Kiến trúc ứng dụng web trong ASP.NET Core

Để phát triển một ứng dụng ASP.NET Core bạn không chỉ cần hiểu cách làm việc của bản thân framework, bạn còn phải biết cách tổ chức code để ứng dụng bạn viết ra không phải là mớ code hổ lốn mà người ta thường gọi là spaghetti code. Để làm điều đó bạn cần phải hiểu về kiến trúc thường sử dụng trong ứng dụng ASP.NET Core cũng như một số nguyên lý chung thường sử dụng trong thiết kế phần mềm.

NỘI DUNG CỦA BÀI [Ẩn]

1. Vấn đề thường gặp khi tổ chức code
2. Ứng dụng nguyên khối và cấu trúc project
3. Kiến trúc đa lớp (N – layer)
4. Kiến trúc đa lớp kiểu "hành tây"
5. Kết luận

Vấn đề thường gặp khi tổ chức code

Trong các bài thực hành về Razor Pages và MVC bạn có thể để ý thấy chúng ta viết code tuân theo một trình tự.

Đầu tiên bạn xây dựng các lớp model thể hiện dữ liệu theo nghiệp vụ (domain model). Nếu bạn sử dụng Entity Framework/Entity Framework Core, bạn sẽ xây dựng lớp Context. Tiếp theo, bạn sẽ xây dựng một class hỗ trợ truy xuất dữ liệu tập trung (Service hoặc Repository). Class này có thể sử dụng context hoặc trực tiếp làm việc với file dữ liệu.

Từ đây về sau, bạn có thể vác các domain model và Service/Repository đến controller (MVC) hoặc model class (Razor Pages). Khi sử dụng Service/Repository trong controller hoặc model class, bạn cũng thường sử dụng dependency injection để tự động sinh và chèn object phù hợp với yêu cầu.

Các ví dụ trong bài giảng này thường kết thúc ở các thao tác CRUD dữ liệu. Chúng ta không có cơ hội thực hiện project phức tạp hơn.

Tuy vậy bạn có thể tự hỏi, tại sao lại thực hiện như vậy? Và nếu dự án tiếp tục mở rộng ra, bạn sẽ tổ chức code như thế nào?

> Ở đây chúng ta tạm quên đi vài bước trong quy trình phát triển phần mềm. Giả sử (1) các yêu cầu của phần mềm đã rõ ràng, (2) bạn đã nắm chắc nghiệp vụ liên quan, (3) bạn đã học được cách sử dụng một framework. Cái bạn muốn là phát huy kỹ thuật lập trình của framework mình đã học để làm ra sản phẩm phần mềm.

Để tổ chức code dự án hợp lý, bạn cần hiểu kiến trúc của ứng dụng và một số nguyên lý liên quan.

Ứng dụng nguyên khối và cấu trúc project

Ứng dụng nguyên khối (monolithic appliacation hay monolith) là loại ứng dụng chạy hoàn toàn độc lập chỉ trong một tiến trình riêng và được triển khai như một đơn vị duy nhất.

Ví dụ thế này cho dễ hiểu. Bạn tạo một dự án ASP.NET Core MVC mặc định, build nó ở chế độ Release và mang nó sang máy chủ nơi nó sẽ hoạt động. Ứng dụng này chạy như một ứng dụng console thông thường trong một tiến trình. Nó là một ứng dụng nguyên khối. Kể cả khi ứng dụng đó có tương tác với cơ sở dữ liệu đi chăng nữa, nó vẫn là một ứng dụng nguyên khối do mọi chức năng của nó chạy gói gọn trong tiến trình này.

