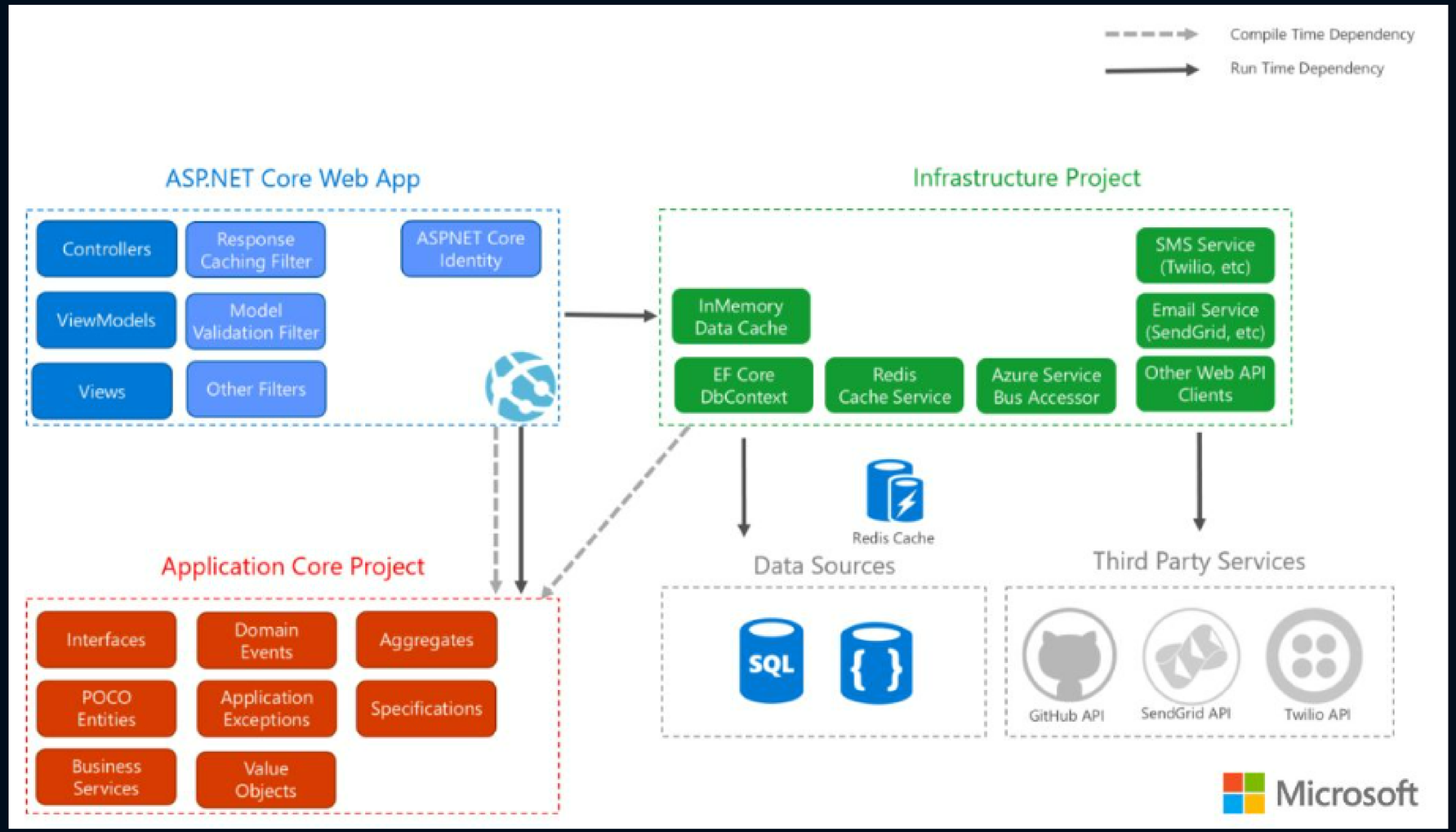
Bob’s Clean architecture

Bob’s Clean Architecture retains the core principles of Clean Architecture, including the separation of concerns and the independence of the core business logic from external dependencies. However, it introduces some modifications and refinements to make the architecture more pragmatic and easier to implement.

Use Cases: In Bob’s Clean Architecture, the central focus is on use cases, which represent the application’s behavior and business logic. Each use case corresponds to a specific action or operation that the application can perform.
Interactors: Use cases are implemented as interactors in Bob’s Clean Architecture. Interactors encapsulate the application’s business rules and logic for each use case. They represent the core application logic.
Interface Adapters: which are responsible for adapting external frameworks and interfaces to the use cases and entities. This includes the implementation of presenters, controllers, and gateways.
Frameworks and Drivers: consists of frameworks and drivers that interact with the external environment.

