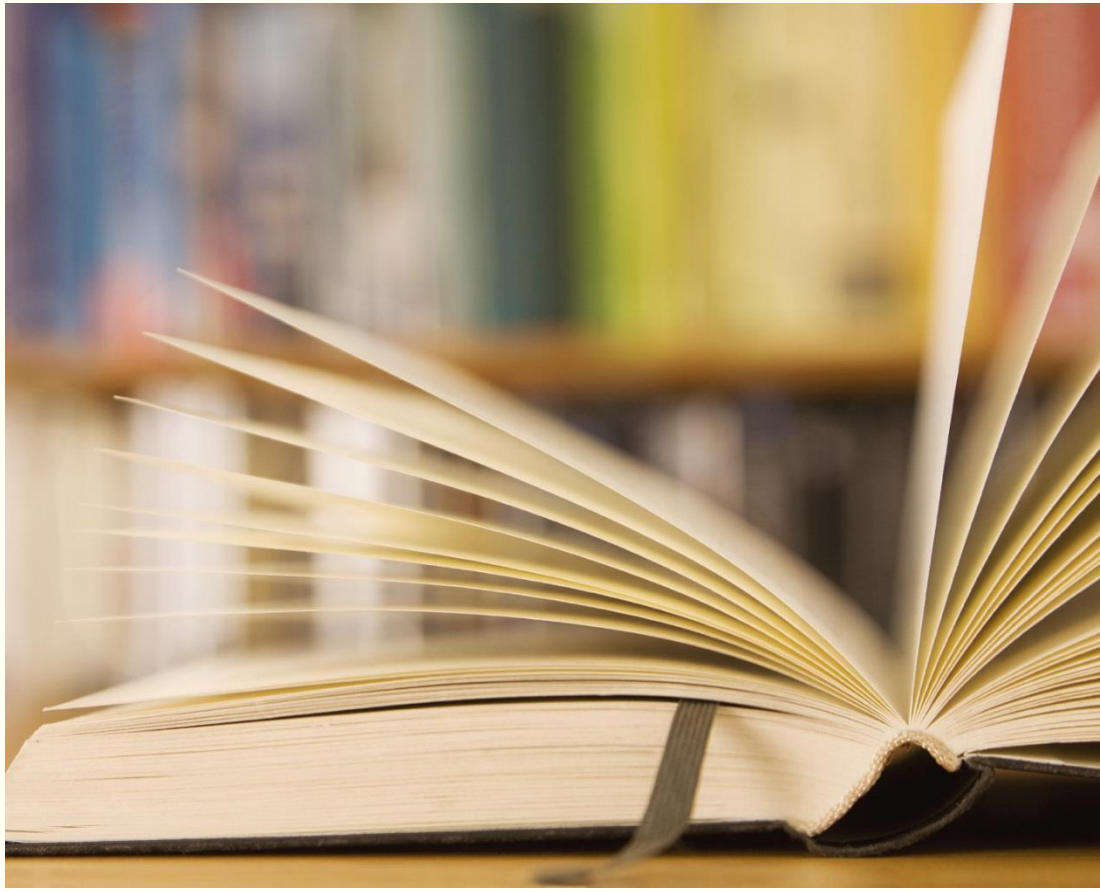
ОГЛЯД ТЕХНОЛОГІЇ ASP.NET CORE

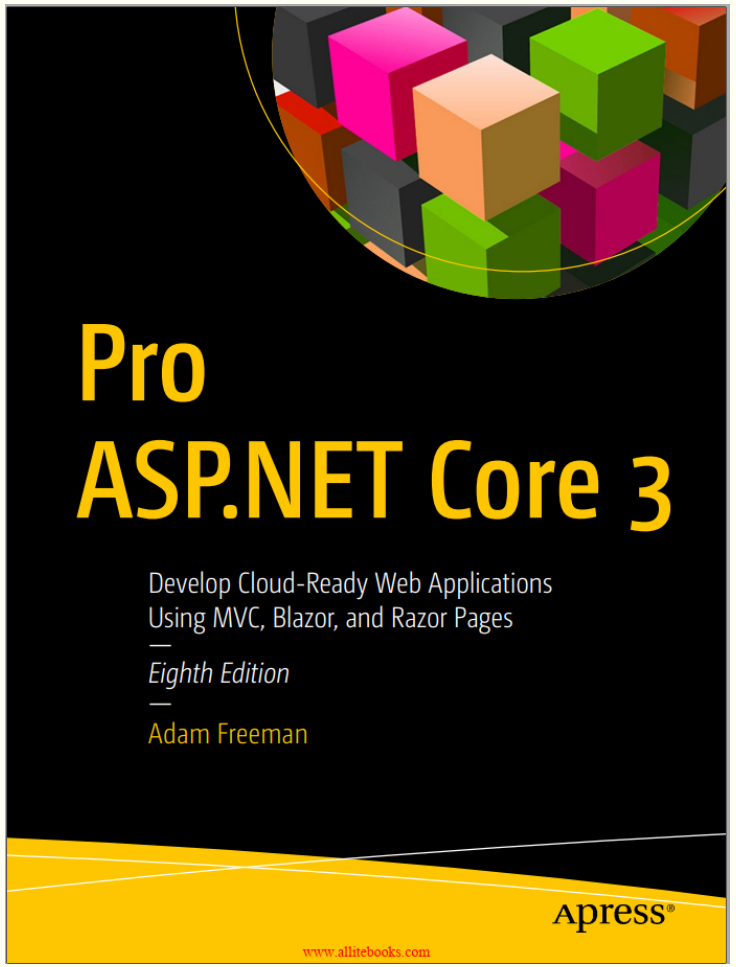
Лекція 05
Інструментальні засоби візуального програмування



План лекції

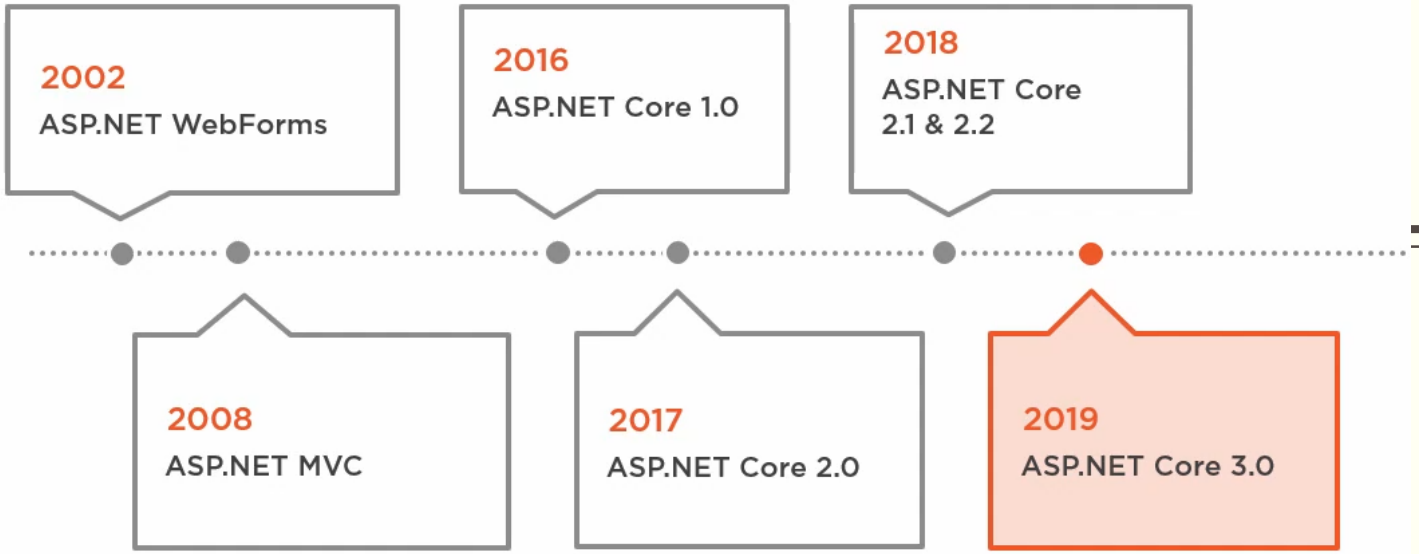
- **Огляд архітектури ASP.NET Core**
- **Створення першого веб-додатку**
- **Основні можливості мови C# для веб-розробки**
- **Тестування додатків ASP.NET Core**