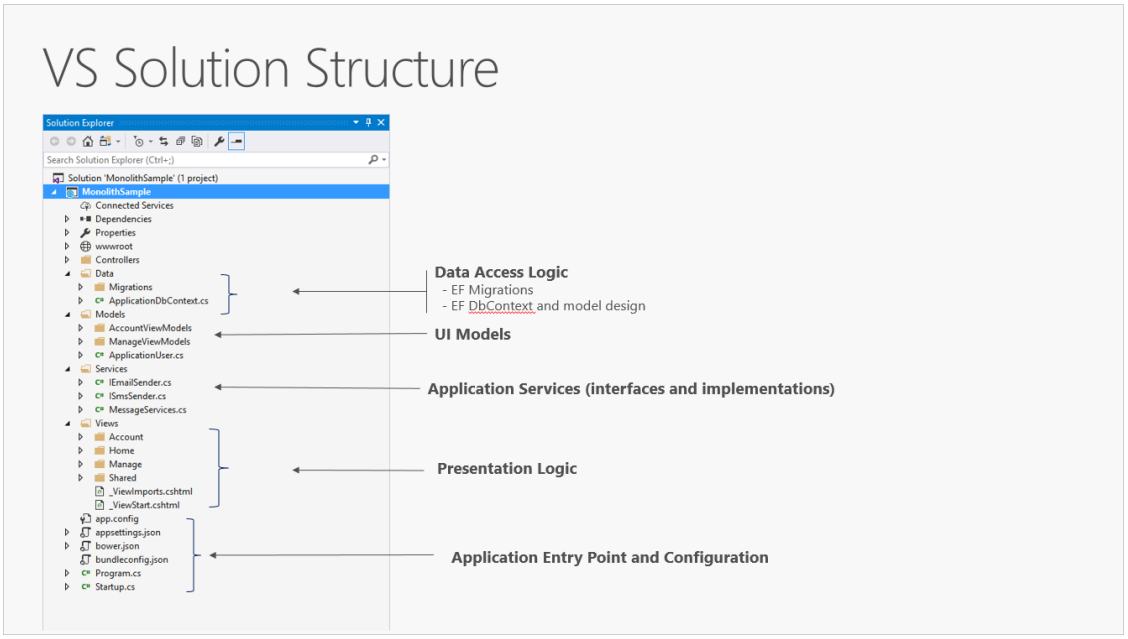
Trên thực tế, ứng dụng nguyên khối như vậy là rất phổ biến. Nếu không có nhu cầu gì đặc thù, người ta sẽ xây dựng ứng dụng ở dạng nguyên khối như vậy do sự đơn giản trong xây dựng và triển khai.

Thông thường, một ứng dụng nguyên khối có thể được xây dựng từ một project duy nhất, hoặc cũng có thể từ nhiều project.

Nếu ứng dụng tạo ra từ một project duy nhất, người ta cũng gọi là ứng dụng all-in-one. Trong ứng dụng all-in-one, mọi thứ nằm trong cùng một project, được dịch ra cùng một assembly, triển khai trong cùng một đơn vị, và thực thi trong cùng một tiến trình.

Trong tình huống này người ta sử dụng thư mục để phân chia chức năng (separation of concerns).

Dưới đây là ví dụ về một ứng dụng dạng all-in-one sử dụng thư mục để phân chia chức năng của các thành phần code.



Đây là cách tổ chức code mặc định của ASP.NET Core MVC. Đối với dự án Razor Pages, cách thức tổ chức code mặc định cũng tương tự như vậy.

Với các đề tài nhỏ, cấu trúc này có thể phù hợp.

Tuy nhiên, khi quy mô ứng dụng tăng lên và bổ sung thêm chức năng, vấn đề sẽ xuất hiện: Các thư mục mới liên tục được thêm vào; Khó theo dõi được vai trò của các thư mục; Cấu trúc thư mục quá phức tạp; Sự phụ thuộc giữa các class khó theo dõi và kiểm soát; v.v..

Tình trạng tổ chức code như vậy thường được gọi là "spaghetti code" – sự rối loạn, khó theo dõi và quản lý code – code trộn nháo nhào như món mỳ Ý.

Đây cũng là tình trạng phổ biến khi các bạn mới học lập trình ASP.NET Core.

Để giải quyết tình trạng này, người ta thường tách một dự án lớn thành các dự án con trong một solution. Mỗi dự án con tạo thành một lớp (layer) của ứng dụng.

Kiến trúc đa lớp (N – layer)

Khi một ứng dụng bắt đầu trở nên phức tạp, một trong những cách tổ chức quản lý là tách nó thành các project riêng.

Tách một project lớn thành nhiều project nhỏ nằm trong cùng một solution có lợi cho tổ chức code. Bạn không còn phải làm việc với những project quá phức tạp với rất nhiều file và thư mục. Mỗi project con sẽ đơn giản gọn gàng hơn.

Tuy nhiên bạn không nên tách project một cách tùy ý.

Nhìn chung trong một ứng dụng nào người ta có thể nhóm các class của nó thành 3 nhóm: giao diện người dùng (User Interface), xứ lý nghiệp vụ (Business Logic), truy xuất dữ liệu (Data Access).

