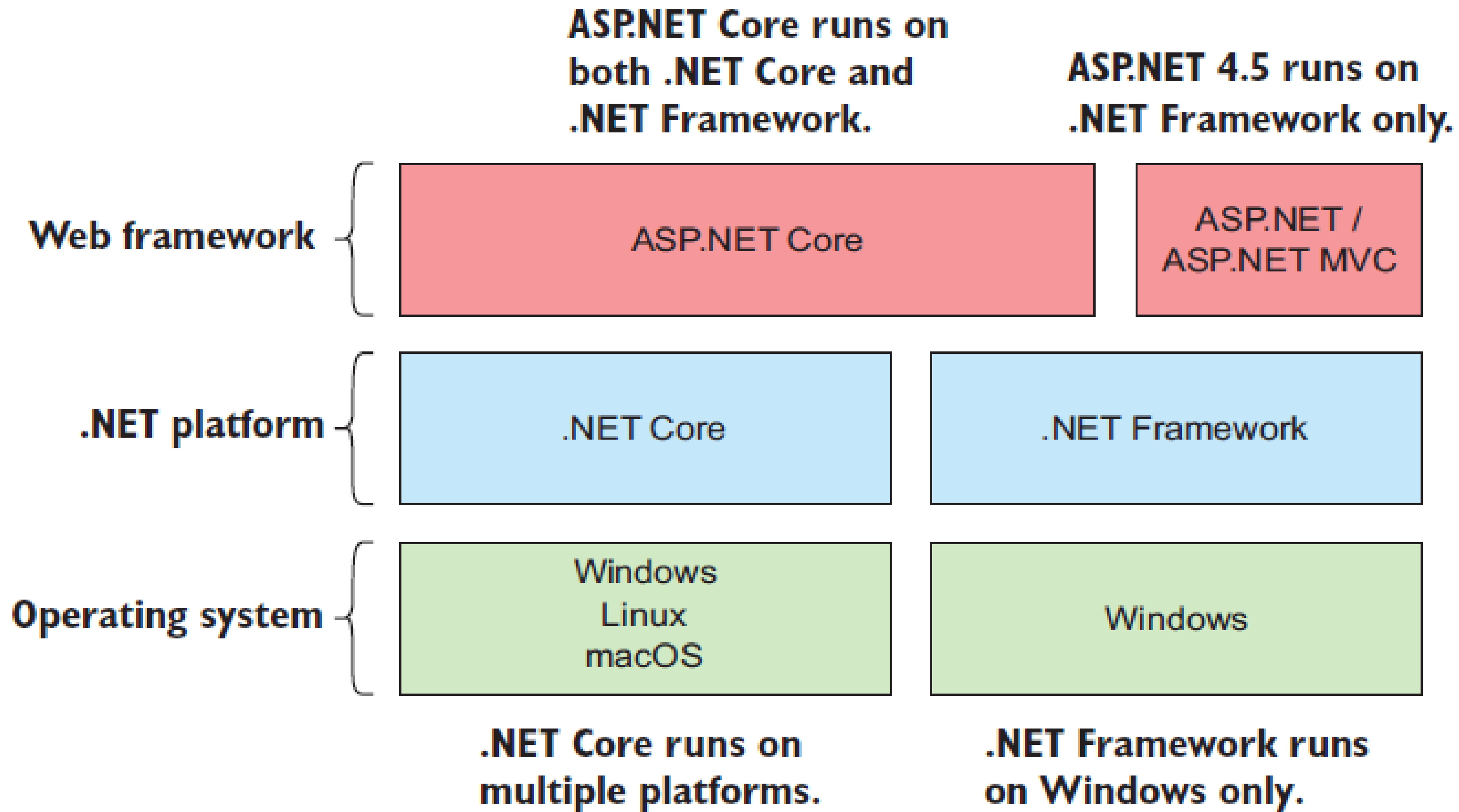
Unit 3

HTTP & ASP.NET Core

The relationship between ASP.NET Core, ASP.NET, .NET Core, and .NET Framework. ASP.NET Core runs on both .NET Framework and .NET Core, so it can run cross-platform.

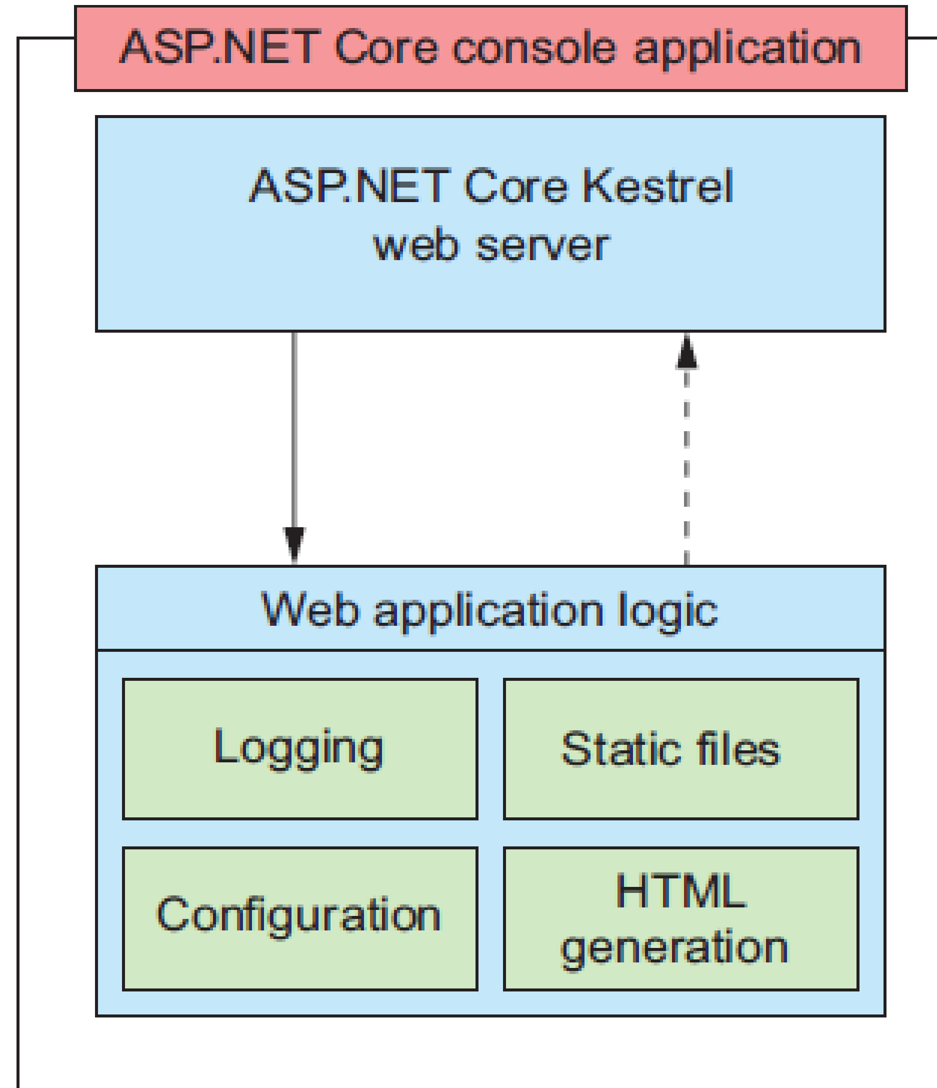


ASP.NET Core application model

You write a .NET Core console app that starts up an instance of an ASP.NET Core web server.