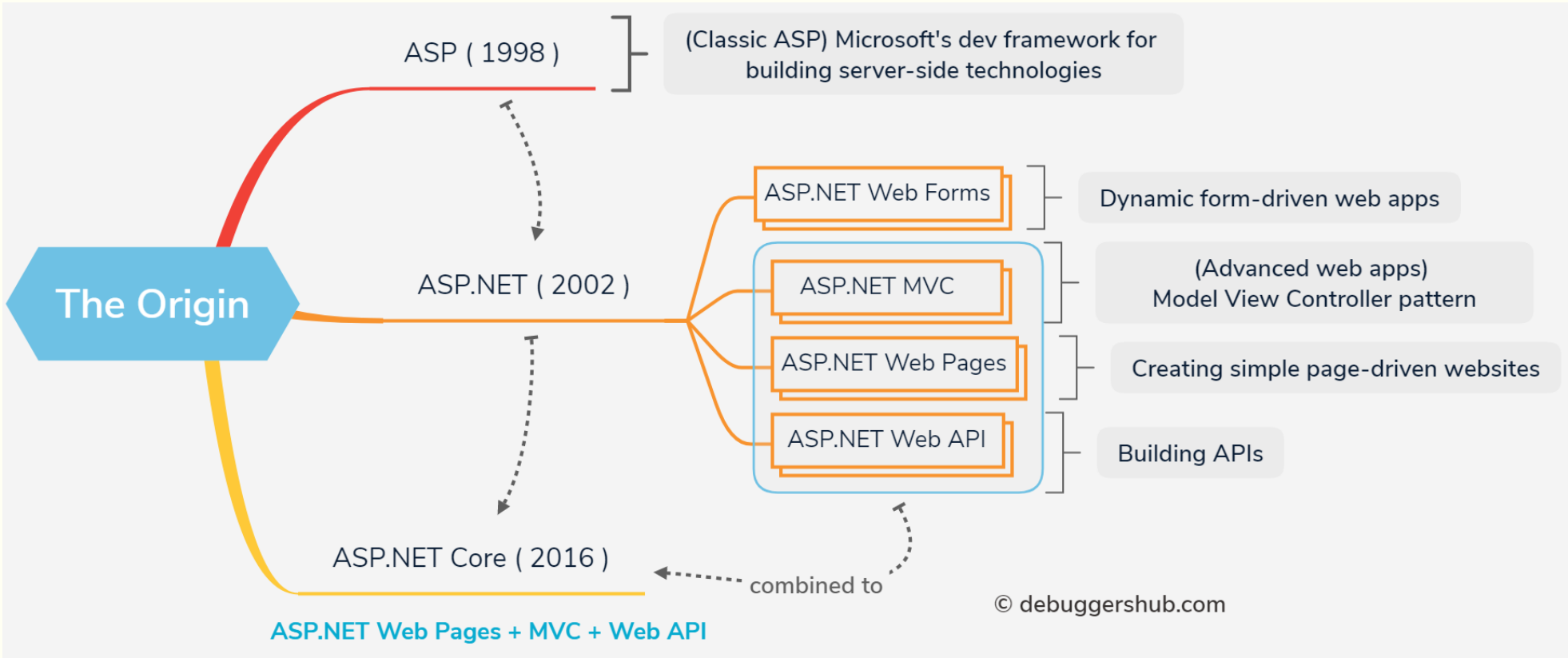
Література


Pro
ASP.NET Core 3
Develop Cloud-Ready Web Applications
Using MVC, Blazor, and Razor Pages
Eighth Edition
Adam Freeman
Apress
www.allitebooks.com

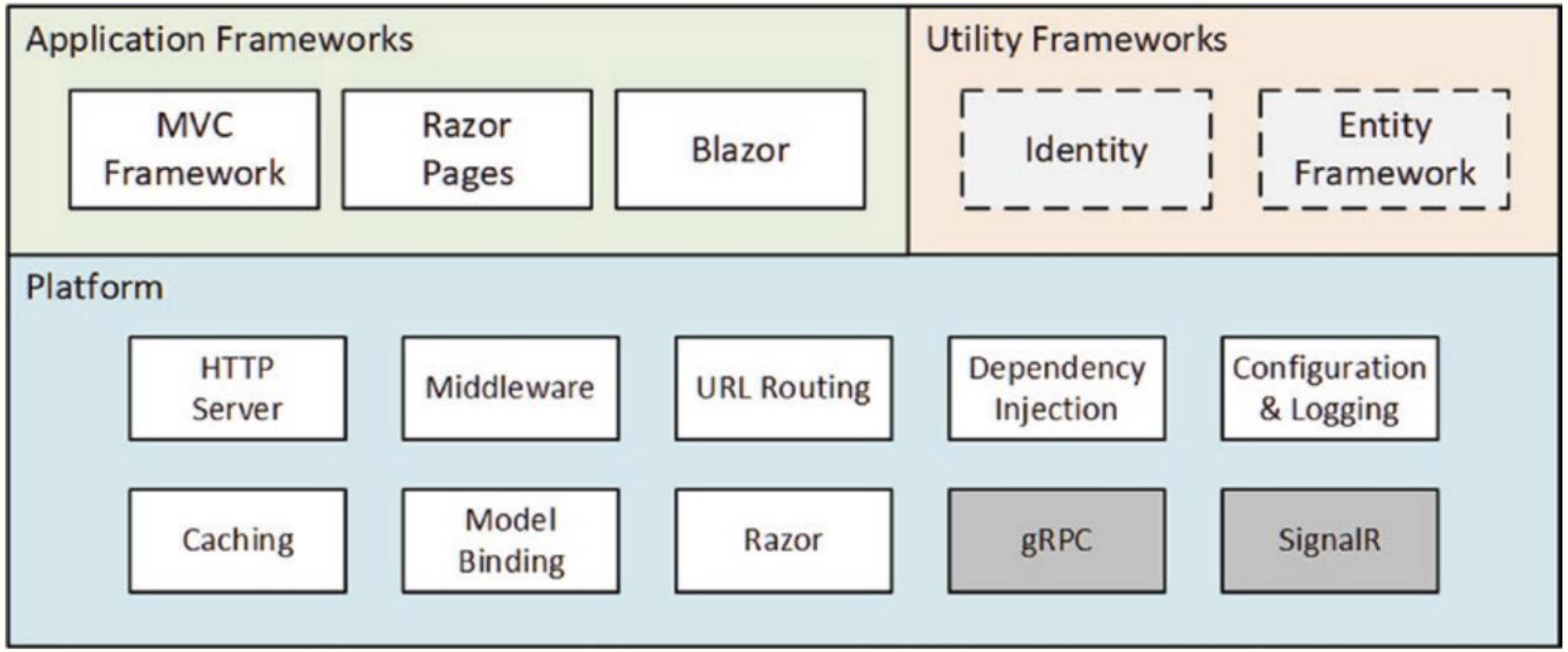
ОГЛЯД АРХІТЕКТУРИ ASP.NET CORE

Питання 5.1.


2002
ASP.NET WebForms
2016
ASP.NET Core 1.0
2018
ASP.NET Core
2.1 & 2.2
2008
ASP.NET MVC
2017
ASP.NET Core 2.0
2019
ASP.NET Core 3.0