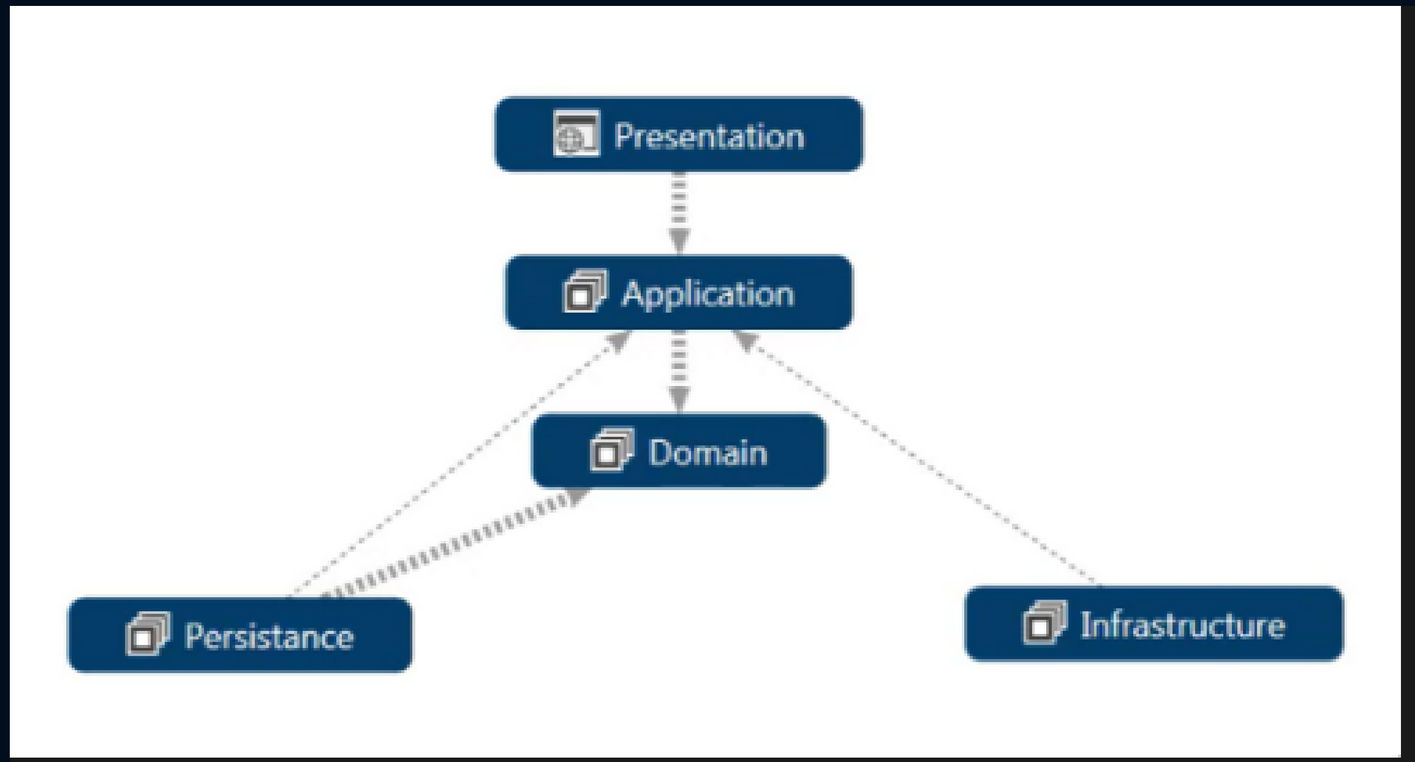


ASP NET Core Web App

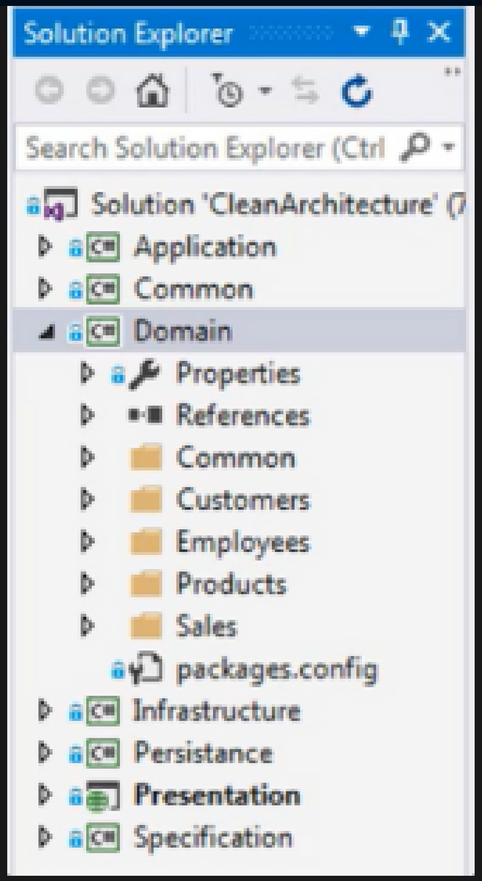


ASP.NET Core Diagram

Example



High level diagram



Scaffolding

Domain Centric architecture

Homework

Homework

Design (not code) an application that helps users to achieve mastery of a particular skill or set of skills over the course of a year.

1. Skill Selection: Users start by selecting a skill they want to master over the year. It could be anything from learning a new language, mastering a musical instrument, becoming proficient in a programming language, or developing a new athletic skill.
2. Goal Setting: Users set specific, measurable, and achievable goals related to their chosen skill. These goals could include milestones, projects, or levels of proficiency they aim to reach within the year.
3. Progress Tracking: The app allows users to track their progress over time. They can input achievements, completed lessons, or any relevant milestones. The app could visually represent their journey, providing a sense of accomplishment as they see how far they've come.
4. Yearly Summary: At the end of the year, the app generates a summary of the user's journey, showcasing their achievements and progress. This summary can be shared on social media or kept as a personal record.

The goal if this Task is for you to architect a solution for this given application, so you must use diagrams, images or pictures to show the architecture of the application and the Domain model