Microsoft provides, by default, a cross-platform web server called Kestrel.

Your web application logic is run by Kestrel. You'll use various libraries to enable features such as logging and HTML generation as required.



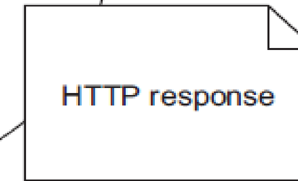
How does an HTTP web request work?

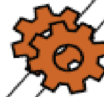
1. User requests a web page by a URL.

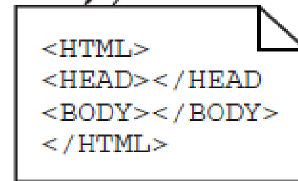
5. Browser renders HTML on page.

2. Browser sends HTTP request to server.

4. Server sends HTML in HTTP response back to browser.





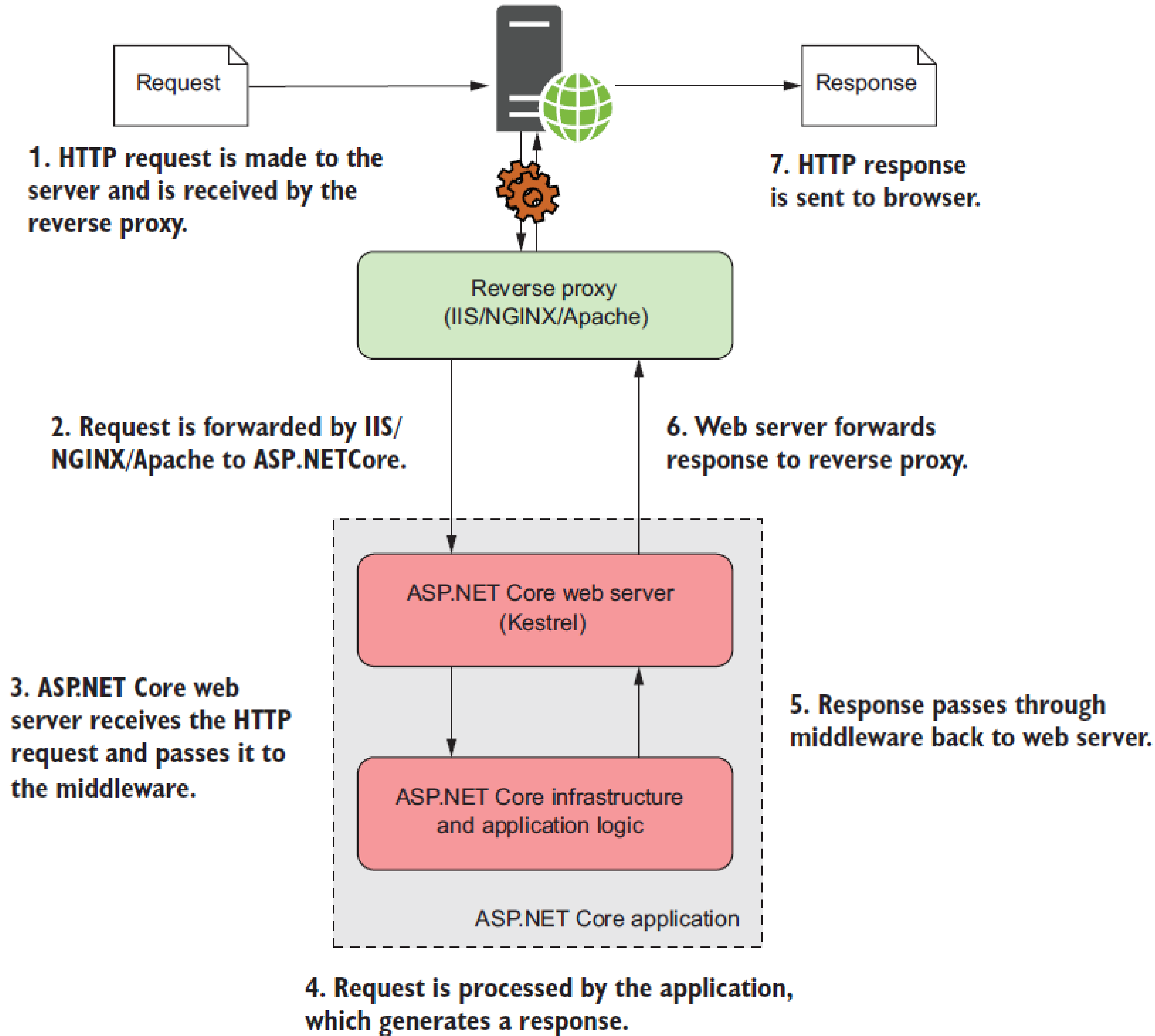


3. Server interprets request and generates appropriate HTML.

How does an HTTP web request work

- the user starts by requesting a web page, which causes an HTTP request to be sent to the server. The server interprets the request, generates the necessary HTML, and sends it back in an HTTP response. The browser can then display the web page.
- Once the server receives the request, it will check that it makes sense, and if it does, will generate an HTTP response. Depending on the request, this response could be a web page, an image, a JavaScript file, or a simple acknowledgment.
- As soon as the user's browser begins receiving the HTTP response, it can start displaying content on the screen, but the HTML page may also reference other pages and links on the server.

How does ASP.NET Core process a request?



How does ASP.NET Core process a request?

- A request is received from a browser at the reverse proxy, which passes the request to the ASP.NET Core application, which runs a self-hosted web server.
- The web server processes the request and passes it to the body of the application, which generates a response and returns it to the web server. The web server relays this to the reverse proxy, which sends the response to the browser.
- benefit of a reverse proxy is that it can be hardened against potential threats from the public internet. They're often responsible for additional aspects, such as restarting a process that has crashed. Kestrel can stay as a simple HTTP server. Think of it as a simple separation of concerns: Kestrel is concerned with generating HTTP responses; a reverse proxy is concerned with handling the connection to the internet.

Common web application architectures

- monolithic application
- All-in-one applications
- Layered Architecture
- Traditional "N-Layer" architecture applications
- Clean architecture