Nếu bạn tách một project lớn thành 3 project con chứa các clas theo nhóm như trên, mỗi project có thể được gọi là một **layer**. Ứng dụng của bạn giờ được tổ chức theo kiến trúc đa lớp (N – layer).

Dưới đây là một ví dụ về tổ chức solution theo kiến trúc đa lớp.

Lưu ý rằng, layer thể hiện sự phân chia về mặt logic của ứng dụng. Nghĩa là, dù tách thành nhiều layer, mỗi layer biên dịch về một assembly riêng, nhưng khi thực thi, các assembly đều tải vào cùng một tiến trình. Do vậy, loại ứng dụng này vẫn là ứng dụng monolith.

Trong tình huống các thành phần của ứng dụng được phân chia về mặt vật lý và thực thi trong các tiến trình khác nhau (trên cùng một máy vật lý hoặc trên các máy khác nhau), mỗi phần như thế được gọi là một **tier**.

Một ứng dụng N-layer cũng thường được triển khai (chạy) trong một tier duy nhất.

Kiến trúc đa lớp kiểu "hành tây"

Tên gọi "kiến trúc đa lớp" có lẽ không hề xa lạ với các bạn. Kiến trúc đa lớp như trên mô tả áp dụng được với nhiều loại ứng dụng chứ không chỉ riêng cho ứng dụng web. Đây là một loại kiến trúc rất tổng quát.

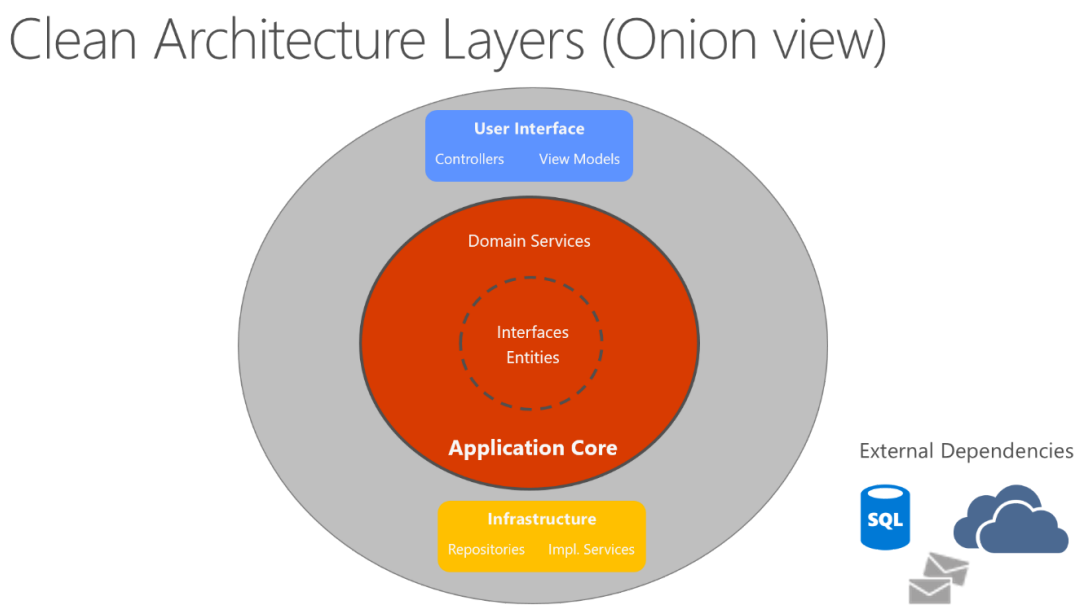
Vì nó quá tổng quát bạn sẽ thấy khó vận dụng nó. Thực tế ít bạn có thể vận dụng được nó. Cuối cùng lại dẫn đến spaghetti code.