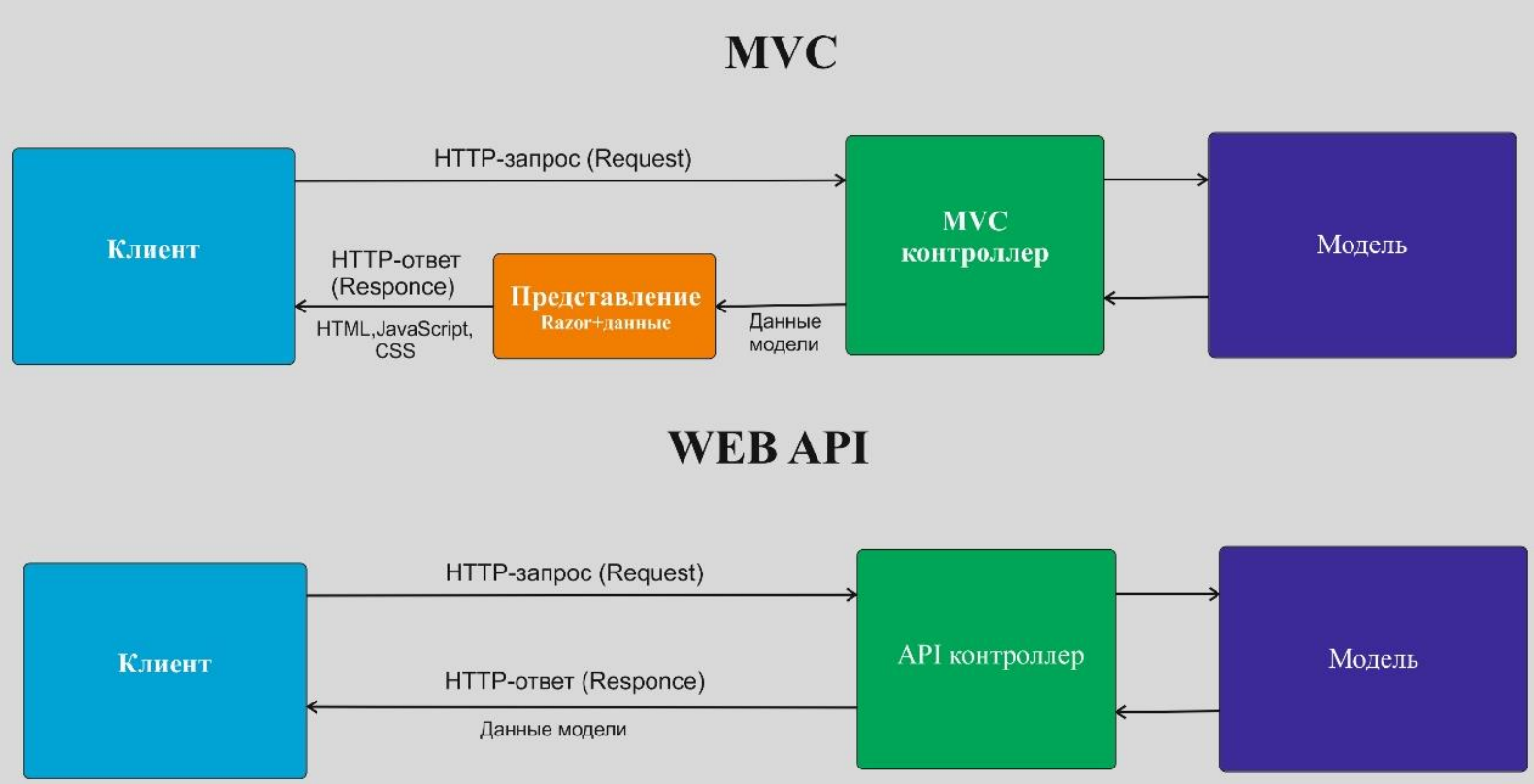

The Origin
ASP (1998)
(Classic ASP) Microsoft's dev framework for building server-side technologies
ASP.NET (2002)
ASP.NET Web Forms
Dynamic form-driven web apps
ASP.NET MVC
(Advanced web apps)
Model View Controller pattern
ASP.NET Web Pages
Creating simple page-driven websites
ASP.NET Web API
Building APIs
ASP.NET Core (2016)
combined to
ASP.NET Web Pages + MVC + Web API
© debuggershub.com

Архітектура ASP.NET Core



- ASP.NET Core складається з
  - платформи для обробки HTTP-запитів,
  - набору важливих фреймворків для створення додатків
  - вторинних utility-фреймворків, що постачають додаткову підтримуючу функціональність

Огляд MVC Framework

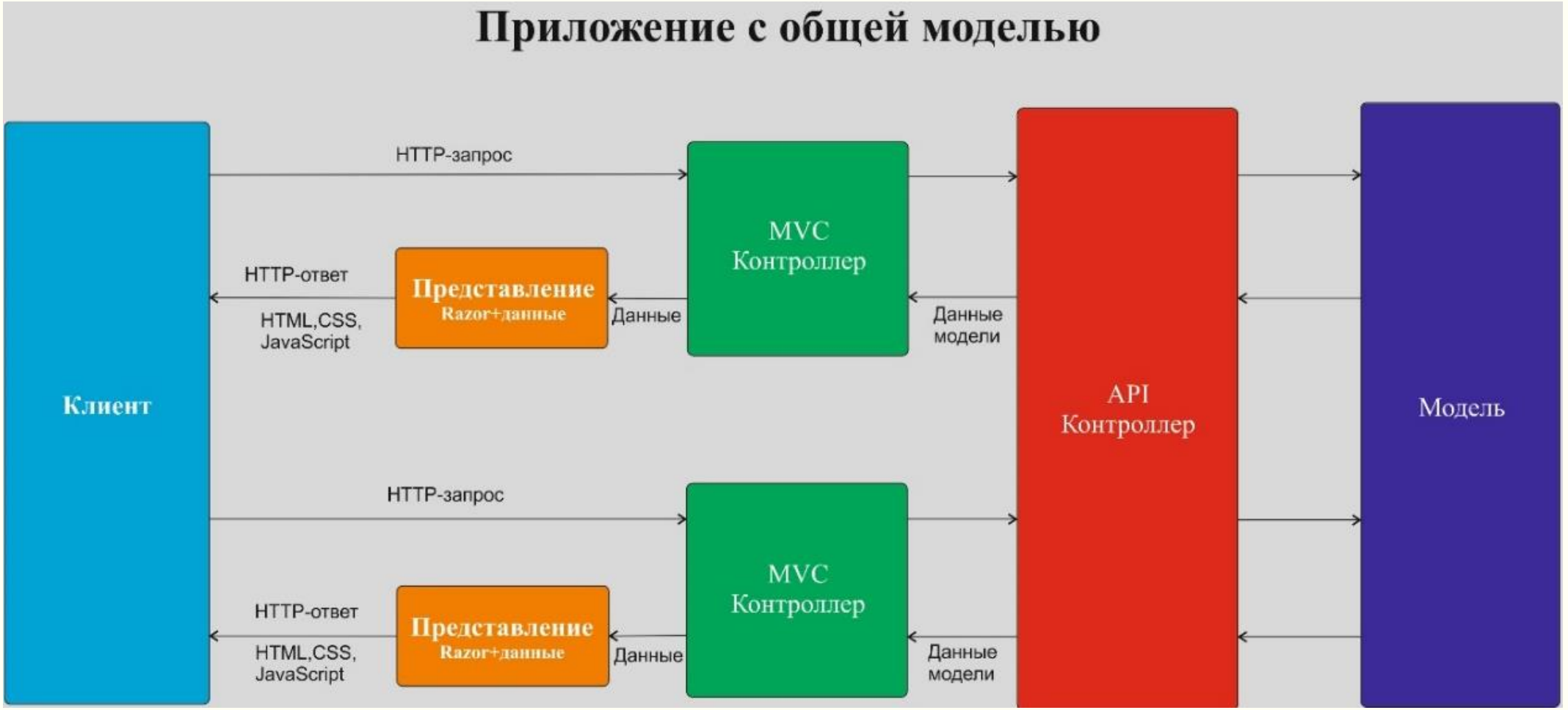


Був представлений задовго до появи ASP.NET Core.

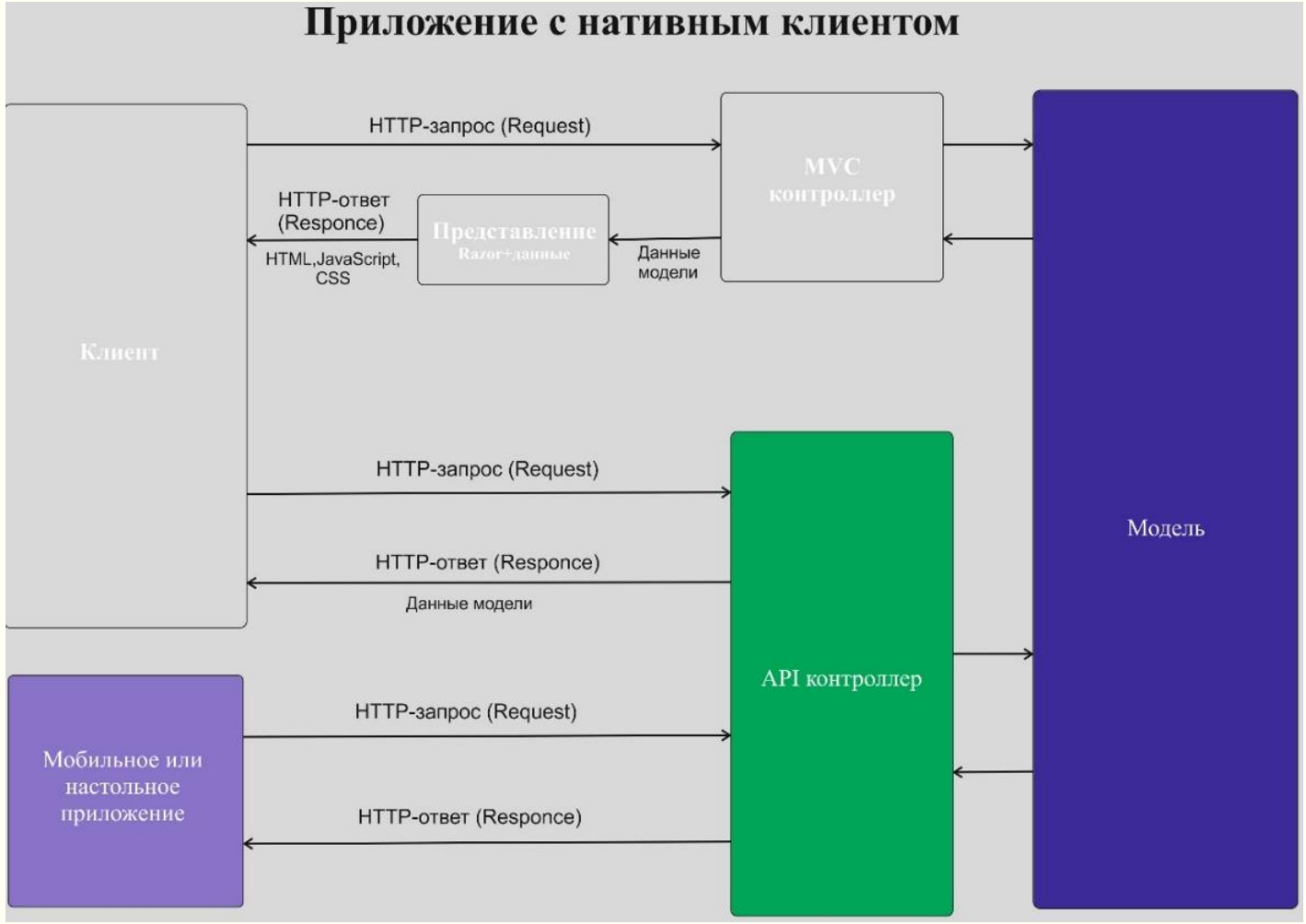
- Початковий ASP.NET базувався на моделі розробки Web Pages, яка відтворювала досвід написання настільних додатків, проте створювала громіздкі та погано масштабовані веб-додатки.
- MVC Framework було представлено разом з Web Pages, щоб надати модель розробки, яка включає характер взаємодії з HTTP та HTML.
- MVC жорсткіше орієнтований на повернення HTML, а Web Api – даних.
- MVC повертає клієнту представлення (комбінацію розмітки Razor HTML, JavaScript, CSS).
- Веб API повертає тільки серіалізовані дані.