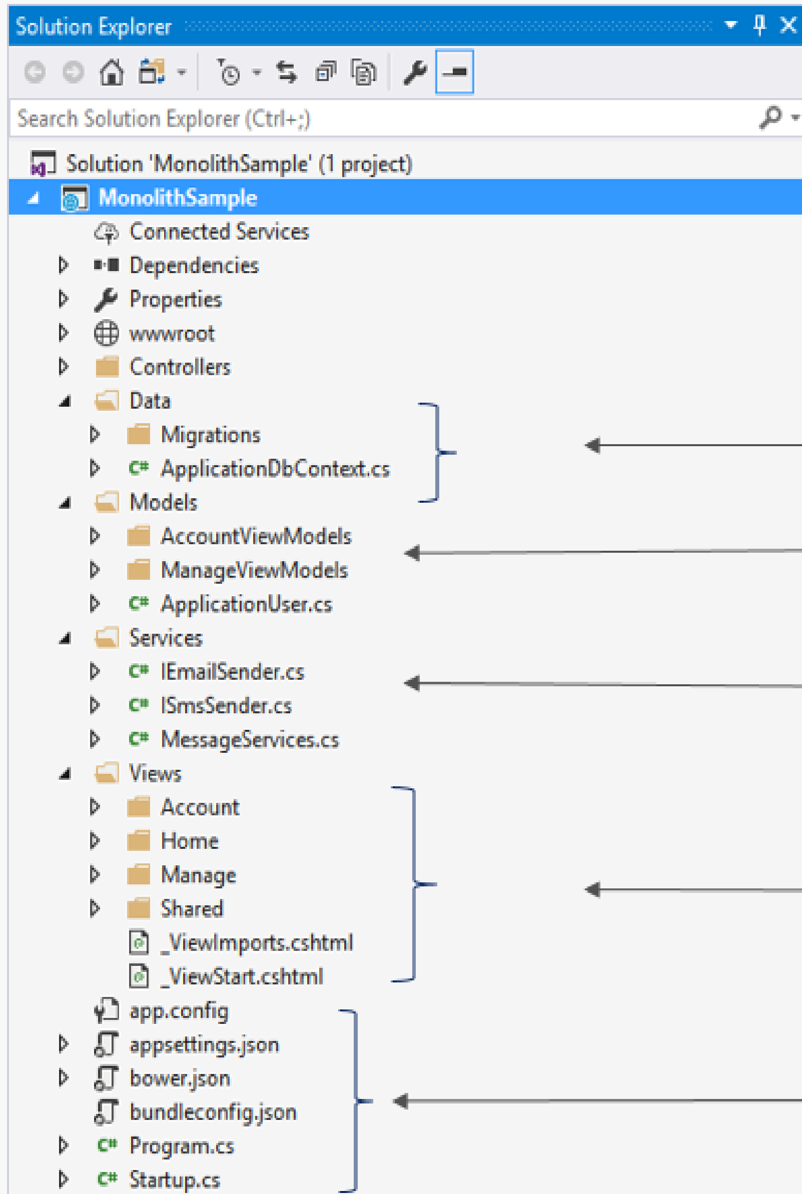
Monolithic Application

- A monolithic application is one that is entirely self-contained, in terms of its behavior.
- It may interact with other services or data stores in the course of performing its operations, but the core of its behavior runs within its own process and the entire application is typically deployed as a single unit.
- If such an application needs to scale horizontally, typically the entire application is duplicated across multiple servers or virtual machines.

All-in-one applications

- The smallest possible number of projects for an application architecture is one. In this architecture, the entire logic of the application is contained in a single project, compiled to a single assembly, and deployed as a single unit.
- A new ASP.NET Core project, whether created in Visual Studio or from the command line, starts out as a simple "all-in-one" monolith. It contains all of the behavior of the application, including presentation, business, and data access logic. In a single project scenario, separation of concerns is achieved through the use of folders. The default template includes separate folders for MVC pattern responsibilities of Models, Views, and Controllers, as well as additional folders for Data and Services. Figure shows the file structure of a single-project app.

VS Solution Structure



Data Access Logic
- EF Migrations
- EF DbContext and model design

UI Models

Application Services (interfaces and implementations)

Presentation Logic

Application Entry Point and Configuration

- Presentation details should be limited as much as possible to the Views folder, and data access implementation details should be limited to classes kept in the Data folder. Business logic should reside in services and classes within the Models folder.
- Although simple, the single-project monolithic solution has some disadvantages:
  - As the project's size and complexity grows, the number of files and folders will continue to grow as well. User interface (UI) reside in multiple folders, which aren't grouped together alphabetically.
  - Business logic is scattered between the Models and Services folders, and there's no clear indication of which classes in which folders should depend on which others. This lack of organization at the project level frequently leads to spaghetti code.
  - To address these issues, applications often evolve into multi-project solutions, where each project is considered to reside in a particular layer of the application.

Layered Architechture

- As applications grow in complexity, one way to manage that complexity is to break up the application according to its responsibilities or concerns. This follows the separation of concerns principle and can help keep a growing codebase organized so that developers can easily find where certain functionality is implemented.
- Layered architecture offers a number of advantages beyond just code organization, though. By organizing code into layers, common low-level functionality can be reused throughout the application.
- With a layered architecture, applications can enforce restrictions on which layers can communicate with other layers. This helps to achieve encapsulation. When a layer is changed or replaced, only those layers that work with it should be impacted. By limiting which layers depend on which other layers, the impact of changes can be mitigated so that a single change doesn't impact the entire application.

Traditional "N-Layer" architecture applications

Application Layers

User Interface

Business Logic

Data Access

Traditional "N-Layer" architecture applications

- These layers are frequently abbreviated as UI, BLL (Business Logic Layer), and DAL (Data Access Layer). Using this architecture, users make requests through the UI layer, which interacts only with the BLL. The BLL, in turn, can call the DAL for data access requests. The UI layer shouldn't make any requests to the DAL directly, nor should it interact with persistence directly through other means. Likewise, the BLL should only interact with persistence by going through the DAL.
- One disadvantage of this traditional layering approach is that compile-time dependencies run from the top to the bottom. That is, the UI layer depends on the BLL, which depends on the DAL. This means that the BLL, which usually holds the most important logic in the application, is dependent on data access implementation details (and often on the existence of a database). Testing business logic in such an architecture is often difficult, requiring a test database. The dependency inversion principle can be used to address this issue.