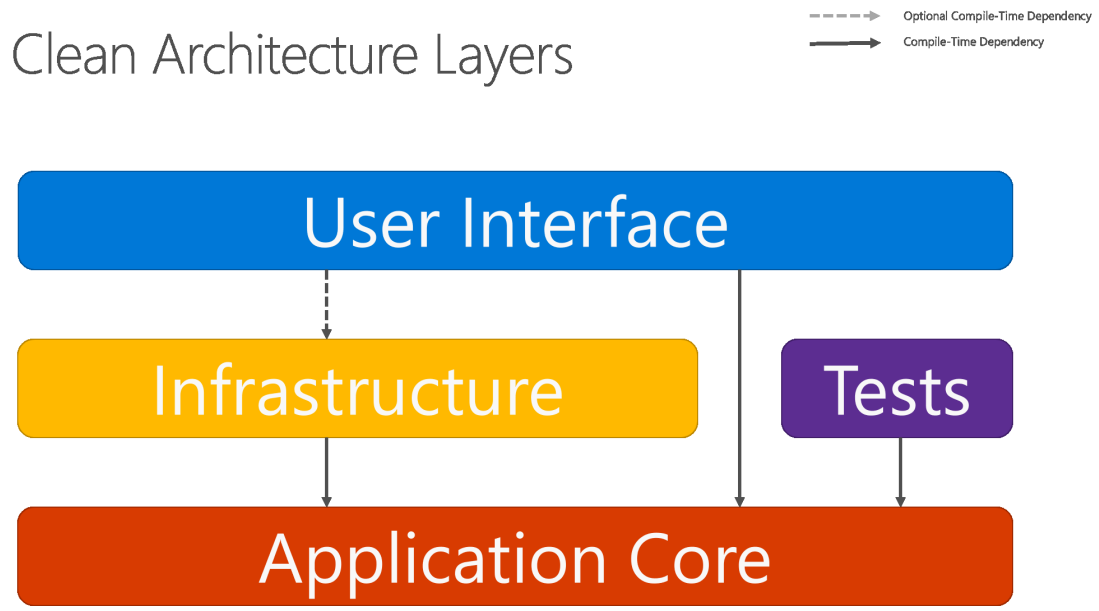
Trong dự án ASP.NET Core (MVC hoặc Razor Pages) bạn có thể vận dụng một biến thể của nó gọi là **clean architecture**. Kiến trúc này đôi khi còn được gọi là kiến trúc hành tây (onion architecture).

Clean architecture là kết quả của vận dụng nguyên lý Dependency Inversion kết hợp Domain-Driven Design đặt trong khuôn khổ của N-layer.

Theo kiến trúc này, bạn chia ứng dụng làm 3 layer: Application Core, Infrastructure, User Interface.

Có hai cách nhìn khác nhau đối với kiến trúc này: onion view và layer view.



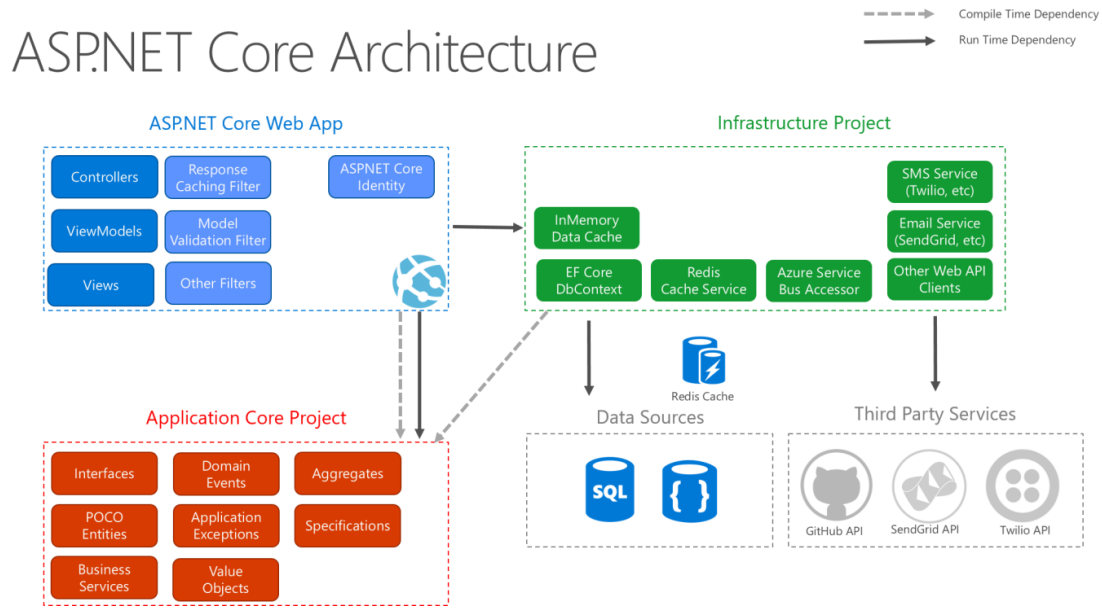


Application Core chứa các lớp domain model (Entities), các giao diện (Interfaces) cần thiết. Đây cũng là thành phần cần xây dựng đầu tiên trong solution. Trong các bài thực hành bạn đã thấy điều này. Core là thành phần độc lập nhất trong kiến trúc này. Tất cả các thành phần khác sẽ phụ thuộc vào Core.