Типи додатків, у яких MVC та Web Api взаємодіють

Типи додатків, у яких MVC та Web Api взаємодіють



Типи додатків, у яких MVC та Web Api взаємодіють



- 4) Сервісні веб-додатки (веб-служби)

Техоногія Razor Pages

- Один з недоліків MVC Framework – він може вимагати багато підготовчої роботи перед тим, як додаток зможе розпочати виробництво контенту.
  - Незважаючи на структурні проблеми, перевагою Web Pages була простота розробки додатків.
- Razor Pages повторює підходи до розробки з Web Pages та реалізує їх з використанням можливостей, початково розроблених для MVC Framework.
  - Код і контент змішуються для формування self-contained pages; це відтворює швидкість розробки в технології Web Pages без деяких технічних проблем, що лежали в її основі.
  - Проте проблема масштабування (scaling up) складних проєктів може залишатись.

Технологія Blazor

- Поширення клієнтських фреймворків на основі JavaScript може бути бар'єром для розробників мовою C#.
  - Технологія Blazor намагається обійти дану проблему, дозволяючи використовувати C# для створення клієнтської сторони веб-додатків.
  - Існує 2 версії Blazor: Blazor Server та Blazor WebAssembly.
  - Blazor Server – стабільна та підтримується в ASP.NET Core, працює з використанням постійного (persistent) HTTP-з'єднання з ASP.NET Core сервером, на якому виконується C#-код додатку.
  - Blazor WebAssembly – експериментальний реліз, який дозволяє виконувати C#-код додатку в браузері.
  - Жодна з версій Blazor не доречна для всіх ситуацій, проте разом вони дають напрямок розвитку ASP.NET Core розробки в майбутньому.

Технологія Utility Frameworks

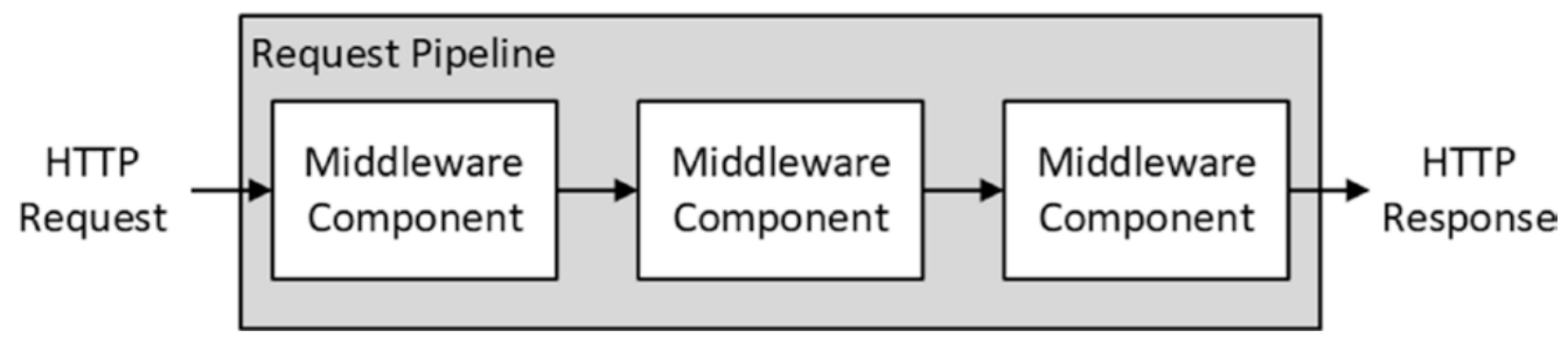
- Два тісно пов’язаних з ASP.NET Core фреймворки, що не використовуються напряму для генерування HTML-контенту або даних.
  - Entity Framework Core – це object-relational mapping (ORM) фреймворк компанії Microsoft, який представляє дані, збережені в реляційній БД, як .NET-об’єкти.
  - Entity Framework Core може застосовуватись у будь-якому .NET Core додатку та широко використовується для доступу до баз даних в ASP.NET Core додатках.
  - ASP.NET Core Identity – це фреймворк Microsoft для аутентифікації та авторизації; він використовується для перевірки user credentials в ASP.NET Core додатках та обмеження доступу до можливостей додатку.

Розуміння платформи ASP.NET Core

- Платформа містить низькорівневі фічі, необхідні для отримання та обробки HTTP-запитів і створення відгуку.
  - Присутній вбудований HTTP-сервер, система проміжних компонентів для обробки запитів та core features, від яких залежать фреймворки додатку, зокрема URL маршрутизація та двигун представлень Razor.
  - Ефективне використання ASP.NET Core вимагає розуміння потужних можливостей, які постачає платформа.
- Ми не розглядатимемо дві важливих можливості платформи: SignalR та gRPC.
  - SignalR використовується для створення низьколатентних каналів зв'язку між додатками, забезпечує основу для фреймворка Blazor Server.
  - SignalR нечасто застосовується напряму, існують кращі альтернативи для незначної кількості проєктів з потребою в низьколатентном обміні повідомлень: Azure Event Grid, Azure Service Bus тощо.
  - gRPC – новий ефективний та масштабований стандарт для кросплатформного віддаленого виклику процедур (RPC) по HTTP, спочатку створений компанією Google.
  - gRPC може стати майбутнім стандартом для web-служб, проте він не може використовуватись у веб-додатках, оскільки вимагає низькорівневого управління HTTP-повідомленнями, що ним надсилаються.
  - Браузери не дозволяють цього (існує browser library, яка дозволяє використовувати gRPC через проксі-сервер, проте вона нівелює переваги використання gRPC).
  - Поки gRPC ще не може використовуватись у браузері, його включення в ASP.NET Core важливе лише для проєктів, що застосовують його для комунікації між back-end серверами, для чого існує багато альтернативних протоколів.

Розуміння платформи ASP.NET Core

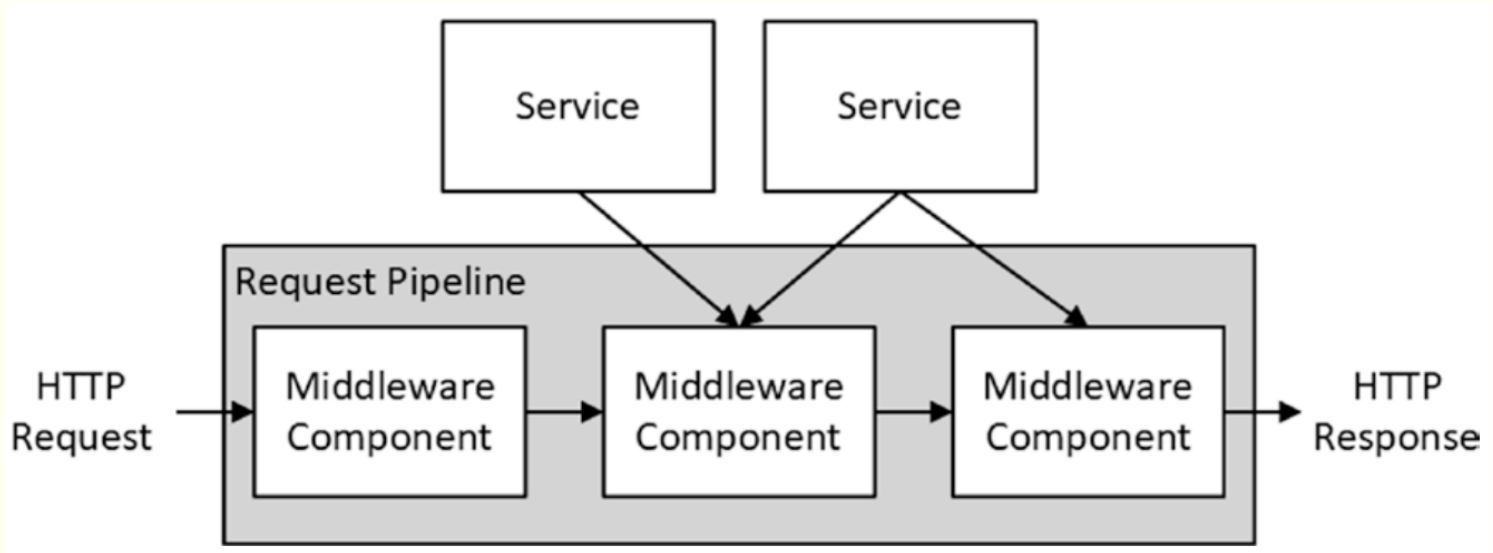
- Конвеєр запитів (request pipeline) ASP.NET Core:



  - Ціль платформи ASP.NET Core – отримувати HTTP requests та надсилати для них responses, що ASP.NET Core делегує middleware компонентам.
  - Компоненти проміжного прошарку організовані в ланцюг, який називають ***конвеєром запитів***.