VS Solution Structure

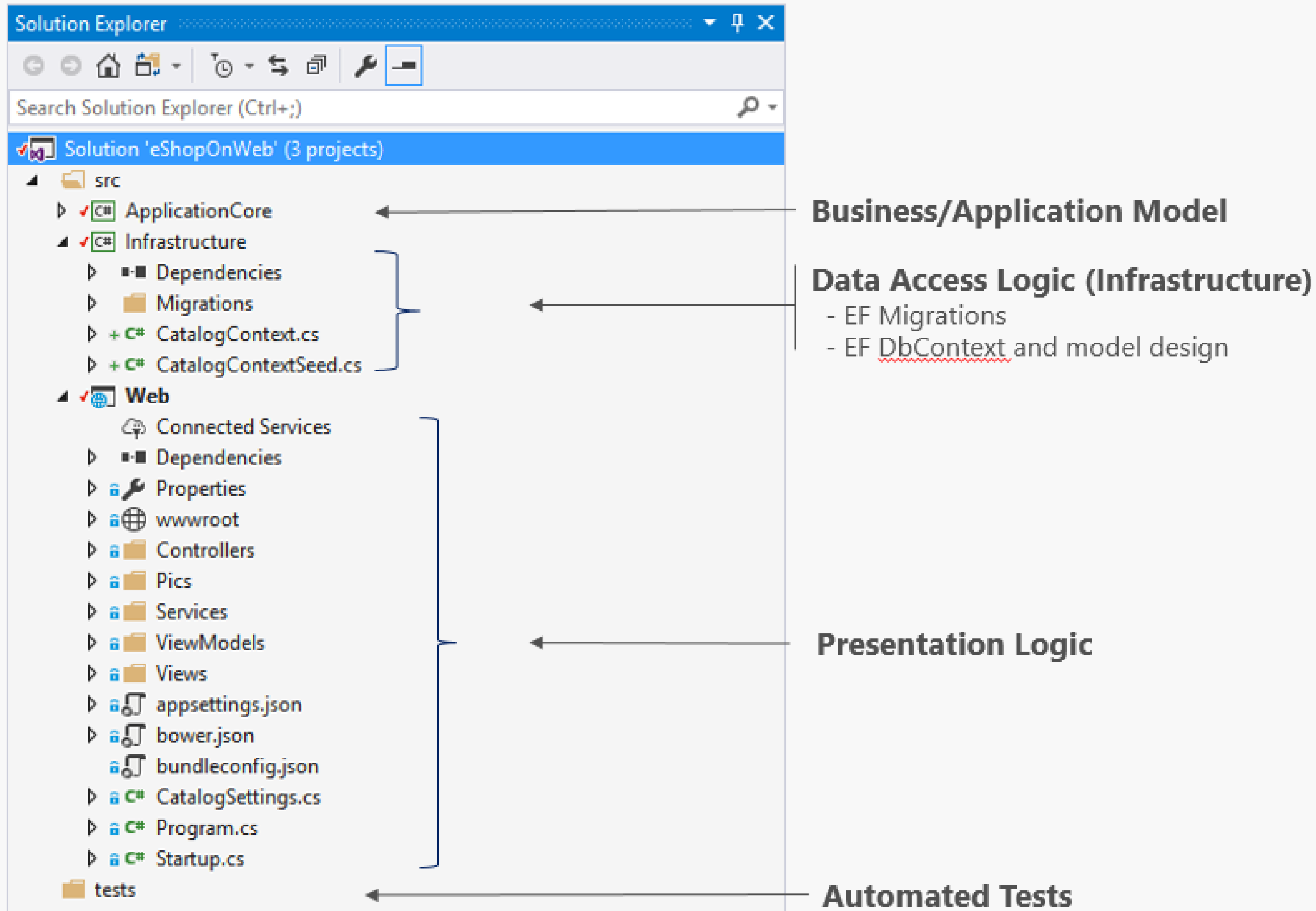


Figure shows an example solution, breaking the application into three projects by responsibility (or layer).

Clean architecture

- Applications that follow the Dependency Inversion Principle as well as the Domain-Driven Design (DDD) principles tend to arrive at a similar architecture. It's been cited as the Onion Architecture or Clean Architecture.
- Clean architecture puts the business logic and application model at the center of the application. Instead of having business logic depend on data access or other infrastructure concerns, this dependency is inverted: infrastructure and implementation details depend on the Application Core.
- This is achieved by defining abstractions, or interfaces, in the Application Core, which are then implemented by types defined in the Infrastructure layer. A common way of visualizing this architecture is to use a series of concentric circles, similar to an onion.

Clean Architecture Layers (Onion view)

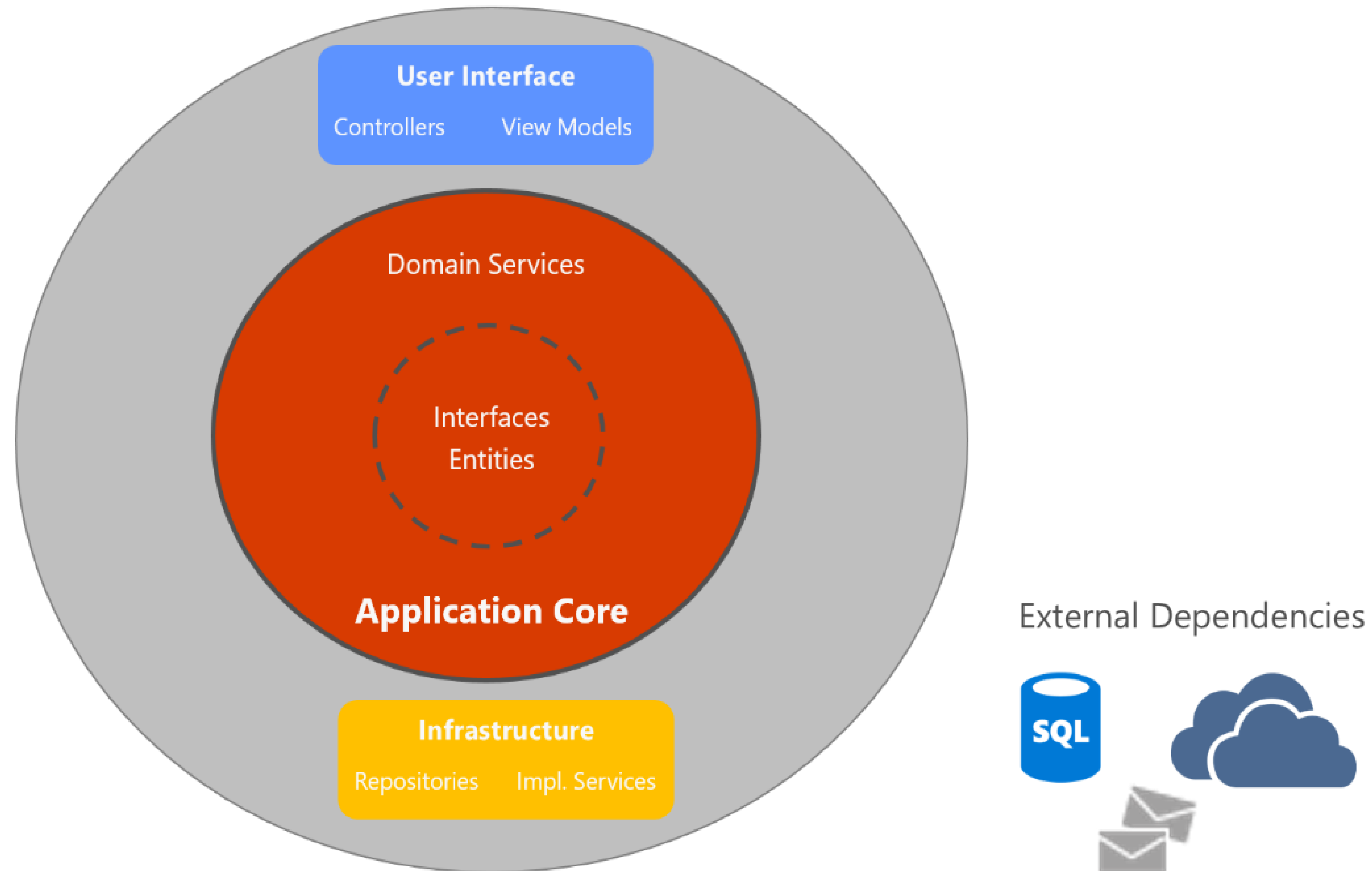


Figure: style of architectural representation.

Clean Architecture

- In the diagram, dependencies flow toward the innermost circle. The Application Core takes its name from its position at the core of this diagram. And you can see on the diagram that the Application Core has no dependencies on other application layers.
- The application's entities and interfaces are at the very center.
- Just outside, but still in the Application Core, are domain services, which typically implement interfaces defined in the inner circle.
- Outside of the Application Core, both the UI and the Infrastructure layers depend on the Application Core, but not on one another (necessarily).