User interface bao gồm những thành phần quen thuộc của MVC: controllers, view models, view.

Infrastructure chứa những thành phần như context của ORM, Web API client (nếu cần gọi), v.v..

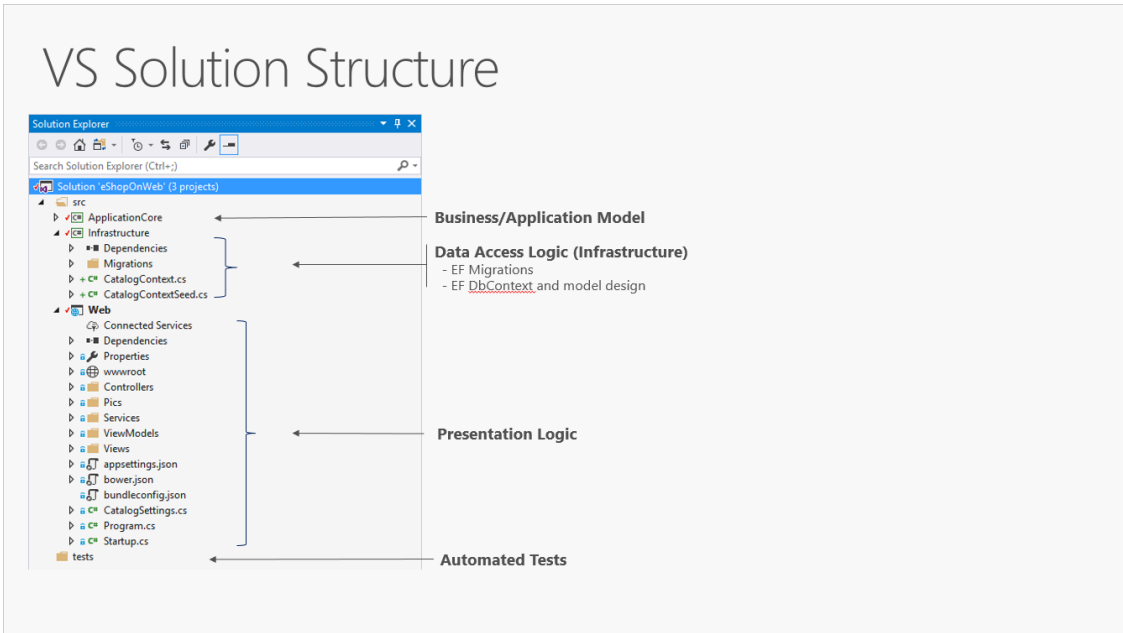
Một cách chi tiết, mỗi layer sẽ bao gồm những thành phần như sau (và sự phụ thuộc giữa chúng):



Lưu ý về sự phụ thuộc giữa các layer. Sự phụ thuộc khi viết code (compile time) chính là việc các project tham chiếu nhau (để sử dụng class trong mỗi project). Sự phụ thuộc khi

chạy (runtime) là việc object của class này gọi phương thức của object, hoặc gọi đến dịch vụ khác.

Dưới đây là một ví dụ về cách tổ chức các project trong solution theo kiến trúc clean.



Kết luận

Trong bài học này chúng ta nhắc đến vấn đề tổ chức code trong dự án ASP.NET Core theo góc nhìn của kiến trúc ứng dụng. Nhìn chung khi xây dựng ứng dụng ASP.NET Core chúng ta có thể sử dụng kiến trúc clean. Loại kiến trúc này kết hợp ưu điểm từ nhiều nguyên lý thiết kế khác nhau và được Microsoft khuyến nghị sử dụng trong các tài liệu hướng dẫn chính thức.

> + Nếu bạn thấy site hữu ích, trước khi rời đi hãy **giúp đỡ** site bằng một hành động nhỏ để site có thể phát triển và phục vụ bạn tốt hơn.
> + Nếu bạn thấy bài viết hữu ích, hãy giúp **chia sẻ** tới mọi người.
> + Nếu có thắc mắc hoặc cần trao đổi thêm, mời bạn viết trong phần **thảo luận** cuối trang.
> Cảm ơn bạn!