- Платформа ASP.NET Core створює об'єкт, який описує конвеєр та відповідний об'єкт, що описує відгук, який буде натомість надіслано.
  - Ці об'єкти передаються першому компоненту проміжного прошарку в ланцюгу, який переглядає запит та вносить зміни у відгук.
  - Потім запит передається наступному компоненту проміжного прошарку в ланцюгу, який робить аналогічну роботу.
  - Як тільки запит пройшов весь конвеєр, платформа ASP.NET Core надсилає відгук.

Розуміння служб

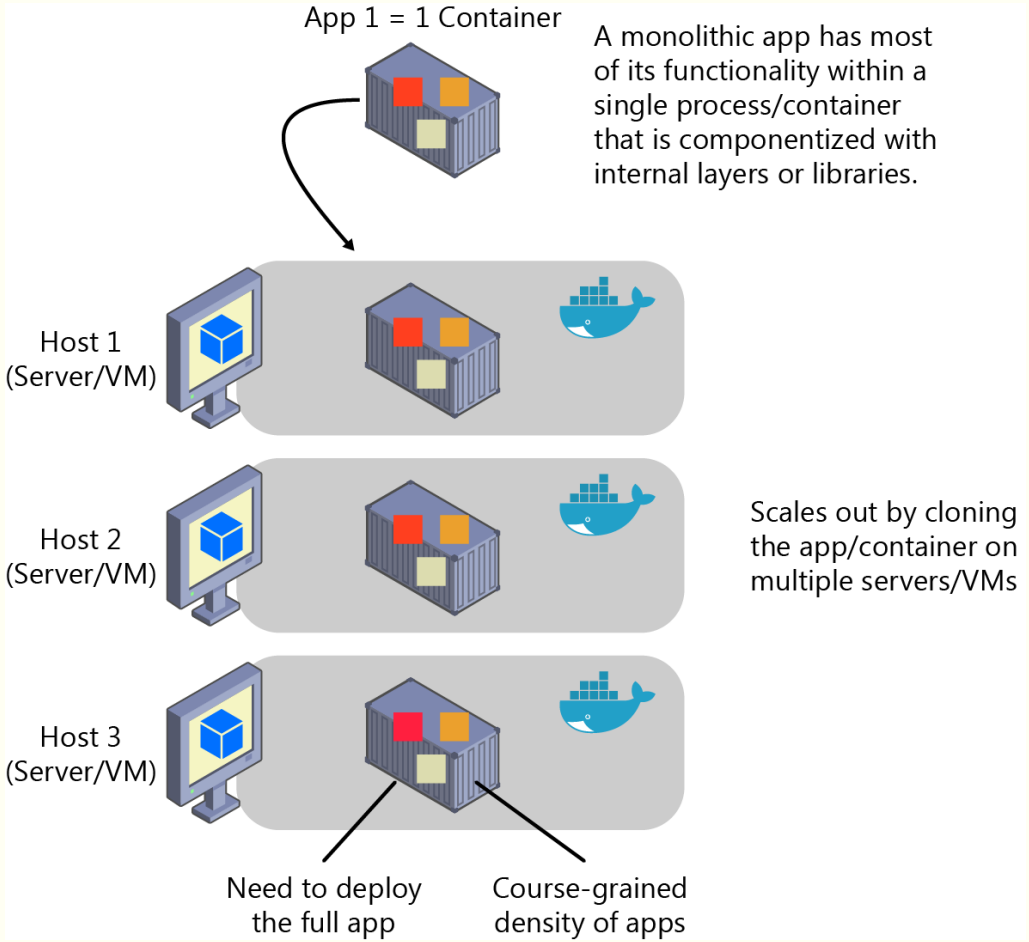


- Служби (Services) – це об’єкти, що постачають можливості для веб-додатку.
  - Будь-який клас може використовуватись як служба, а на можливості, що забезпечує служба, немає обмежень.
  - Особливістю служб є управління ними платформою ASP.NET Core через механізм впровадження залежностей (dependency injection), що дає можливість отримувати доступ до служб з будь-якого місця в додатку, включаючи middleware components.
  - Компоненти проміжного прошарку використовують тільки служби, що їм потрібні для роботи.

Загальні архітектури веб-додатків

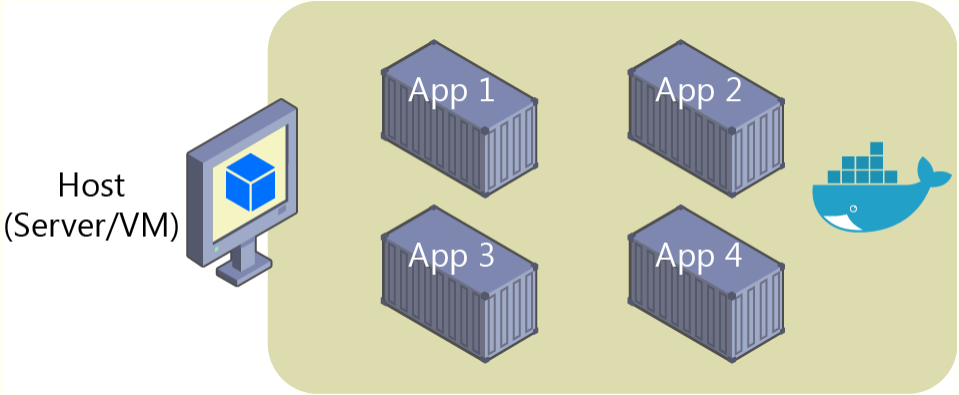
- Більшість традиційних додатків .NET розгортається у вигляді одного елемента, що відповідає виконуваному файлу, або одного веб-додатку, що виконується в домені додатків служб IIS.
  - Це найпростіша модель розгортання, яка оптимально підходить для багатьох внутрішніх і невеликих загальнодоступних додатків.
  - Проте навіть у такій простій моделі розгортання більшість бізнес-додатків використовує переваги логічного поділу на прошарки.

Монолітні додатки



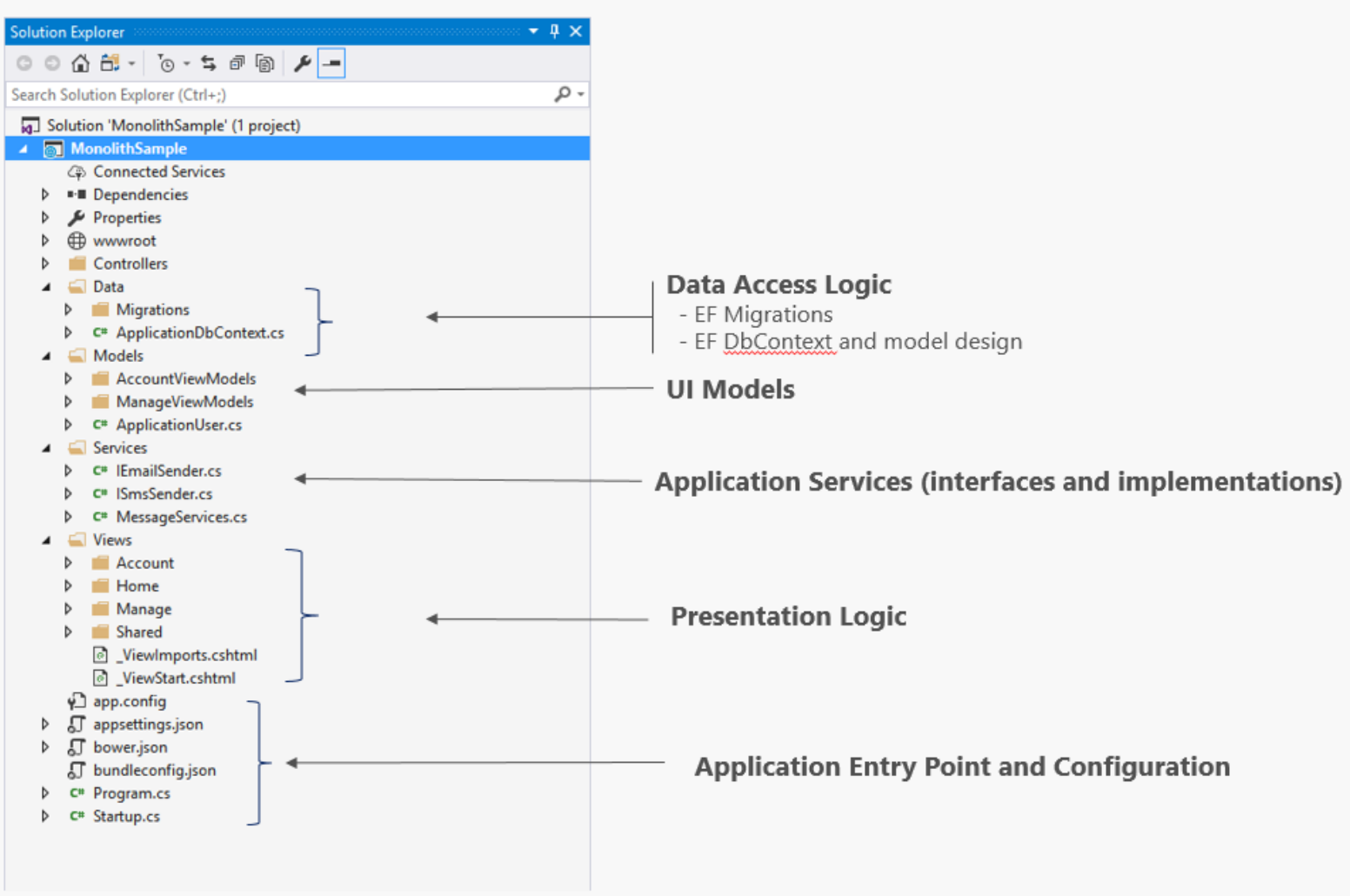
- Цей додаток може не мати монолітну внутрішню структуру і складатися з кількох бібліотек, компонентів або навіть рівнів (прикладний рівень, рівень домену, рівень доступу до даних і т. д.).
  - Зовні він буде являти собою єдиний контейнер: процес, веб-додаток або служба.
- Для управління цією моделлю ви розгортаєте один контейнер, що являє собою додаток.
  - Для масштабування просто додайте додаткові копії, розмістивши перед ними підсистему балансування навантаження.
  - Керувати одним розгортанням в одному контейнері або віртуальній машині набагато простіше.