Clean Architecture Layers

Optional Compile-Time Dependency
Compile-Time Dependency

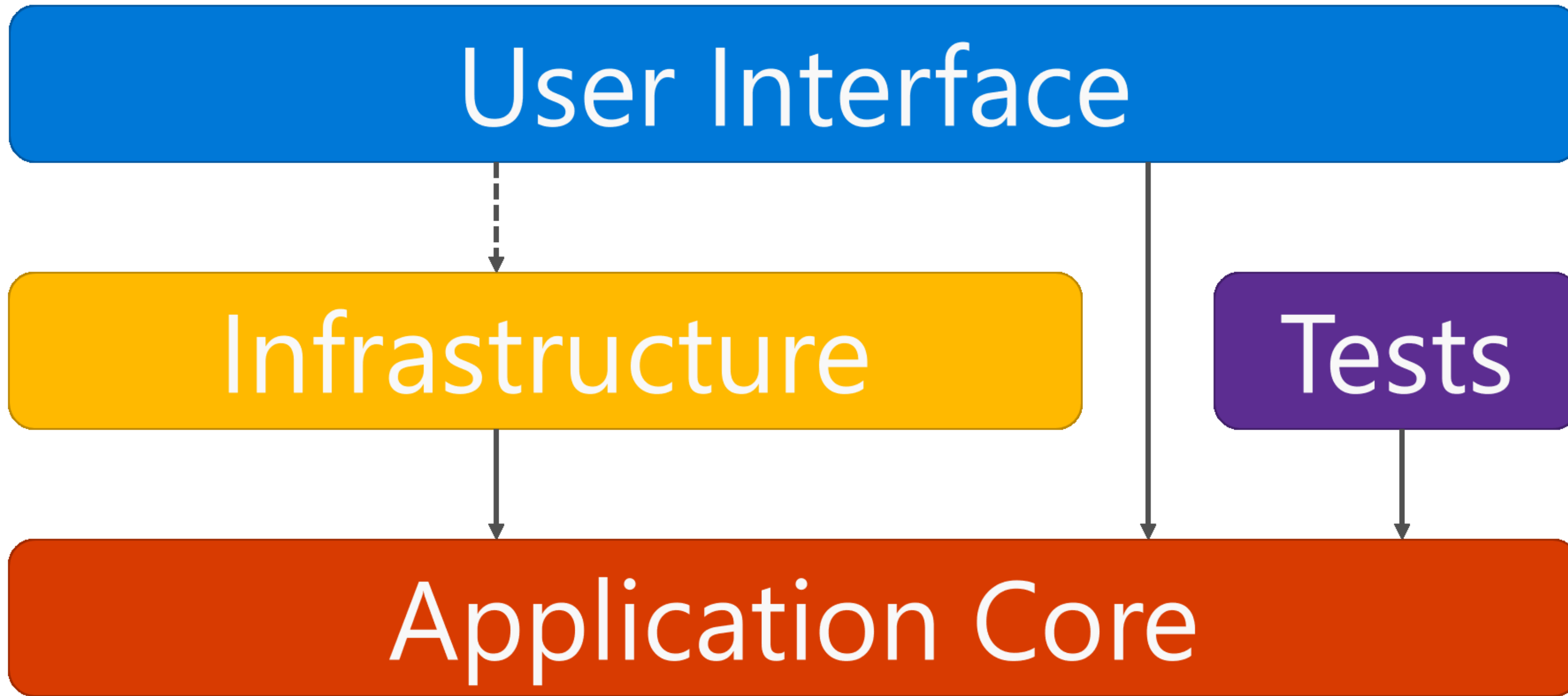


Figure shows a more traditional horizontal layer diagram that better reflects the dependency between the UI and other layers.

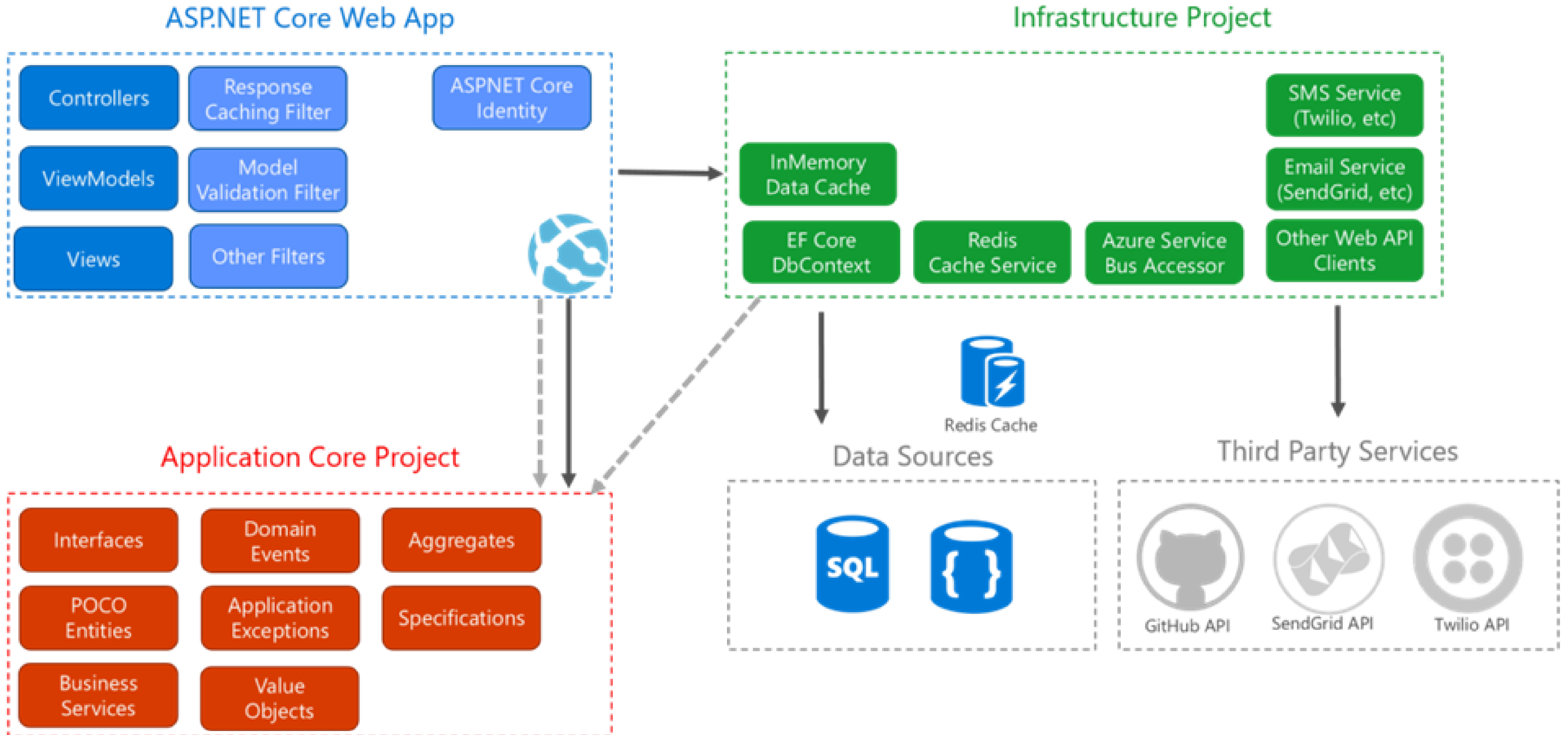
ASP.NET Core Architecture Overview

- The ideology behind ASP.NET Core in general, as the name suggests, is to lay out web logic, infrastructure, and core components from each other in order to provide a more development-friendly environment.
- The concept is somewhat similar to "N" tier/layer architecture, the only difference is that ASP.NET Core defines the layers as the core component of the platform which relieves the developer from redefining it in order to make a solution more modular and reusable.
- What happens in ASP.NET Core is that the main business logic and UI logic are encapsulated in ASP.NET Core Web App layer, while the database access layer, cache services, and web API services are encapsulated in infrastructure layer and common utilities, objects, interfaces and reusable business services are encapsulated as micro-services in application core layer.