Монолітні додатки



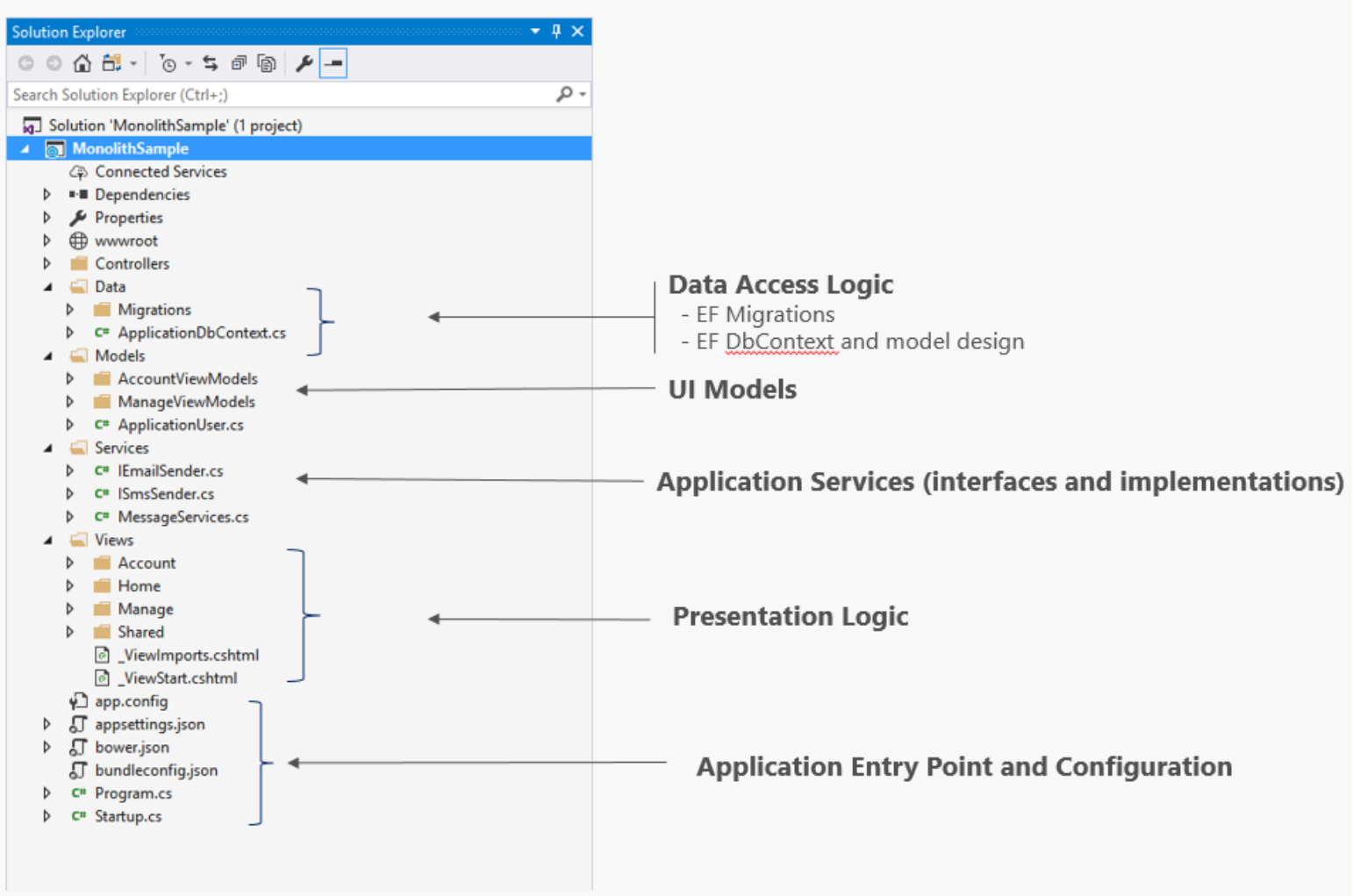
- Монолітний підхід значно поширився і використовується багатьма організаціями при розробці архітектури.
  - У багатьох організаціях додатки будувалися за такою моделлю, оскільки кілька років тому за допомогою існуючих інструментів та інфраструктури надто складно було створювати архітектури SOA, і проблем не виникало, поки додаток не починав розростатись.
- З точки зору інфраструктури, кожний сервер може виконувати багато додатків у одному вузлі та застосовувати допустиме співвідношення ефективності використання ресурсів.
- З точки зору доступності монолітні додатки потрібно розгортати цілком.
  - Якщо буде необхідно виконати зупинку та запуск, протягом періоду розгортання буде вплив на всі функціональні можливості та всіх користувачів.

Комплексні додатки



- Архітектура додатку містить принаймні 1 проєкт.
  - Тоді вся логіка додатку, що заключена в одному проєкті, компілюється в одну збірку та розгортається як один елемент.
- Будь-який створюваний у Visual Studio або з командного рядка проєкт ASP.NET Core відразу буде представляти собою комплексний монолітний проєкт.
  - У ньому буде включено всю поведінку додатку, включаючи презентацію даних, бізнес-логіку та логіку доступу до даним.

Комплексні додатки



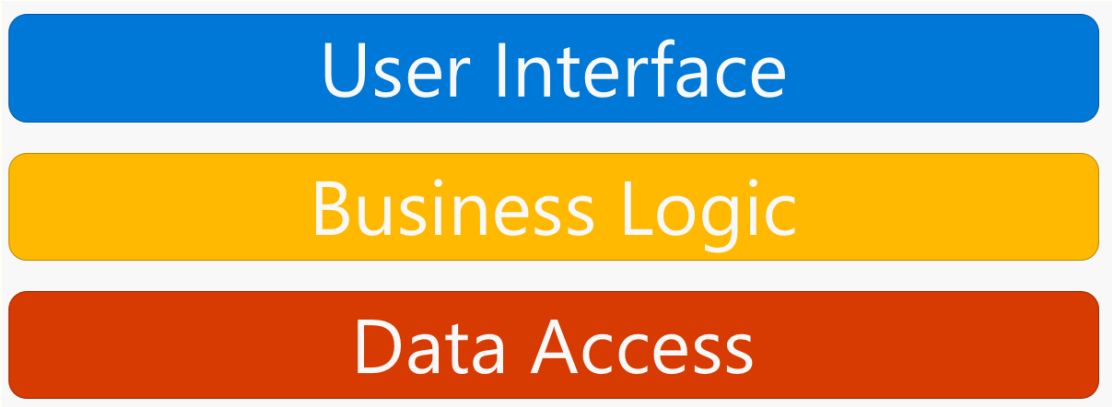
- У сценарії з одним проєктом розділення задач реалізується за допомогою папок.
  - Шаблон за умовчанням включає окремі папки для обов'язків шаблона MVC (моделі, представлення та контролери), а також додаткові папки для даних и служб.
  - При такій організації деталі презентації даних в максимально можливій степені розміщуються в папці представлень (Views), а деталі реалізації доступу до даних повинні обмежуватись класами з папки даних (Data).
  - Бізнес-логіка при цьому розміщується в службах і класах, які знаходяться в папці моделей (Models).

Проблеми монолітного рішення з одним проектом

- По мірі збільшення розміру та складності проєкту буде рости число файлів і папок.
  - Задачі, пов'язані з користувацьким інтерфейсом (моделі, уявлення, контролери), розміщуються в різних папках, які не впорядковані за алфавітом.
  - З додаванням в окремі папки конструкцій рівня користувацького інтерфейсу, наприклад фільтрів або зв’язувачів моделі, ситуація тільки погіршується.
  - Бізнес-логіка губиться в папках моделей (Models) і служб (Services), в результаті чого неможливо чітко визначити, які класи в яких папках повинні залежати від інших класів.
  - Подібна неефективна організація на рівні проєкта часто призводить до отримання погано структурованого коду.
- Для вирішення подібних проблем додатки часто організовуються у вигляді рішень, що складаються з безлічі проектів, де кожен проект розміщується в окремому прошарку додатку.
  - У додатках з багатошаровою архітектурою можуть встановлюватися обмеження на взаємодію між шарами. Таким чином вдається реалізувати інкапсуляцію.
  - При зміні або заміні прошарку будуть порушені тільки ті прошарки, які працюють безпосередньо з ним.
  - Обмежуючи залежності шарів один від одного, можна зменшити наслідки внесення змін, в результаті чого одинична зміна не впливатиме на весь додаток.

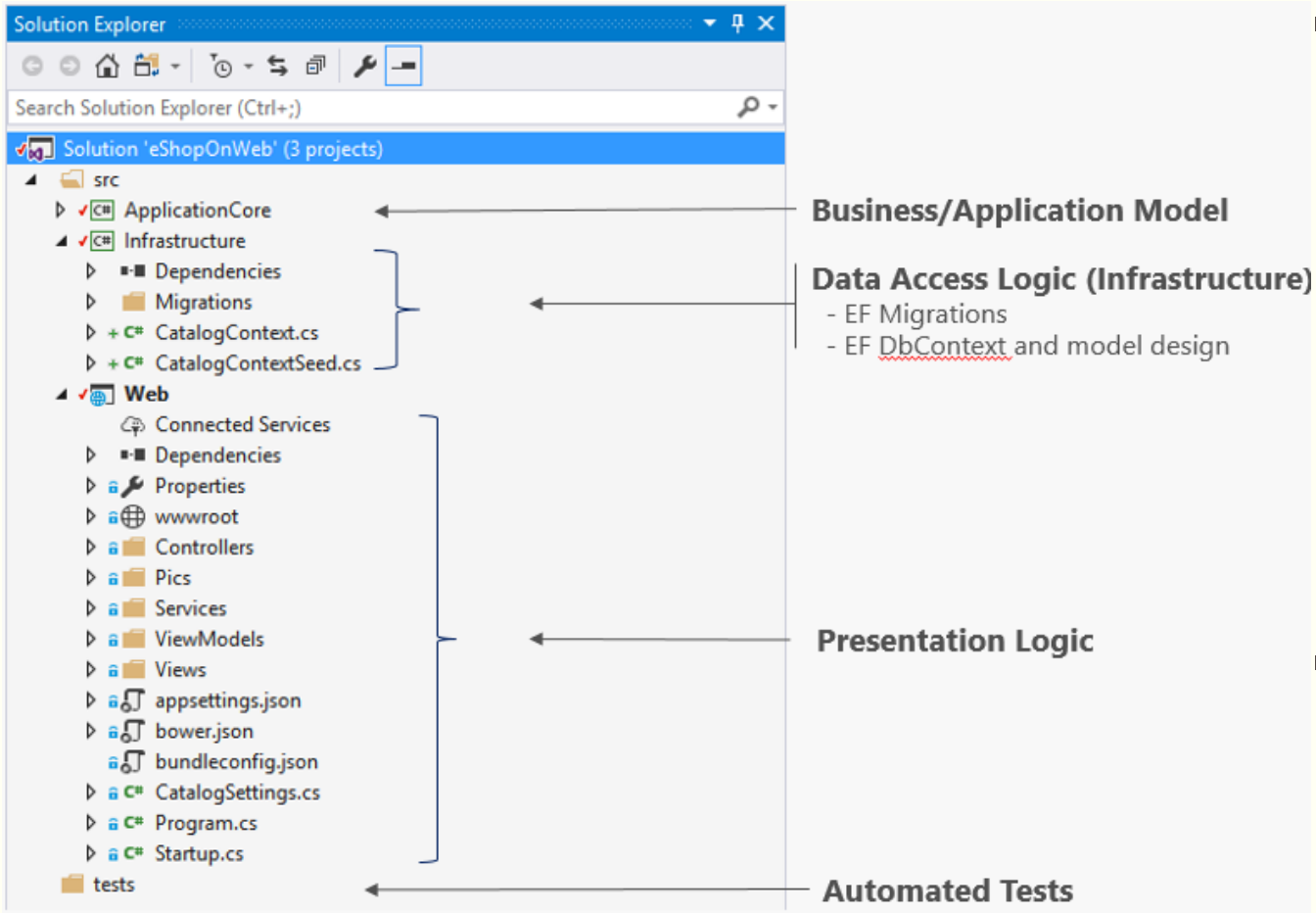
Традиційні додатки з N-шаровою архітектурою

- Прошарки типового додатку



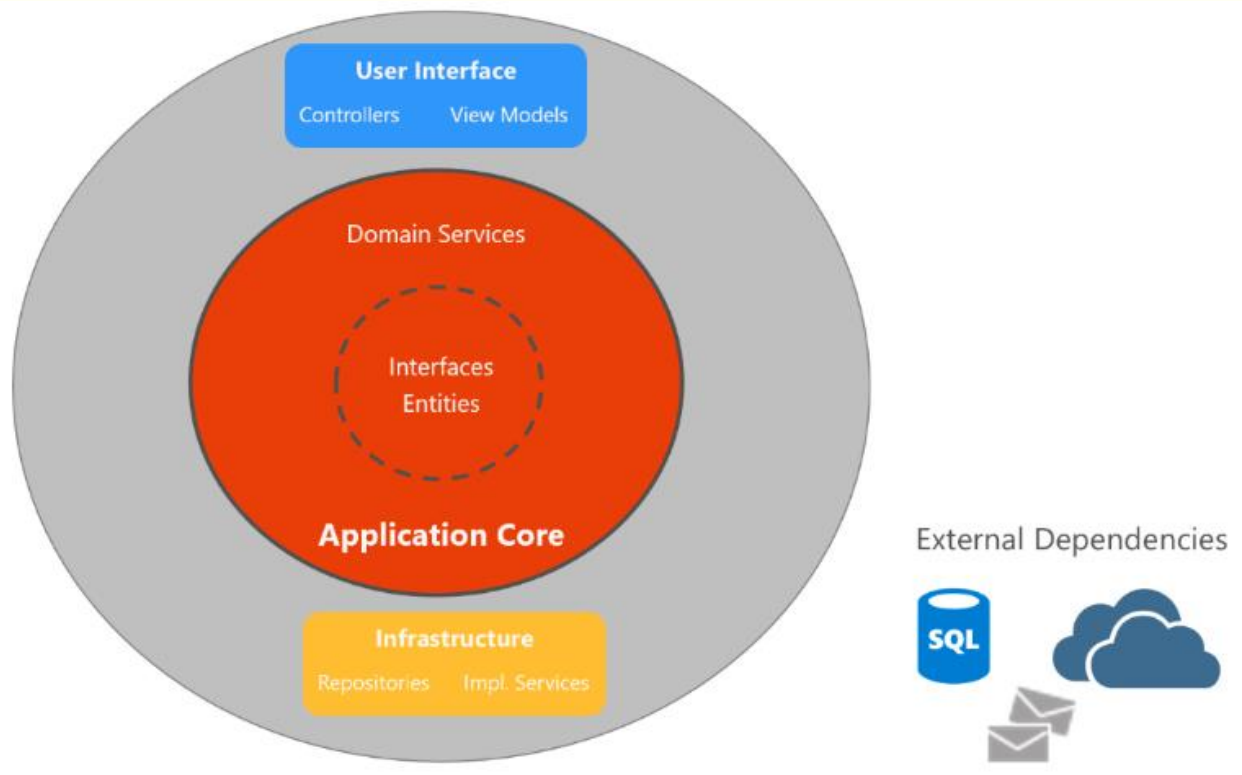
  - У рамках такої архітектури користувачі виконують запити через прошарок користувацького інтерфейсу, який взаємодіє тільки з прошарком бізнес-логіки.
  - Прошарок бізнес-логіки, в свою чергу, може викликати прошарок доступу до даних для обробки запитів.
  - Прошарок користувацького інтерфейсу не повинен виконувати запити напряму до прошарку доступу до даних і будь-якими іншими способами напряму взаємодіяти з функціями збережуваності.
  - Аналогічним чином, прошарок бізнес-логіки повинен взаємодіяти з функціями збережуваності тільки через прошарок доступу до даних.