ASP.NET Core Architecture Overview

- ASP.NET Core creates necessary pre-defined "N" tier/layers architecture for us developers automatically, which saves our time and effort to worry less about the complexity of necessary "N" tier/architecture of the web project and focus more on the business logic.
- ASP.NET Core that brings the benefit of a pre-built architectural framework that eases out tier deployment of the project along with providing pre-build Single Page Application (SPA) design pattern, Razor Pages (Page based more cleaner MVC model) design pattern, and traditional MVC (View based model) design pattern.
- These design patterns are mostly used in a hybrid manner but can be utilized as an individual-only pattern as well.

ASP.NET Core Architecture



MVC(Model – View - Controller) Design Pattern

- The MVC design has actually been around for a few decades, and it's been used across many different technologies.
- The MVC design pattern is a popular design pattern for the user interface layer of a software application.
- In larger applications, you typically combine a model-view-controller UI layer with other design patterns in the application, like data access patterns and messaging patterns.
- These will all go together to build the full application stack.

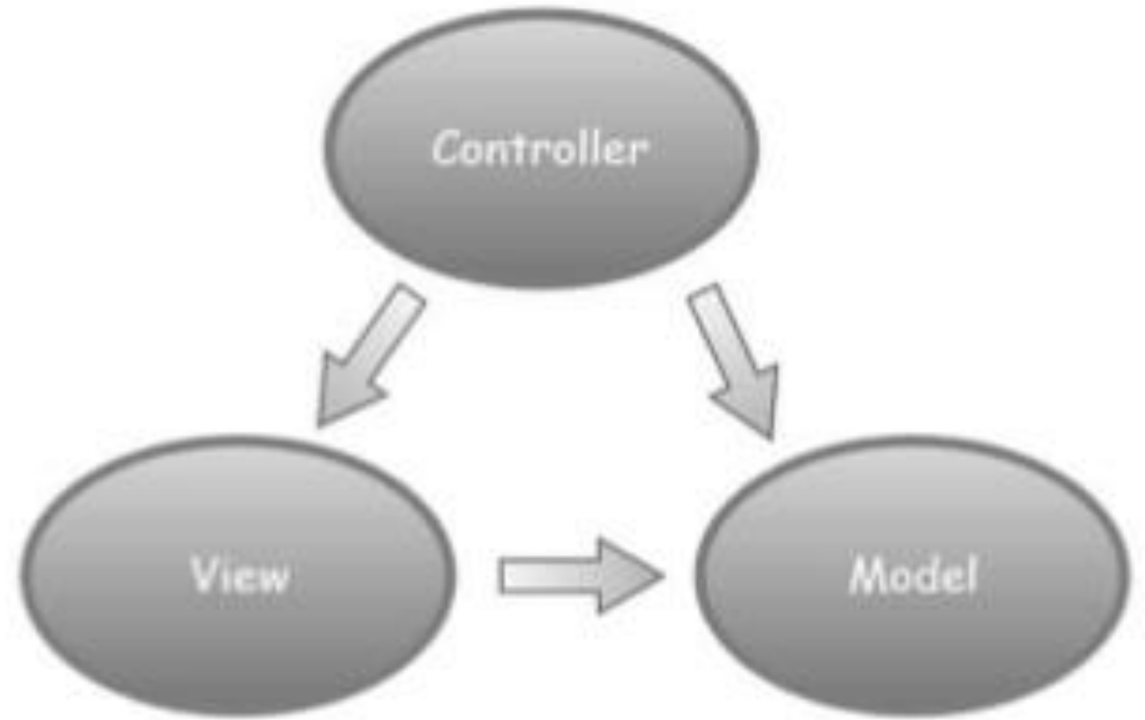
MVC(Model – View - Controller) Design Pattern

- The MVC separates the user interface (UI) of an application into the following three parts −
- **The Model** − A set of classes that describes the data you are working with as well as the business logic.
- **The View** − Defines how the application's UI will be displayed. It is a pure HTML which decides how the UI is going to look like.
- **The Controller** − A set of classes that handles communication from the user, overall application flow, and application-specific logic.

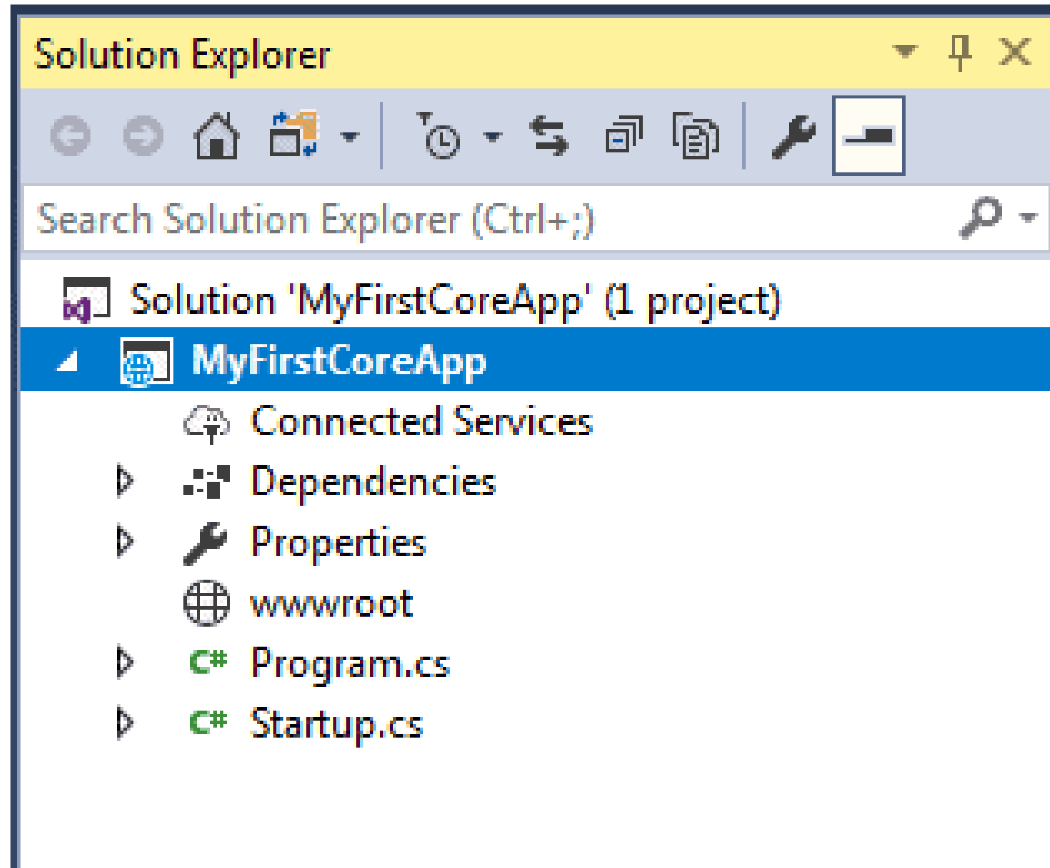
Idea Behind MVC

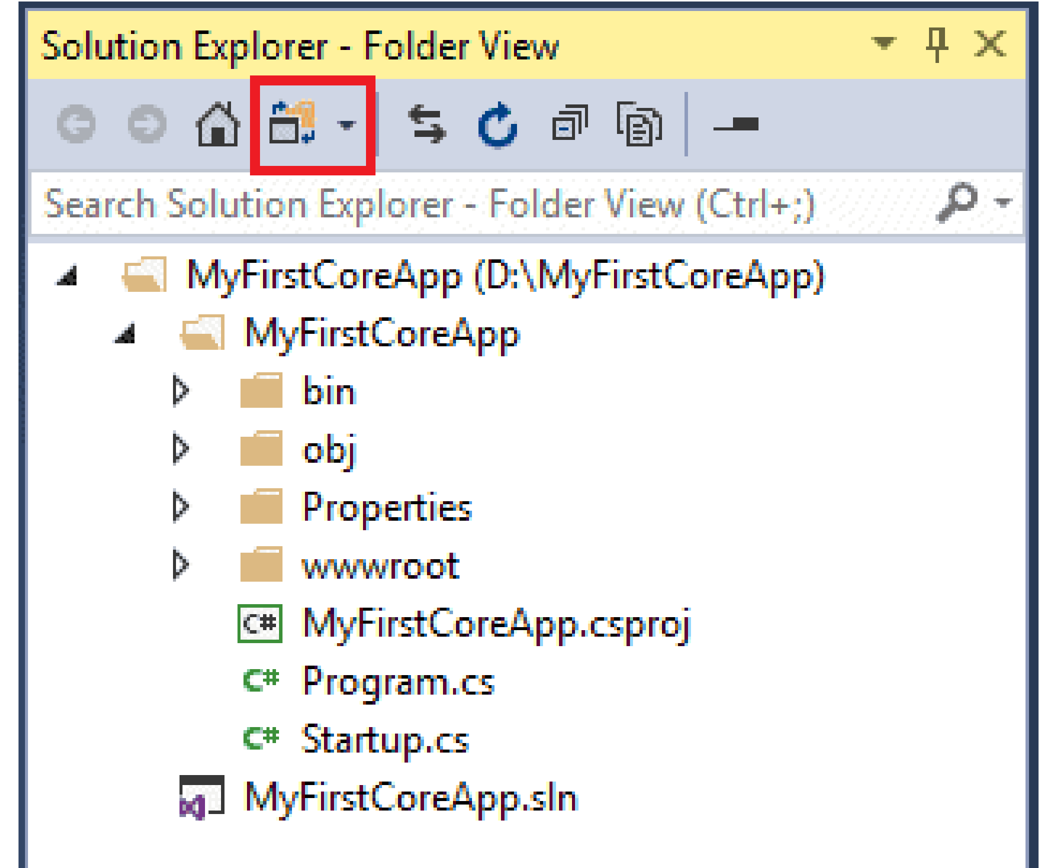
- The idea is that you'll have a component called the view which is solely responsible for rendering this user interface whether it should be HTML or whether it actually should be a UI widget on a desktop application.
- The view talks to a model, and that model contains all the data that the view needs to display.
- In a web application, the view might not have any code associated with it at all.
- It might just have HTML and then some expressions of where to take the pieces of data from the model and plug them into the correct places inside the HTML template that you've built in the view.
- The controller organizes everything. When an HTTP request arrives for an MVC application, the request gets routed to a controller, and then it's up to the controller to talk to either the database, the file system, or a model.

Idea Behind MVC



Projects and Conventions





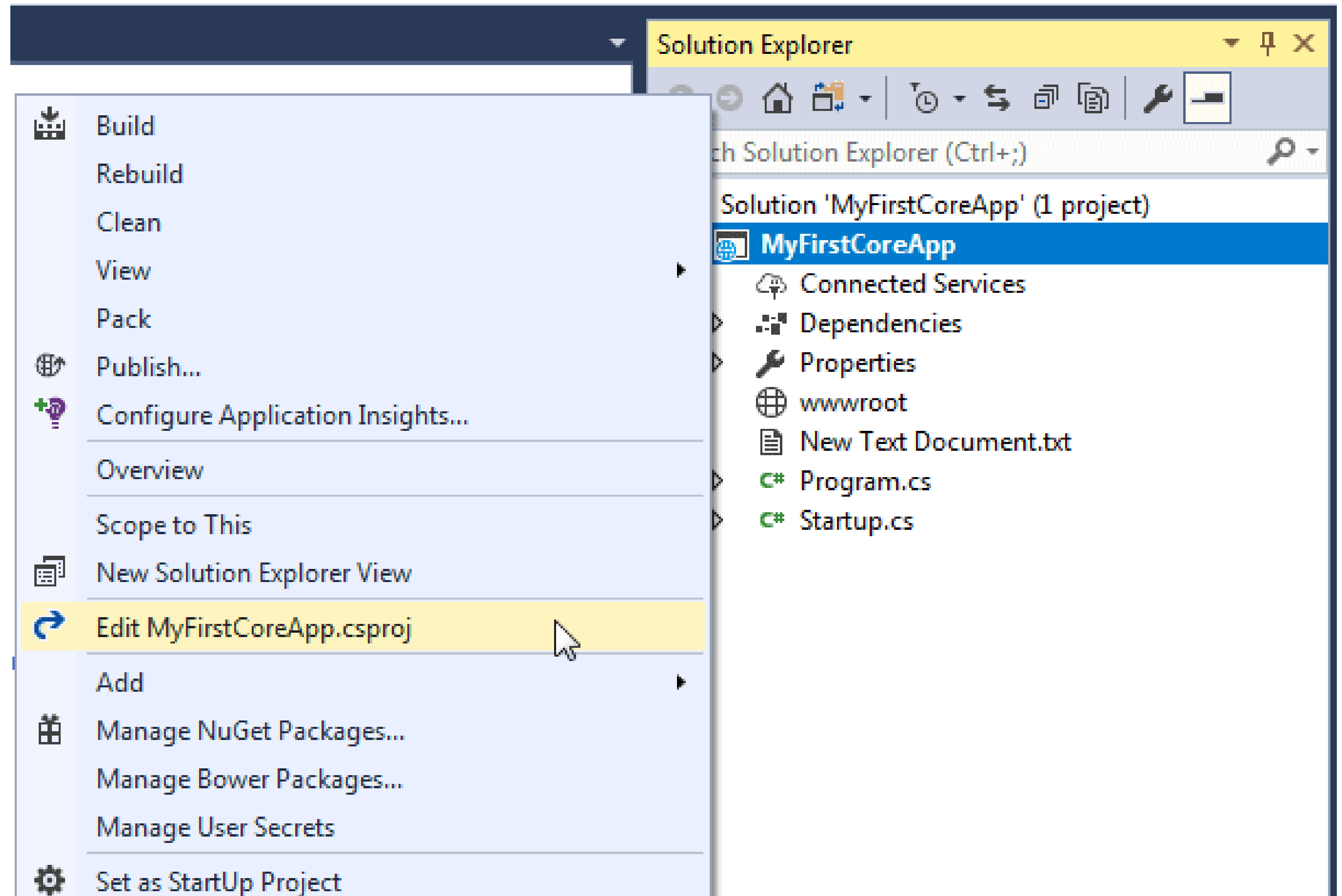
Projects and Conventions

- **.csproj –**

Visual Studio now uses .csproj file to manage projects.

We can edit the .csproj settings by :

- right click on the project
- Select Edit < project-name>.csproj as shown below.



Projects and Conventions

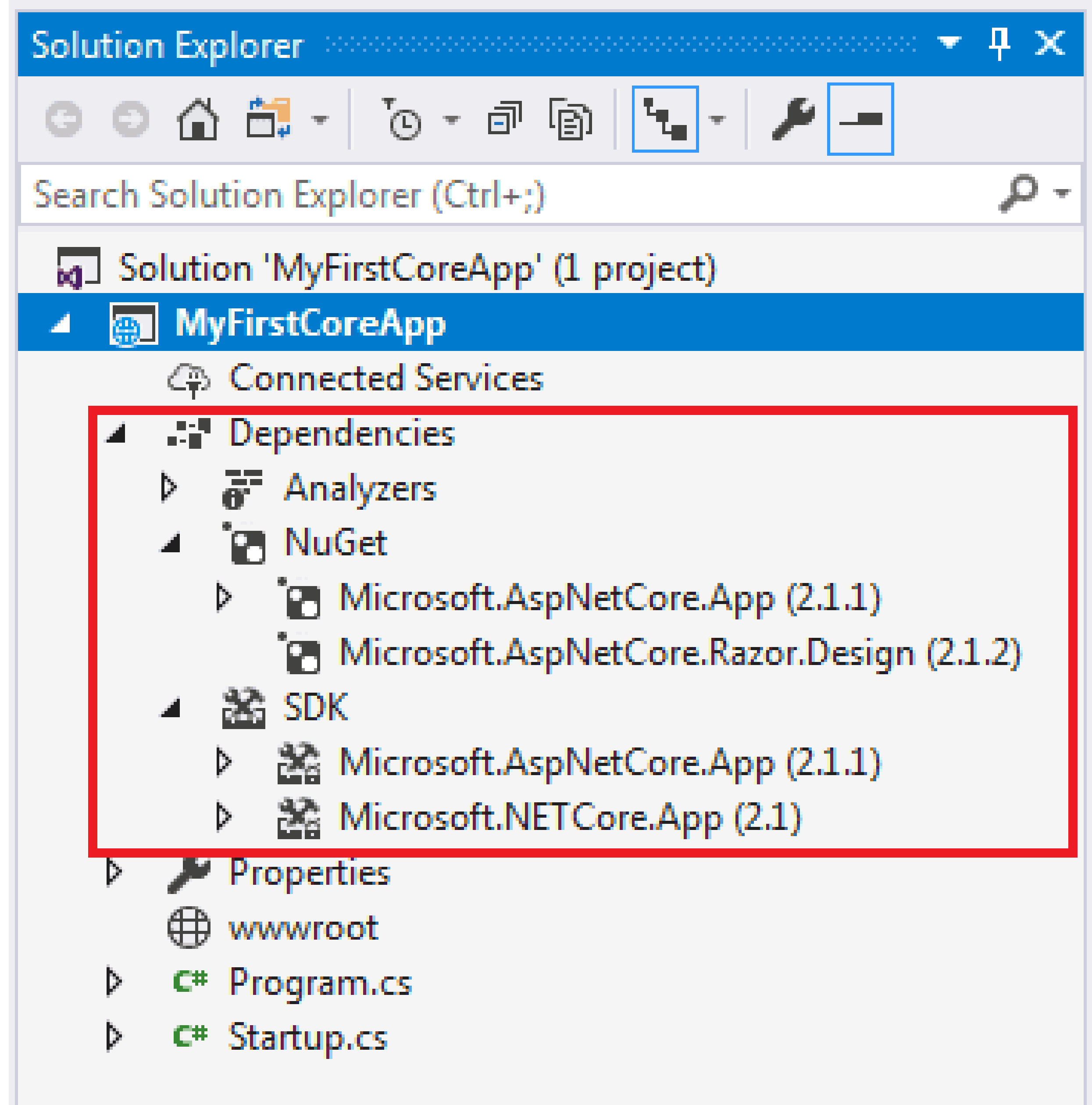
```
1  <Project Sdk="Microsoft.NET.Sdk.Web">
2
3    <PropertyGroup>
4      <TargetFramework>netcoreapp2.1</TargetFramework>
5    </PropertyGroup>
6
7    <ItemGroup>
8      <Folder Include="wwwroot\" />
9    </ItemGroup>
10
11   <ItemGroup>
12     <PackageReference Include="Microsoft.AspNetCore.App" />
13     <PackageReference Include="Microsoft.AspNetCore.Razor.Design" Version="2.1.2" PrivateAssets="All" />
14   </ItemGroup>
15
16 </Project>
17
```

- The .csproj for the project looks like above.
- The csproj file includes settings related to targeted .NET Frameworks, project folders, NuGet package references etc.

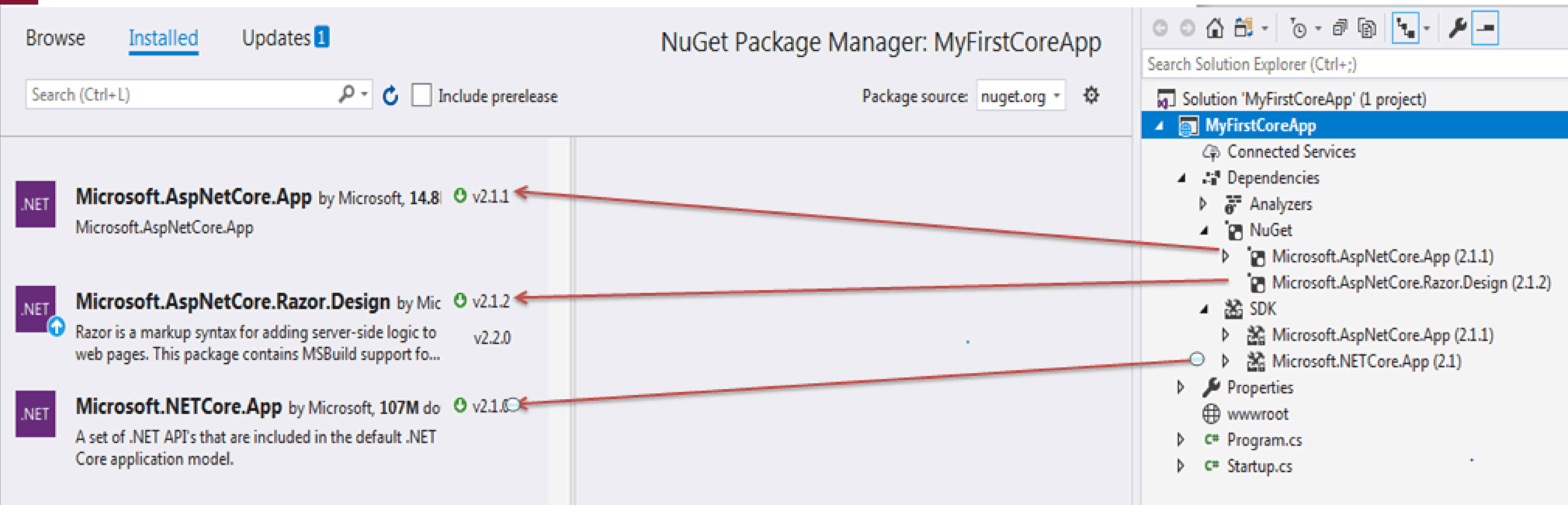
Projects and Conventions

- **Dependencies**

The Dependencies in the ASP.NET Core project contain all the installed server-side NuGet packages, as shown.



- Right click on "Dependencies" and then click "**Manage NuGet Packages..**" to see the installed NuGet packages, as shown below.
- It has installed three packages, Microsoft.AspNetCore.App package is for ASP.NET web application, Microsoft.AspNetCore.Razor.Design package is for Razor engine, and Microsoft.NETCore.App package is for .NET Core API.



Properties : The Properties node includes launchSettings.json file which includes Visual Studio profiles of debug settings. The following is a default launchSettings.json file.