Простий монолітний додаток, що складається з 3 проєктів



- Одним з недоліків традиційного багатошарового підходу є те, що обробка залежностей під час компіляції здійснюється згори донизу.
  - Прошарок користувацького інтерфейсу залежить від прошарку бізнес-логіки, який залежить від прошарку доступу до даних.
  - Прошарок бізнес-логики, який зазвичай містить ключові функції додатку, залежить від деталей реалізації доступу до даних (і часто від наявності самої БД).
  - Тестування бізнес-логіки в такій архітектурі часто ускладнене та потребує наявності тестової БД.
  - Для вирішення цієї проблеми може застосовуватись принцип інверсії залежностей
- Незважаючи на те, що з метою впорядкування в цьому додатку використовується кілька проєктів, він все ще розгортається, як єдиний елемент, і його клієнти взаємодіють твуют с ним как с одним веб-приложением.
  - Це дозволяє реалізувати дуже простий процес розгортання.

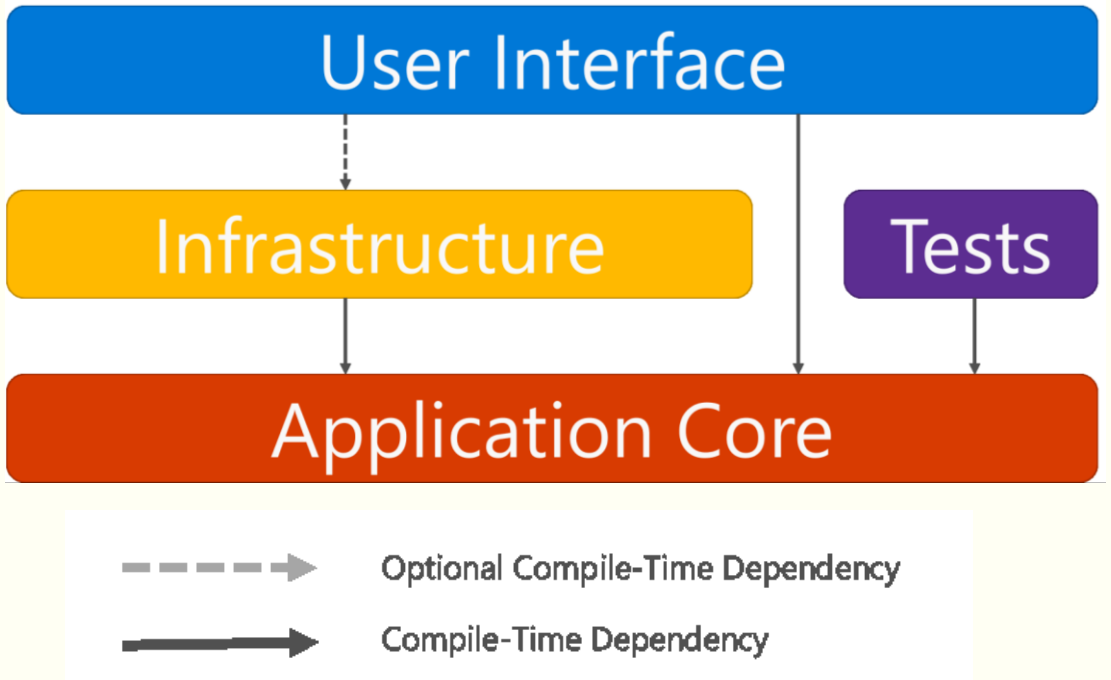
Чиста архітектура



- У рамках чистої архітектури центральним елементом додатку є його бізнес-логіка і модель.
  - Тоді бізнес-логіка не залежить від доступу до даних або інших інфраструктур, тобто стандартна залежність інвертується: інфраструктура і деталі реалізації залежать від ядра додатку.
  - Це досягається шляхом визначення абстракцій або інтерфейсів у ядрі додатку, які реалізуються типами, визначеними в прошарку інфраструктури.
- На схемі залежності направлені з найбільш внутрішнього кругу.
  - Ядро додатку називається так тому, що знаходиться в центрі схеми.
  - Воно не має залежностей від інших прошарків додатку.
  - Сутності та інтерфейси додатку знаходяться в самому центрі.
  - Відразу після них, проте все ще в межах ядра додатку, розташовуються доменні служби, які зазвичай реалізують інтерфейси, визначені у внутрішньому крузі.
  - За межами ядра додатку розташовуються прошарки користувацького інтерфейсу та інфраструктури, які залежать від ядра додатку, проте не одна від одної (обов'язково).

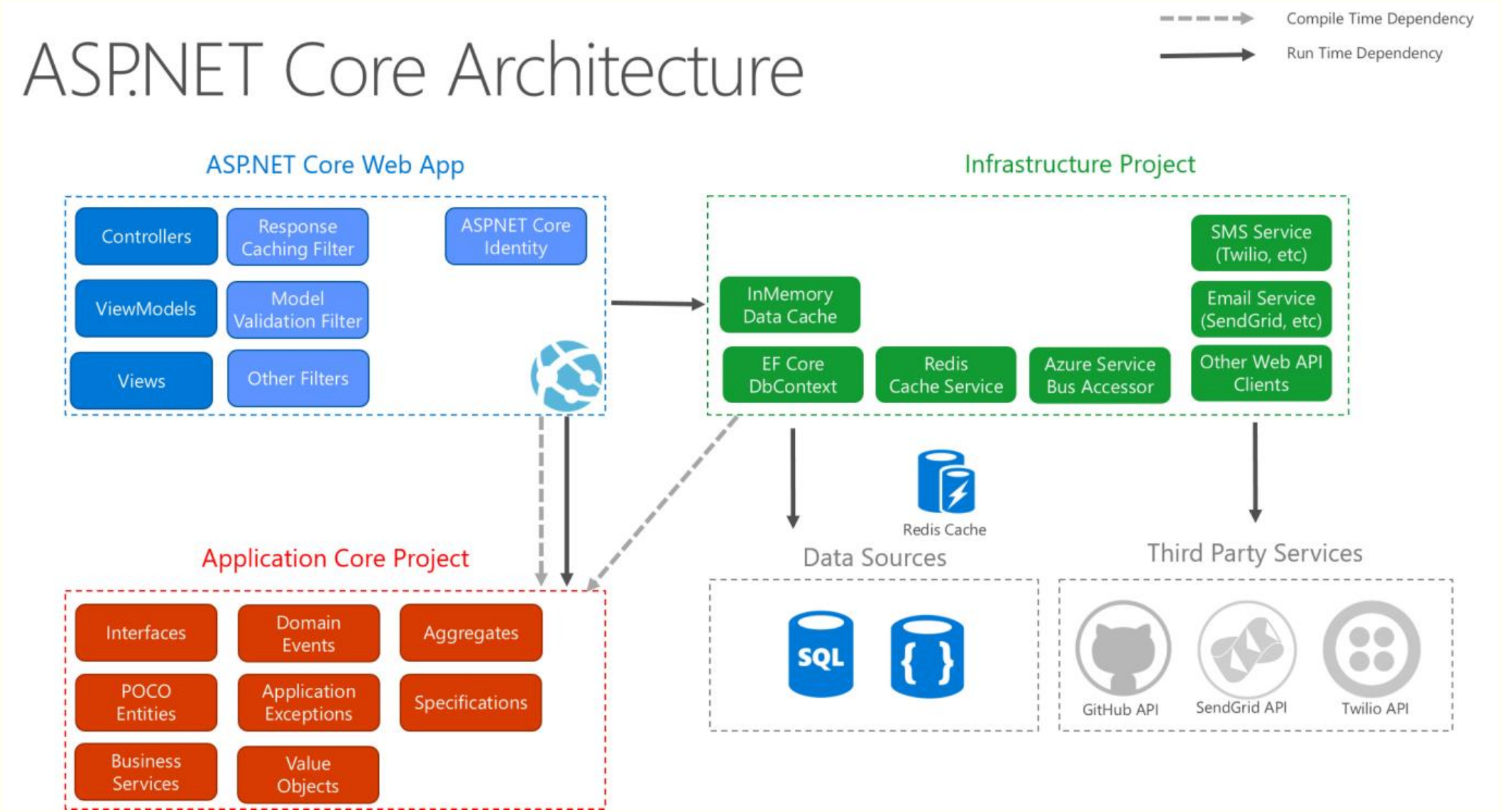
- Код еталонного додатку.

Більш звична горизонтальна схема прошарків



- Зверніть увагу, що суцільні стрілки відповідають залежностям часу компіляції, а пунктирні — залежностям, що існують лише під час виконання.
  - В рамках чистої архітектури шар користувацького інтерфейсу працює з інтерфейсами, які визначені в ядрі програми під час компіляції, і в ідеальному випадку не повинен знати нічого про типи реалізації, визначені у прошарку інфраструктури.
  - Проте під час виконання ці типи реалізації необхідні для виконання програми, тому вони повинні існувати і бути прив'язані до інтерфейсів ядра додатка за допомогою впровадження залежностей.

Детальніше представлення архітектури додатку ASP.NET Core, побудованого відповідно до цих рекомендаіцй



ДЯКУЮ ЗА УВАГУ!

Наступне питання: