**ASP.NET CORE**

1. **Khái Niệm**

* Đây là một bản thiết kế lại hoàn toàn của .NET Framework
* .net core là Open Source (Mã nguồn mở hoàn toàn) và dc phát triển chính là Microsoft và cộng đồng mã nguồn đặt trên GitHub
* Webforms,Windows Forms,WPF chúng không có trong .net Core tương có thể có
* Sao này vào năm 2020 sẽ có .Net 5 (là sự kế hợp của .net framework và .net core) và chỉ phát triển chính là .net core
* .net Standard là thư viện chuẩn khi ta Bulid phiên bản .net Standard thì cái DLL này chạy cho cả .net framework và .net core
  + Tiêu chuẩn .NET là một đặc tả chính thức của các API .NET nhằm mục đích có sẵn trên tất cả các triển khai .NET.
  + Nó xác định một tập hợp các quy tắc thống nhất cần được tuân theo trên tất cả các triển khai .NET
* Sự khác nhau giữa .net core và Famework:
  + .net frame là môi trường thực thi cũ hơn chỉ chạy dc trên windows , tr dây ta có .net Mono là phiên bản mini chạy dc trên linux mà k phổ biến
  + .core có thể chạy đa nền tảng
* Tính năng quan trọng của ASP.NET Core
  + Chạy trên mọi hệ điều hành như mac window, linux và là open source
  + Thống nhất giữa mvc va api
    - MVC là: xây dựng WEB APP có giao diện
    - API là tạo ra giao diện lập trình ứng dụng các api giúp tương tác vs DB và nó k tương tác vs người dung
  + ASP core nó có thể tự host dc
  + Cơ chế Denpendency Injection dc cung cấp sẵn dễ dàng tích hợp vs client-side framworks như Angular,Knockout etc.
  + Cơ chế request pipeline mới gọn nhẹ
  + Dung chung nutgetpaket

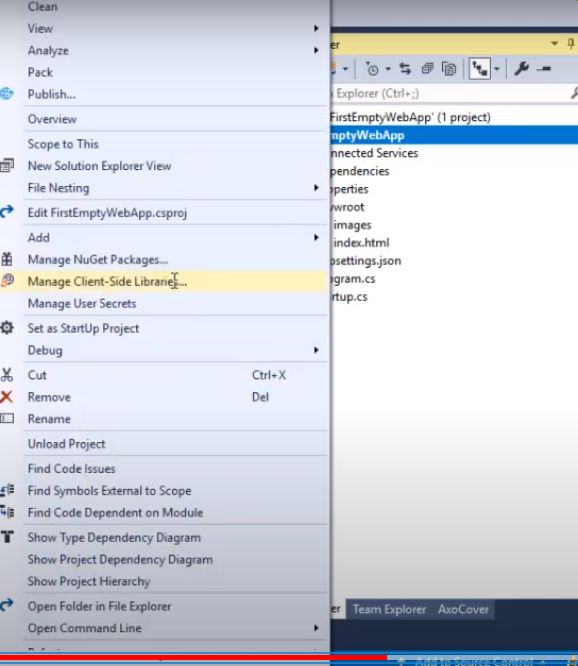
1. **Dotnet command line trong .NET**

Công cụ Donet command dung để khi ta làm việc trên mac hay linux mà ta k có sử dụng visual studio

1. Đầu tiên ta vào forlder nơi sẽ lưu trữ dự án sao đó ta gõ cmd và dung lệnh (dot net new web) nó sẽ lấy tên thư mục hiện tại đặt tên cho dự án và tạo ra những file cần thiết
2. **ASP.NET Core Application Structure (cấu trúc dự án ASP.net Core)**

* WWWroot
* Dependencies
* New project system (hệ thống dự án mới)
  + TargetFramework(mục tiêu framework)
  + Nuget Packages

khi tạo một project đầu tiên ta sẽ có như sao

* Connected services : dung để kết nối đến 1 web service bên ngoài vào (Azure Storage ,WCF)
* Properties: Chứa Các **launchSettings.json** (chứa các profiles thực thi ứng dụng vd như ta thay Ctrl F5 or F5 thì chạy các **CommanName**: như IISExpress or Project) và trong mỗi profiles trong **launchSetting** thì nó sẽ chạy 1 cái môi trường trong **appsettings.json** và ta có thể tạo them nhìu môi trường trong appsettings để chạy đa nền tảng , **launchSettings.json chỉ chạy trên local còn trên server sẽ chạy appsetting.json**
* Wwwroot : là nơi chứa css ,html , file tĩnh , ảnh ,….
* Dependencies
  + Chứa các nugget mạc định sẽ appnetcore app(phiên bản)
  + SDK: là gói môi trường liên quan
* Cơ chế tải Bootrap rồi jquery
  + Gọi là Libraner client side
  + Cách sử dụng
    - Ta chuột phải vào project và chọn Manage Client side Libraner 
    - Tiếp đến nó sẽ sinh ta 1 file libman.json ta chỉ cần cấu hình cho libraries và ctrl F là tự động tải về
* **appsettings.json :** là nơi chứa các file cấu hình chung dung để thiết lập đa môi trường
* **Program.cs và Statup.cs: là 2 file chính để chạy ứng dụng dc (sẽ tìm hiểu kỹ sau)**

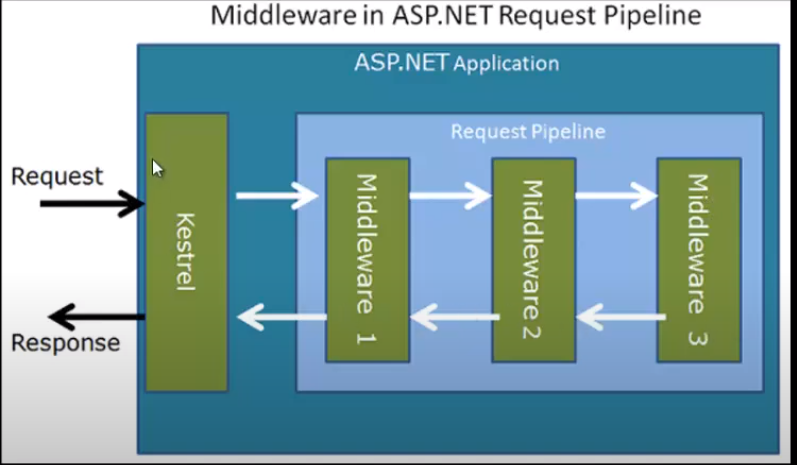
1. **Cơ Chế Application Startup**
2. Program.cs :tạo 1 web server với cái Main Method của nó
   * Tất các application (ứng dụng) trong .net Core điều là application (ứng dụng) Console
   * Application Console là gì?
     + Là ta sẽ chạy và cho kết quả trên màn hình đen (cửa số dòng lệnh)
     + Khác vs chạy console bt vs web tạo cái hosting để nó chạy web side còn lại khac j console bt
     + Điều chứa phương thức void Main
   * Trong Program ta có pt **BuildWebHost** là 1 pt tĩnh nó cấu hình build và trả về 1 đối tượng Host,
     + Phương thức **BuildWebHost** sử dụng class **WebHost**. Method **CreateDefaultBuilder**của class **WebHost** có trách nhiệm khởi tạo thể hiện **WebHostBuilder**với các cấu hình được yêu cầu.
     + **CreateDefaultBuilder dc thực hiện như sao**
       - Cấu hình Kestrel như là web server
       - Đặt thư mục gốc của ứng dụng sử dụng **Directory.GetCurrentDirectory()**
       - Load cấu hình từ
         * Appsettings.json
         * Appsettings.{Environment}.json
         * Load các cấu hình riêng (user secrets) khi chạy ứng dụng ở môi trường Development
         * Load các biến môi trường (Environment Variables)
         * Đọc các tham số truyền vào ứng dụng qua Command line argument
       - Bật logging
       - Tích hợp **Kestrel**chạy với IIS
3. Startup.cs : Configures(cấu hình) services cần thiết cho ứng dụng để nó có thể xử lý các request
   * Là một class đơn giản k kế thừa hay triển khai bất cứ class hay interface nào có 2 chức năng chính:
     + Cấu hình request pipeline để xử lý các request gửi đến ứng dụng
     + Cấu hình các services cho phần denedency injection
     + Trong startup.cs ta có 2 phương thức chính là :
       - ConfigureServices : cho phép ta thêm hoặc đăng ký các service vào ứng dụng
         * Phần khác của ứng dụng có thể cần đến các service này cho dependency injection(là 1 tính năng mới trong asp.core)
         * Vd như ta services.AddMvc(); services.AddSession();
       - Configure: cho phép ta cấu hình cho HTTP Request Pipeline chỉ ra cách mà ứng dụng cần phải hồi đáp các HTTP Request và các thành phần tạo nên request pipeline dc gọi là Middleware
4. **Kestrel Web Server**
5. **Khái niệm:**

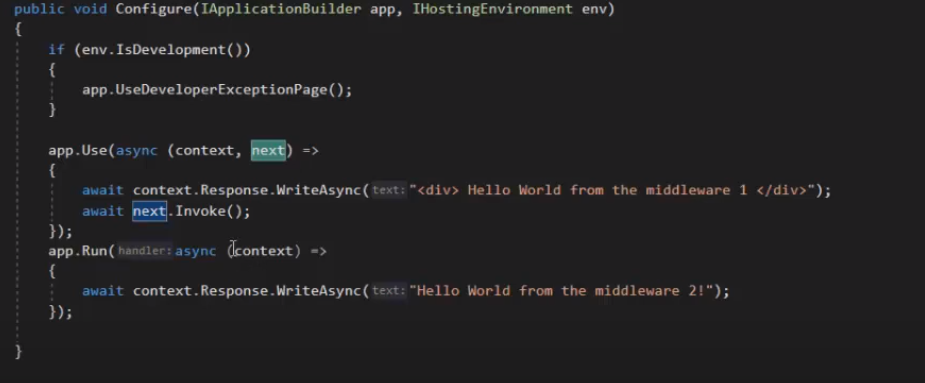
* Kestrel lá 1 web server là máy chủ web lắng nghe HTTP Server
* Lá web server open source , chạy đa nển tảng , hướng sự kiện và chạy bất đồng bộ
* Và nó dc phát triển là host chính cho asp core

1. **Tại sao dùng kestrel**

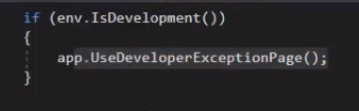
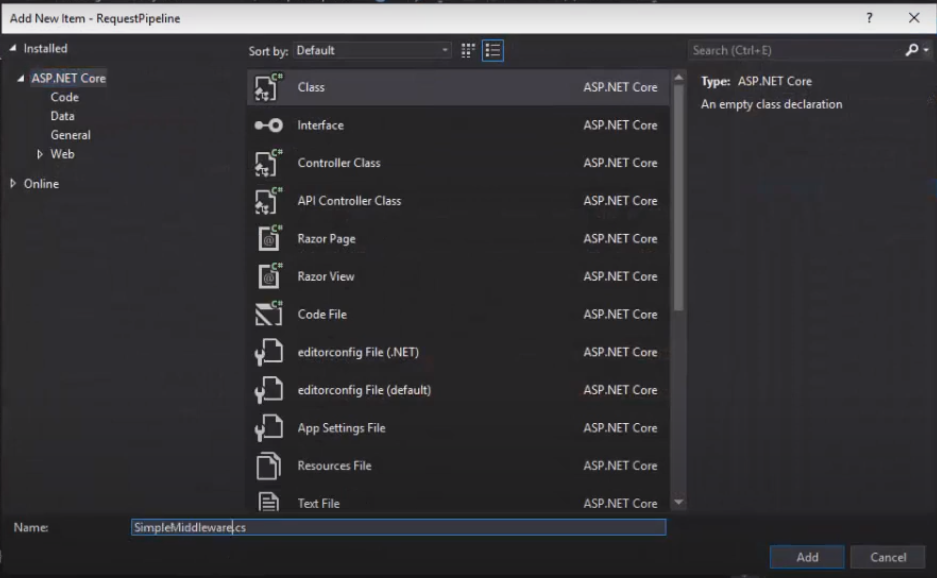
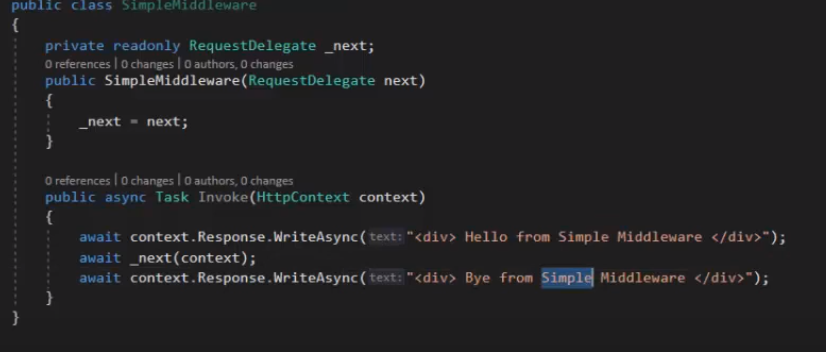
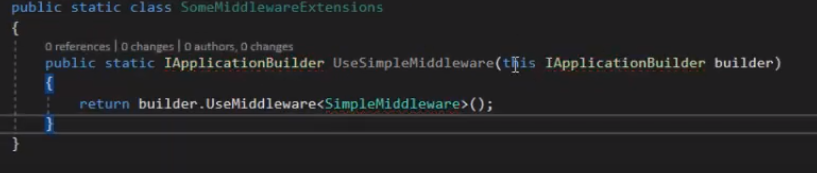
* Tr đây ta sử dụng .net cũ sử dụng IIS cũng là 1 web server đầy đủ tính năng và cũng vì thế mà IIS rất nặng và chậm , đặc biệt là IIS dc gắn trặc vs windows mà ta asp.net core đang hướng theo đa nền tảng
* Nên Ta cần 1 web server mới cho việc chạy đa nền tảng Từ Đó Kestrel xuất hiện
  + Kestrel có 2 cách để sử dụng
    - Self Hosting: tức là khi ta chạy chương trình ta chạy IIS press thì t có thể chọn cái host có tên cùng vs tên dự án
    - Behing another Web Server : cách 2 đó là t thông qua 1 máy chủ khác
      * Vd trên windows là IIS
      * **Trên linux Nginx , Apache**
* **Vì kerstrel**  là web server cơ bản nên thiếu rất nhiều tính năng vd như bảo mật, chức chỉ bảo mật HTTps,… vì thiếu khá nhìu tính năng cần thiết cho websever nên ta thường dùng **Cách thông qua 1 máy chủ để kết nối vs Kerstrel (Behing another Web Server)**
* Ngoài kerstrel ra asp core còn có hổ trợ 1 webserver khác là **HTTP.sys** giành cho window và ta có thể cầu hình trong PT CreateHostBuilder
  + webBuilder.UseStartup<Startup>().UseHttpSys();
  + <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/httpsys?view=aspnetcore-3.1>

# Middleware và Request Pipeline trong ASP.NET Core

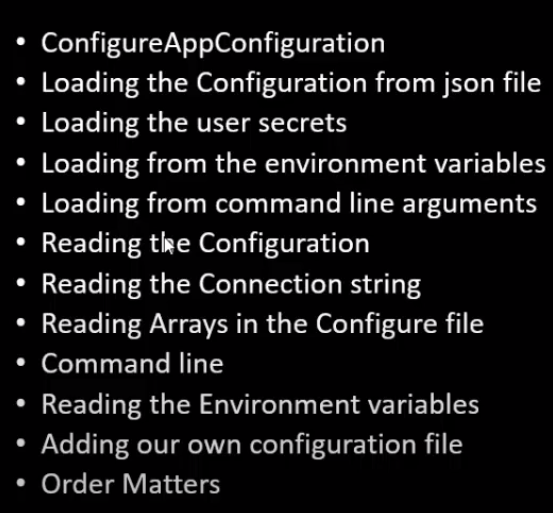
1. Middleware
   1. Là 1 thành phần ở giữa nó can thiệp vào request chạy qua request pipeline và can thiệp vs cả Response quay về nữa
   2. Mỗi middleware thao tác vs request và nhận kết quả từ các middleware trước vd ở tấm lọc nước 1 sẽ lọc ra than từ đó tấm lọc nước 2 sẽ nhận vào dòng nước k có than và tiếp tục làm nv của mình vả trả về tấm lọc nước kế tiếp khi chạy đến tấm lọc nước cuối cùng thì nó sẽ trả ngược về tấm lọc nước tr đó cứ như vậy mỗi request sẽ đi qua MiddleWare 2 lần
   3. MiddleWare cuối gọi là terminating the pipeline
2. Request Pipeline : là cơ chế giúp request nhận request cho đến xử lý và trả về response Request Pipeline chỉ ra cách dể ứng dụng làm việc vs các request gồm những bước nào
   1. do ta cấu hình các request pipeline từ request đến thông qua trình duyệt cho đến trã dữ liệu
   2. vd như đường ống nước ,khi ta nhận vào ở giữa sẽ có những tấm lọc nước (lọc cát , lọc các chất bẩn trong nước ) và mỗi thành phần trong tấm lọc nước sẽ ảnh hưởng đến dòng nước(nhiệm vụ) chạy qua và cho ra kết quả
3. Luồn đi khi nhận request đến khi reponse
   * 1. Nhận request s đó đi qua server Kesterl
     2. Tiếp đến ta đi vào Requset PiPeLine (trong Pipeline sẽ có những Middle Ware ) và sao khi xử lý xog ta sẽ đi ngược về Server Kestrel và trả Respone
     3. 
4. Thực hành
   * 1. Cách tạo và sử dụng app.use va app.run
        1. app.use



Trong app.use có parameter là next đó là cách t tiếp tục chạy middler ware 2 phía dưới trong app.use ta phải có dòng await next.Invoke(); thì nó mới chạy middler kế tiếp

* + 1. App.run giống như là midder ware cuối cùng vậy
    2. **Tạo Middleware bên ngoài :**nếu để ý thì tất cả app.Use đều là middleware isline(tức là những dòng code trong mỗi middleware điều viết cả vào trong File Startup.cs) điều này ảnh hưởng khi mỗi middleware có chục dòng code như vậy rất khó quản lý thay vì viết 1 nùi code trong Startup.cs thì ta tạo ra middleware bên ngoài như trong if từ đó ta chỉ cần gọi nó ra thôi
* Đầu tiên ta tạo forder đăt tên là Middlewares và từ đó khi ta tạo bất kỳ middleware nào cũng bỏ vào trong thư mục này
* Tiếp đến ta tạo file .cs đặt tên middleware mà mình muốn 
* Cách viết 1 middleware :
* Để làm 1 midd khai báo public cho contructor k khai báo static va ít nhất 1 tham số
* Class midd phải dinh nghĩa pt public là Invoke nhận 1 tham số HTTPcontext trả vế 1 Task
* Khi tạo xog ta dk cho nó
* Trên là cách 1 còn có 1 cách nữa là ta tạo Extensions
* Ta cũng tạo folder đặt tên là Extensions rồi s đó ta tạo 1 file .cs
* Extensions ta phải cho hàm đó là static 
* S đó t dk lại do là static method nên việc dk cũng khá dễ 

1. **FILE cấu hình trong ASP.net Core**

* **H**ệ thống cấu hình đã thay đổi rất nhiều trong ASP.net Code mới
* **Vs ASP.NET cũ ta sử dụng Web.config**
* **Còn ASP.NET CORE mới này cung cấp thẳng API để ta làm việc Cấu hình hổ trợ nhiều định dạng như Json,XML,INI, Command line ,Environment varibles (các biến môi trường cấu hình từ window) khuyên dùng để trong appsetting là 1 file Json**
* **Định dạng phổ biến là Json ,lưu trong json dùng Configuration để đọc json**
* **Loading the Configuration** 

1. **Static Files in ASP.NET CORE**

* **ASP.NET CORE** cho phép ta duyệt các file Tĩnh ,File tĩnh là những File nội dung không thay đổi vd HTML,CSS , JS.Images cho phép ta k cần di qua cả Request pepiline mà chỉ cần di qua Middle ware Static
* **Wwwroot(Web root:** là thư mục gốc của ứng dụng web là thư mục để file tĩnh **)**: là thư mục mặc định chứa các file tĩnh và khi ta bỏ các file tĩnh vào trong root tức là bạn đang public cho mọi người có thể truy cập dc
* **Vấn đề bảo mật:** nếu ta muốn bảo mật 1 file ta phải đi qua cơ chế Controller Action tức là trả về từ Action Controller

1. **MVC Design Pattern**
2. **Design Pattern là gì ?**

* **Design Pattern là :** là giải pháp tổng thể dc tối ưu hóa , dc tái sử dụng cho các vấn đề phổ biến trong thiết kế phần mềm má chúng ta thường gặp phải thằng ngày
* **Design Pattern Có Tác Dụng Gì ?**

Những lập trình viên có thể áp dụng giải pháp này để giải quyết các vấn đề tương tự. Các vấn đề bạn có thể sẽ nghĩ ra các giải quyết như nó chưa phải tối ưu nhất

Bạn cần phải hiểu rõ nó không phải là ngôn ngữ cụ thể nào cả. Design patterns có thể thực hiện được ở phần lớn các ngôn ngữ lập trình. Nó giúp bạn giải quyết vấn đề một cách tối ưu nhất, cung cấp cho bạn các giải pháp trong lập trình hướng đối tượng (OOP).

## **Tại sao phải sử dụng Design Pattern?**

Giúp sản phẩm của chúng ta linh hoạt, dễ dàng thay đổi và bảo trì hơn.

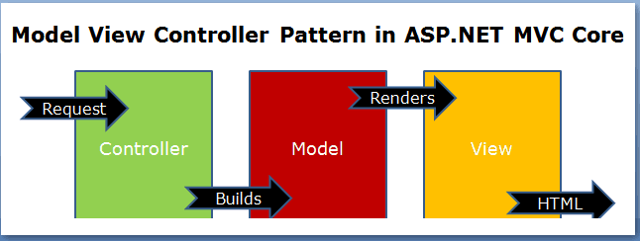
Có một điều luôn xảy ra trong phát triển phần mềm, đó là sự thay đổi về yêu cầu. Lúc này hệ thống phình to, các tính năng mới được thêm vào trong khi performance cần được tối ưu hơn.

Design pattern cung cấp những giải pháp đã được tối ưu hóa, đã được kiểm chứng để giải quyết các vấn đề trong software engineering. Các giải pháp ở dạng tổng quát, giúp tăng tốc độ phát triển phần mềm bằng cách đưa ra các mô hình test, mô hình phát triển đã qua kiểm nghiệm.

Những lúc khi bạn gặp bất kỳ khó khăn đối với những vấn đề đã được giải quyết rồi, design patterns là hướng đi giúp bạn giải quyết vấn đề thay vì tự tìm kiếm giải pháp tốn kém thời gian.

Giúp cho các lập trình viên có thể hiểu code của người khác một cách nhanh chóng (có thể hiểu là các mối quan hệ giữa các module chẳng hạn). Mọi thành viên trong team có thể dễ dàng trao đổi với nhau để cùng xây dựng dự án mà không tốn nhiều thời gian.

1. **MVC Design Pattern là gì?**

* **Khái Niệm:** Trong ASP.NET Core MVC dc áp dụng triệt để MVC viết tắt của Model View Controller , MVC là 1 partter chia để trị mỗi thành phần điều có nv riêng k ảnh hướng đến các phần khác (phụ thuộc vào nhau quá nhiều)
* **Model :** là chứa những dữ liệu cần có hiển thị cho khách hang đơn giản là những class c# bt gồm tt pt dùng để chuyên chỡ dữ liệu , nắm bắt dữ liệu
* **View :**thành phần giúp cho tạo ra giao diện cho người dùng và nhận những request từ người dùng rồi đẩy cho Controller xử lý vd hiển thị html , css,..
* **Controller:** là 1 thành phần chính nó đứng giữa view và Model controller sẽ nhận request thông qua view sao đó xử lý
* **Luồng đi của MVC như sao** 

1. **Basic Controller (controller có 3 nv chính )**

* Handle(xử lý) the request
* Build(xây dựng) a model
* Send(Gửi) the Response

**Action method** có những điều cần nhớ sao:

* Action method phải là một public method
* Action không thể là một Static method và một extension method
* Contructor getter or setter không dc sử dụng
* Inherited(kết thừa) method k dc sử dụng cho action method
* Action không thể chứa tham số ref và out
* Action không thể chứa attribute(NonAction)
* Action không dc overloaded

**Routing (cơ chế routing)**

* Là 1 phần rất quạng trong ASP.NET CORE
* Có 1 phần riêng về routing Hiểu nom na là nó sẽ quyết định action nào xử lý request đến

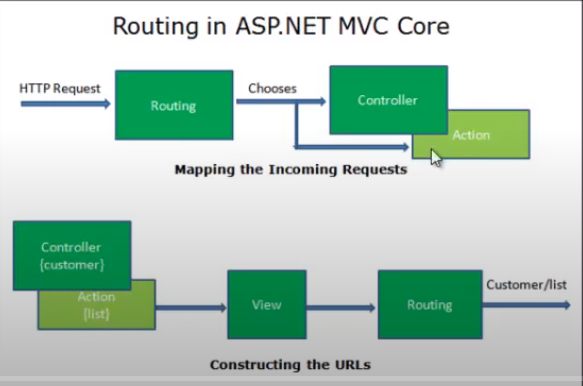
**Return Types (các kiểu trả về)**

* ViewResult trả về trang HTML
* EmptyResult trả vê Empty
* RedirectReult chuyển hướng đến 1 URL mới
* JsonResult trả về json Object
* javaScroptResult trả về Java script
* ContentResult trả về 1 bản text
* FileContentResult trả về 1 file có thể download dc (với dạng nhị phân)
* FilePathResult trả về 1 file có thể download dc (dạng đường dẫn)
* FileStreamResult trả về 1 file stream

1. **Routing mechanism(cơ chế)**

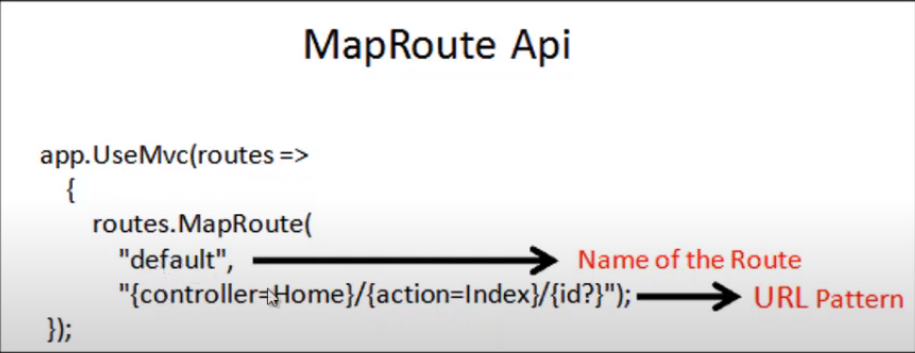
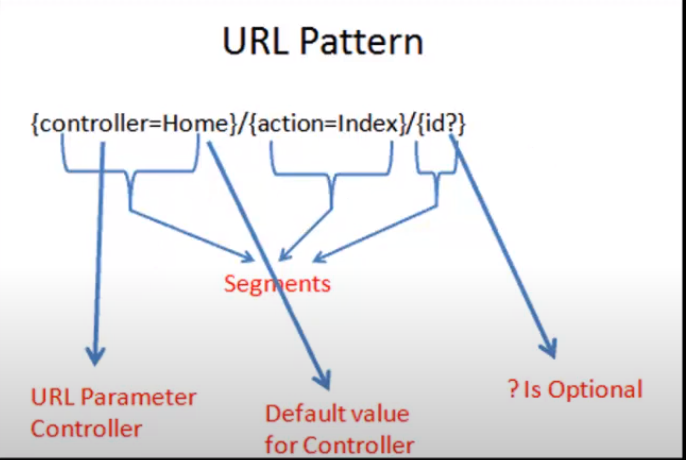
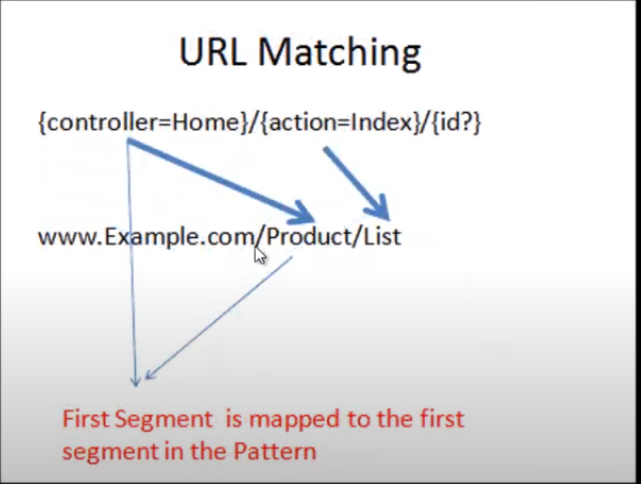
* **Một trong các thành phần quan trọng nhất của kiến trúc MVC là cơ chế routing (định tuyến). Nó là cơ chế quyết định xem Controller nào sẽ được xử lý request nào**
* **Routing là một quá trình khi ASP.NET Core xem xét các URL request gửi đến và "chỉ đường" cho nó đến Controller Actions. Nó cũng được sử dụng để tạo ra URL đầu ra. Quá trình này được đảm nhiệm bởi Routing Middleware. Routing Middleware có sẵn trong thư viện Microsoft.AspNetCore.Routing**
* Routing có 2 trách nhiệm chính:

Nv1: là maps cái yêu cầu cho Controller

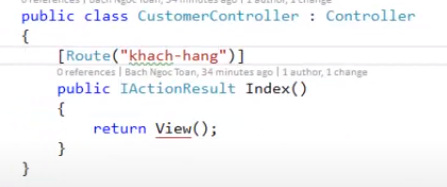
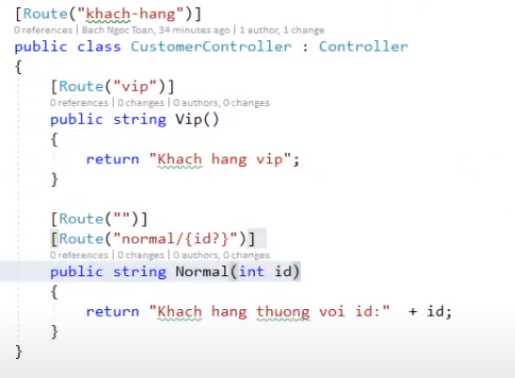
Nv2: Generate(tạo ra) một URL đầu ra correspond to controller action (tương ứng vs controller action) 

* **Routing làm việc ra sao** :
* Đầu tiên nó sẽ nhận HTTP request và Parse(phân tích) URLs
* Tiếp đến nó xem coi có cái matching(phù hợp) Route trong Routes Collection không
* Nếu có chạy đến RouteHandler
* Nếu không thì chạy tiếp Middleware kế ,trong trường hợp k có middleware nào phù hợp thì nó sẽ trả ra trang 404
* **Route là gì**
* Đơn giản là một roadmap(lộ trình) để đi đến đích
* ASP.NET CORE APP sử dụng Route đi đến Controller action
* The Each(Mỗi) Route contains(chứa đựng) một name , URL Pattern(Template)
* Mỗi Route bao gồm các thông tin như tên, mẫu URL (URL pattern) hay còn gọi là template url, thông tin controller action mặc định và ràng buộc (constraints). URL pattern được so sánh với URL đến xem có đúng mẫu không. Một ví dụ của URL pattern là: {controller=Home}/{action=Index}/{id?}
* Route được định nghĩa trong Microsoft.AspNetCore.Routing.

## **Route Collection là gì?**

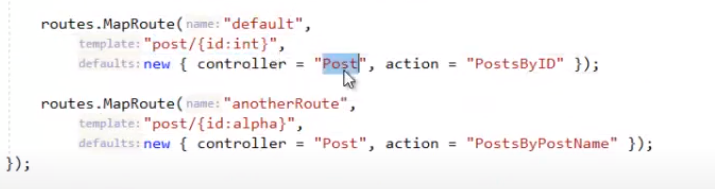
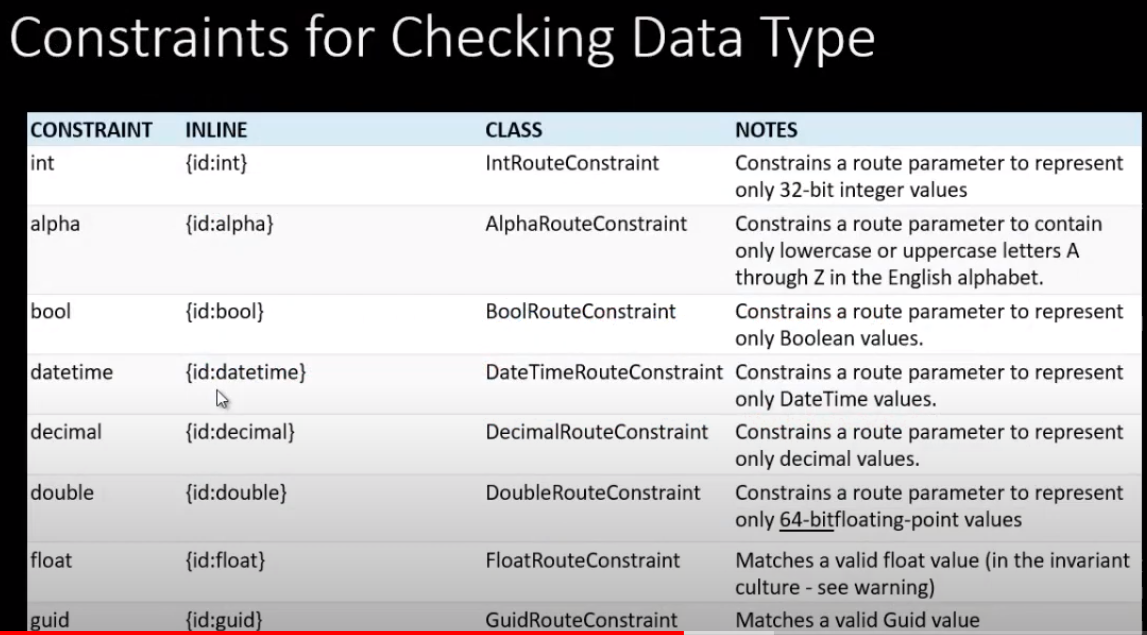
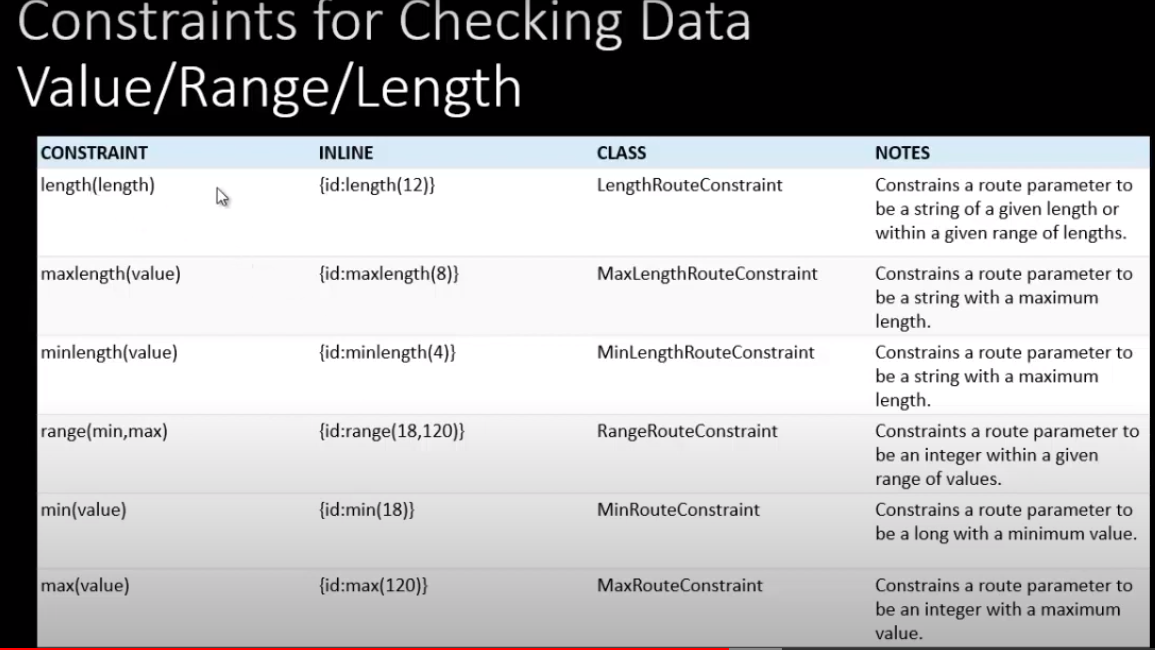
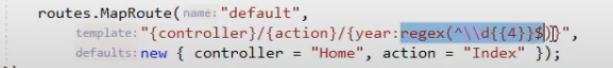
* Route Collection là một tập hợp tất cả ác Route trong ứng dụng. Một ứng dụng sẽ lưu một tập hợp các route ở một nơi duy nhất trong bộ nhớ. Các Route này sẽ thêm vào collection khi ứng dụng khởi động. Route Module sẽ tìm kiếm một Route match với URL request đến trong mỗi một Route của Route Collection. Route Collection được định nghĩa trong **Microsoft.AspNetCore.Routing.**
* **Route Handler là gì?**
* Route Handler là một thành phần quyết định sẽ làm gì với Route. Khi cơ chế routing tìm được một Route thích hợp cho một request đến, nó sẽ gọi đến **RouteHandler**và gửi route đó cho **RouteHandler**xử lý. Route Handler là class triển khai từ interface **IRouteHandler**. Trong ASP.NET Core thì Route được xử lý bởi **MvcRouteHandler**.
* **MVCRouteHandler**
* Đây là Route Handler mặc định của ASP.NET Core MVC Middleware. **MVCRouteHandler**được đăng ký khi đăng ký MVC Middleware. Bạn có thể ghi đè việc này bằng cách tự tạo cho mình một custom implementation của Route Handler.
* MVCRouteHandler được định nghĩa trong namespace: **Microsoft.AspnetCore.Mvc**
* MVCRouteHandler có trách nhiệm gọi Controller Factory, sau đó nó sẽ tạo ra một thể hiện của Controller được ghi trong Route. Controller sẽ được nhận và nó sẽ gọi một Action Memthod và tạo ra View. Vậy là hoàn thành request.
* **URL Pattern**
* Mỗi Route phải chứa một URL Pattern. Pattern này sẽ được so sánh với URL requét. Nếu pattern đúng với URL thì nó sẽ được sử dụng bởi hệ thống routing để xử lý URL đó. Mỗi một URL Pattern bao gồm một hoặc nhiều phần. Các phần chia tách bởi dấu gạch chéo.
* Mỗi phần có thể là một hằng số (constant) hoặc một Route Parameter.
* Route Parameter được bao gọc bởi một cặp dấu ngoặc nhọn ví dụ {controller}, {action}.
* Route Parameter có thể có giá trị mặc định như {controller=Home} khi Home là giá trị mặc định của controller. Một dấu = sẽ gán giá trị cho tên parameter.
* Bạn có thể có một thành phần dạng hằng số. Ví dụ: admin/{controller=Home}/{action=Index}/{id?}. Ở đây thì "admin" là một hằng tức là một chuỗi cố định phải tồn tại trên URL.
* Dấu ? trong {id?} chỉ ra là tham số này không bắt buộc. Một dấu ? sau tên tham số chỉ ra tham số đó không yêu cầu phải có giá trị.
* URL Pattern {controller=Home}/{action=Index}/{id?}. Đăng ký một route có thành phần đầu tiên trên URL là một controller, phần thứ 2 là Action method trong controller đó. Và phần cuối là dữ liệu thêm vào tên là id.
* URL Matching
* Mỗi phần trong URL request đến sẽ match tương ứng với thành phần của URL Pattern. Route {controller=Home}/{action=Index}/{id?} có 3 thành phần. Phần cuối là tùy chọn
* 
* 
* URL MATCHING

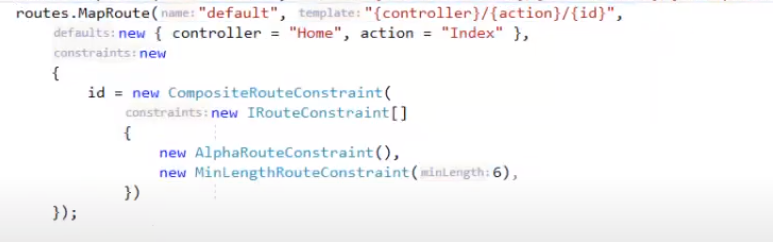
# Cơ chế Attribute Based Routing

* Là cơ chế định nghĩa routing trực tiếp trong Controller Action thay vì trong Startup.cs
* Các routing dc cấu hình trong strartup.cs sẽ không trực quan
* Vd: 
* Ta có thể thiết kế ra nhiều Route khác nhau
* Vd: ta thiết lập Routing chung cho class Customer thay vì ta ghi nhiều Routing khach-hang cho nhiều route bên dưới thì ta thiết lập chung 1 routing cho cả class lúc này khi ta chạy trang web mà k gõ route “Khach-hang” thì nó sẽ chẳng hiện route Normal mặc dù nó đang để route (“”)

1. **Route Constraints(Ràng buộc)**

Route Constrains giúp chúng ta lọc và giới hạn các giá trị không mong muốn truyền vào controller action. Nó được kiểm tra bởi các ràng buộc áp dụng cho giá trị truyền vào URL Parameter.

* Đầu tiên ta xem vd về rang buộc route  ở chỗ id:int? Ta có :int đây chính là cách ta ràng buộc route ở đây dc hiểu là khi ta truyền id vào index thì nó chỉ nhận là kiểu value là int còn nếu value là kiểu khác vd như string thì nó để giá trị mặc định đây ra cách ra tường minh hơn cách trên như khá dài dòng
* Vd thứ 2: ta sẽ thiết lập controller và sử dụng action method để test ta tạo 2 route có controller là Post Nhưng khác Action 1 cái sẽ có kiểu ràng buộc là int:int(nhận giá trị số) action là PostByID, còn cái kia sẽ nhận vào int:alpha(là chữ) action là PostByPostName tiếp đến ta tạo controller  khi ta chạy nếu id = số nó sẽ chạy vào hàm PostByID và ngược lại sẽ chạy PostByPostName và nhận tham số tương ứng
* 
* 
* 1 vd về rang buộc vd này sẽ làm cho bắt buộc giá trị id phải là số và phải là 4 số ,if id=1số,2 số điều k dc 
* Ta có thể kết hợp nhiều cách rang buộc 

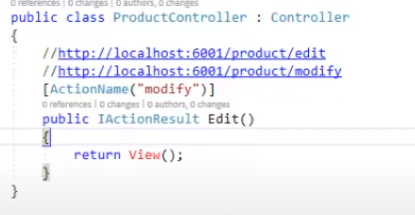
ở đây vừa là chữ vừa chỉ dc 6 ký tự (6 ký tự chỉ là chữ nó mới nhận) vd này chỉ là cách ta viết tường minh thôi chứ tác dụng k khác j vd trên

1. **Action Seletor(Bộ chọn) và Action Verbs(những động từ)**

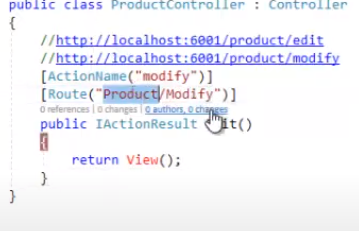
**Action Selector types**

* **Action Name** **:** tức là ta muốn thay đổi name của URL khác k muốn nó giống vs action method

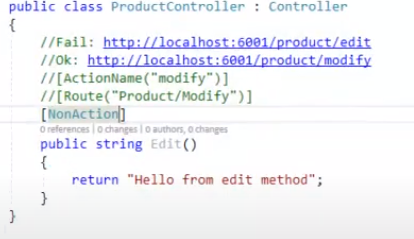
**Cách 1:** ta tạo một controller đặt tên là Product trong product có action method là Edit vậy đường dẫn mặc định là (localhost:5555/product/edit) lúc này ta k muốn để thành edit mà muốn để là modify thì ta sử dụng acctionName để làm việc này

 khi đó dg dẫn sẽ khác đi

**Cách 2: ta dùng route**



* **Non(không) Action:**là đánh dấu(một attiubiu đánh dấu) khi mà method k thỏa mản dk là public k muốn public ra ngoài(tức là tuy nó là public action như nó k dc chia sẽ bất kỳ đâu)

cho dù có gọi cũng fail

* **Action Verbs:** gồm các push ,post, get ,delete dùng trong API

1. **Action Results**
2. **Khái Niệm Action Result:**thì trong action method Client có thể nhận về rất nhìu dử liệu khác nhau có thể là 1 chuỗi json, hay 1 kiểu string, hoặc view html, có đôi khi là 1 file cho người dùng dowload về

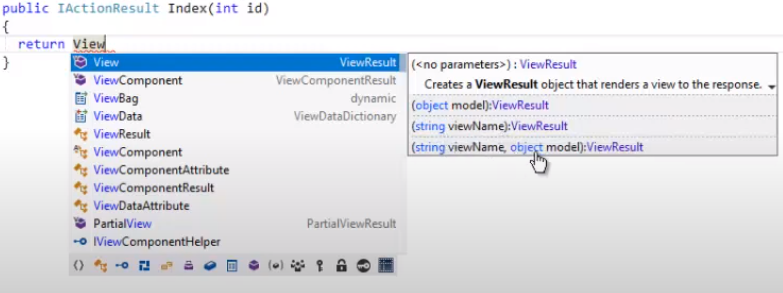
Controller trong ASP.NET Core chỉ đơn giản là các class C#. Nó không cần phải kế thừa từ bất cứ base class nào. Nhưng ASP.NET Core cung cấp một class Controller nó kế thừa từ một ControllerBase class. Điều này giúp class cung cấp rất nhiều các method hữu ích, nó giúp Controller làm việc dễ dàng hơn. Vì thế thông thường các controller của chúng ta đều kế thừa từ Controller class.

Controller Base class triển khai các loại Action Result khác nhau sẵn có giúp xây dựng các loại kết quả trả về khác nhau cho client. Ví dụ, ViewResult trả về một HTML response. Một RedirectResult chuyển hướng đến URL khác. Content Result trả về một chuỗi văn bản. Các kiểu trả về này được biết đến là Action Result.

1. **Phân biệt IActionResult và ActionResult**

* IActionResult là 1 interface và ta sử dụng Iaction hay actionresult trả về điều đc bởi vì acctionresult cũng kế thừa từ IactionResult
* Cho nên actionResult có nhiều con vd như: viewResult , JsonResult,FileResult,…
* Tại sao lại dùng nó: thì nếu ta k dùng ActionResult thì controller của chúng ta tự làm hết
* IActionResult là một interface nó định nghĩa một khuôn mẫu cho toàn bộ các Action Result của một action method. ActionResult là một abstract base class triển khai interface IActionResult. Action result như ViewResult, PartialViewResult hay JsonResult...đều kế thừa từ ActionResult base class.

1. **Action Result Sets(bộ)**

* **Producing(sản xuất) the HTML**
* **ViewResult** 
* **PartialViewResult(Xem 1 phần kết quả):** PartialView Result sử dụng model để tạo ra một phần của View. Chúng ta sử dụng ViewResult để tạo ra một view hoàn chỉnh còn PartialView trả về một phần của View. Kiểu trả về này hữu ích với Single Page Application (SPA) ki bạn muốn cập nhật một phần của View thông qua AJAX.
* **Redirecting(chuyển hướng) the users**
* Redriect result được dùng khi bạn muốn chuyển hướng người dùng đến một URL khác. Có 4 loại redirect result có sẵn. RedirectResult, LocalRedirectResult, RedirectToActionResult và RedirectToRouteResult.
* Mỗi một redirect này có thể trả về bất cứ mã trạng thái (status code) dưới đây:

[302 Found (Chuyển tạp thời)](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/302)

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/302>

[301 Moved Permanently](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/301)

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/301>

[307 Temporary Redirect](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/307)

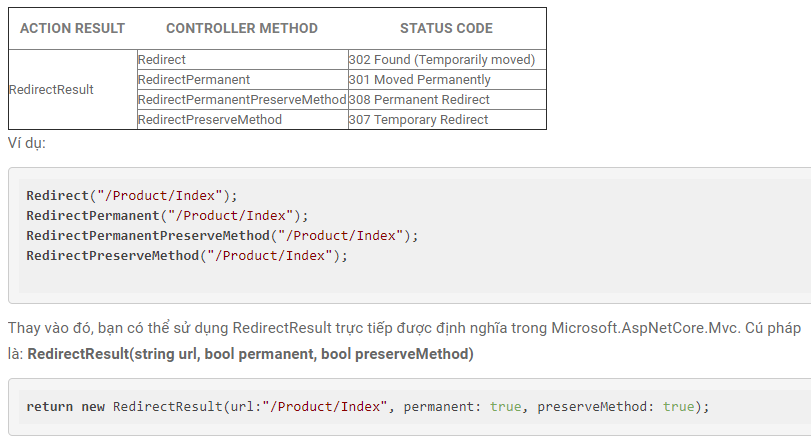
<https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/307>

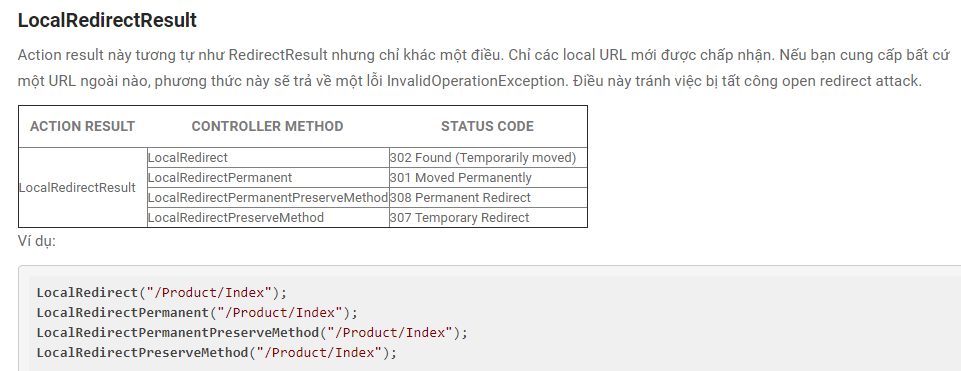
[308 Permanent Redirect](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/308)

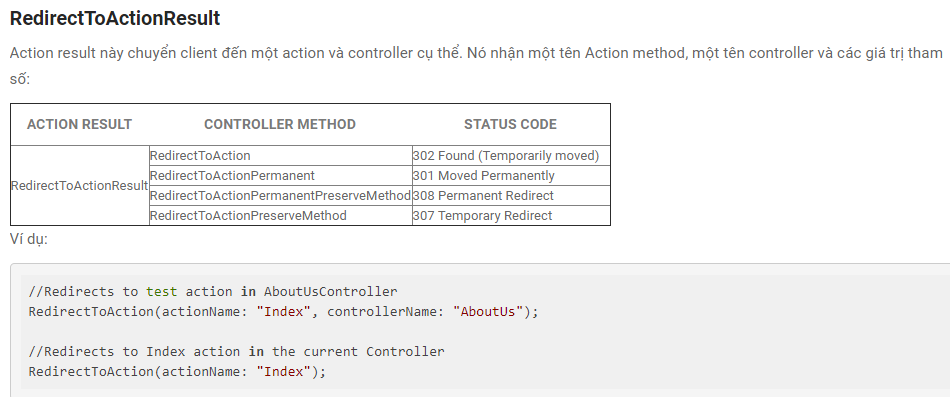
<https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/308>

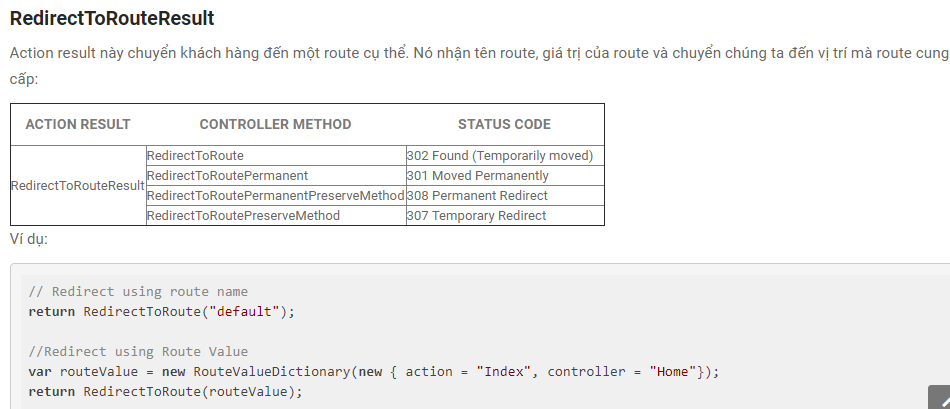
### RedirectResult

### RedirectResult sẽ trả về cho user bằng cách cung cấp đường dẫn tuyệt đối hoặc tương đối:









* **Returning(Trả về )Files**

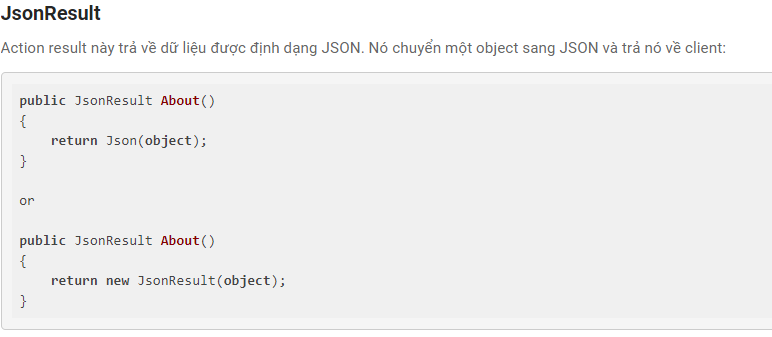
FileResult là một Action result sử dụng bởi Controller action để trả về file cho người dùng.

* **FildeResult**

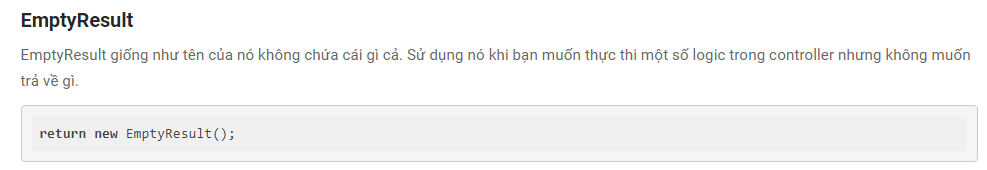




* **Content(nội dung) Result**







* **Returning Error and HTTP codes**

Loại Action result này được dùng trong Web API Controller. Kết quả sẽ được gửi về kèm HTTP Status Code. Một trong số chúng thì có thể gửi một đối tượng vào response.

### StatusCodeResult

StatusCodeResult gửi kết quả và chỉ ra một HTTP Status code:

**return** StatusCode(200);

hoặc

**return** **new** **StatusCodeResult**(200);

### ObjectResult

Action result này sẽ trả về một đối tượng kèm một HTTP Status Code là 200. Nó là một overload của method StatusCode

**return** StatusCode(200, **new** { message = "Hello" });

hoặc

**return** **new** **ObjectResult**(**new** { message = "Hello" });

### OkResult

Action result này trả về nguyên chỉ có HTTP Status code 200:

**return** Ok();

hoặc

**return** **new** **OkResult**();

### OkObjectResult

Action result này trả về một HTTP Status code 200:

**return** Ok(**new** {message="Hello"});

hoặc

**return** **new** **OkObjectResult**(**new** { message = "Not found" });

### CreatedResult

CreatedResult sử dụng khi một tài nguyên được tạo ra sau request Post. Nó gửi trạng thái 201 về kèm đối tượng vừa được tạo:

**return** Created(**new** Uri("/Home/Index", UriKind.Relative), **new** {message="Hello World"});

hoặc

**return** **new** CreatedResult(**new** Uri("/Home/Index", UriKind.Relative), **new** { message = "Hello World" });

### CreatedAtActionResult

Cái này tương tự CreatedResult nhưng nó nhận vào Controller và Action thay vì URL:

**return** CreatedAtAction("Index", **new** {message="Hello World"});

### CreateAtRouteResult

Action Result này nhận vào gái trị route và tương tự như CreatedResult và CreatedAtActionResult

CreatedAtRoute("default", **new** { mesage = "Hello World" });

### BadRequestResult

Action result này gửi về một HTTP Status code 400 cho client. Sử dụng response status code này khi chỉ ra cú pháp không đúng hoặc một request không được rõ ràng.

return BadRequest()

### BadRequestObjectResult

Action result này tương tự BadRequestResult. Khác nhau là bạn có thể gửi về một ModelStateDictionary (chứa chi tiết lỗi) và cũng là status 400:

**var** modelState = **new** ModelStateDictionary();

modelState.AddModelError("message", "errors found. Please rectify before continuing");

**return** BadRequest(modelState);

Phương thức BadRequest có một overload thứ 2, trả về một BadRequestObjectResult

### NotFoundResult

Action result này trả về lỗi HTTP 404 cho client:

return NotFound();

### NotFoundObjectResult

Action result này tương tự nuhw NotFoundResult nhưng trả về một đuối tượng kèm lỗi 404. Overload thứ 2 của NotFound giúp nhận một đối tượng làm tham số để trả về NotFoundObjectResult.

**return** NotFound( **new** { message = "Not found" });

### UnsupportedMediaTypeResult

Action result này gửi về lỗi HTTP 415. Sử dụng action result này khi request với định dạng không được hỗ trợ bởi server.

**return** **new** UnsupportedMediaTypeResult();

### NoContentResult

Action result này gửi lỗi HTTP 204 về. Sử dụng NoContentResult này khi request thành công nhưng không có nội dung nào được trả về

return NoContent();

* **Security(bảo vệ) Related(Liên Quan) Results**

### **SignInResult**

SignInResult là kết quả của hành động đăng nhập. SignInManager.SignInAsync hoặc PasswordSignInAsync trả về một SignInResult. Nó có 4 thuộc tính là Succeeded, IsLockedOut, IsNotAllowed và RequiresTwoFactor

Bạn có thể tham khảo về ASP.NET Identity Core.

### **SignOutResult**

SignOutResult là kết quả của hành động logout

### **ForbitResult**

ForbitResult trả về lỗi 403 tức là người dùng không được cấp quyền để thực hiện một hành động nào đó trên tài nguyên nào đó. ForbitResult không có nghĩa là người dùng chưa chứng thực. Người dùng chưa chứng thực nên trả về ChallengeResult hoặc UnAuthorisedResult.

**return** **new** ForbidResult();

hoặc

**return** **Forbid**();

Forbit là phương thức của Controller base class trả về thể hiện của ForbidResult. Thay vào đó bạn có thể trả về Status Code:

return StatusCode(403);

### **ChallengeResult**

ChallengeResult trả về khi chứng thực thất bại. Kết quả sẽ không gọi đến baatscuws middleware nào để tạo response. Ví dụ trả về lỗi 401 (Unauthorized) hoặc 403 (Forbidden) hoặc chuyển hướng người dùng đến trang đăng nhập.

### **UnauthorizedResult**

UnauthorizedResult trả về lỗi “401 – Unauthorized”. Controller sử dụng phương thức Unauthorized để trả về thể hiện của UnauthorizedResult.

**return** Unauthorized();

hoặc

**return** **new** **UnauthorizedResult**();

Khác nhau giữa UnauthorizedResult và ChallengeResult là một thằng trả về Status Code còn 1 thằng là không làm gì với nó.

1. **View**

## **View là gì?**

View có trách nhiệm tạo ra giao diện cho người dùng từ model. Controller trong ASP.NET Core sẽ nhận request sau đó thực thi với logic tương ứng với dữ liệu đầu vào từ request. Sau đó nó trả về Model cho View.

## **Trách nhiệm của View**

Render ra giao diện và hiển thị model lên là trách nhiệm của View. View không nên chứa bất cứ logic nào và không được xử lý logic. View có thể sử dụng bất cứ định dạng nào để trả về cho user. Định dạng có thể là HTML, JSON, XML hay là bất cứ định dạng nào khác.

## Ví dụ

Mở project sau đó tạo một ứng dụng ASP.NET Core. Project có chứa một Controller và một Index view.

Tham khảo: [Xây dựng ứng dụng ASP.NET Core MVC đầu tiên](https://tedu.com.vn/lap-trinh-aspnet-core/xay-dung-ung-dung-aspnet-core-mvc-dau-tien-225.html)

## **Làm sao để tạo View**

Project ví dụ ở trên đã có một Index Action method. Giờ chúng ta sẽ tạo một Action method khác:

**public** IActionResult **About**()

{

**return** View();

}

Các method trong Controller trả về một Action **Result**(hoặc một class kế thừa từ ActionResult). Nên tham khảo: [Action Result trong ASP.NET Core](https://tedu.com.vn/lap-trinh-aspnet-core/action-result-trong-aspnet-core-237.html)

ActionResult trả về HTML là một **ViewResult**. Phương thức View của Controller base class trả về một **ViewResult**. Chuột phải vào bất cứ đâu trong phương thức và chọn**Add View**

### https://tedu.com.vn/uploaded/images/072019/Adding-View-in-ASP.NET-MVC-Core.png

Sau đó sẽ thấy cửa sổ Add MVC View như sau:

### https://tedu.com.vn/uploaded/images/072019/Add-View-Dialog.png

### **Tên View**

Nhập tên View tại đây. Theo quy tắc, tên phải cùng tên với Action Method name. Bạn có thể ghi đè phương thức này bằng cách đặt tên khác đi. Nếu bạn làm điều này thì bạn sẽ cần phải chỉ ra tên của View trong tham số đầu tiên của phương thức View: View(“ViewName”)

#### **Template**

Template có vài tùy chọn như Create, Delete, Details, Edit, List, Empty (without Model). Các template ở trên cần một model, ngoại trừ Empty Template. Visual Studio tạo ra một view để create, edit, delete...trên model được cung cấp phụ thuộc vào template mà bạn chọn. Mỗi Template trên đều có giải thích phía dưới.

#### **Create**

Tạo một HTML Form để tạo mới một model. Nó tạo ra một label và một input cho mỗi thuộc tính trong model.

#### **Delete**

Tạo một HTML Form cho việc xóa model. Nó tạo ra một label và giá trị heienj tại cho mỗi thuộc tính của model.

#### **Details**

Tạo mới một view để hiển thị model. Nó tạo ra một label và giá trị của mỗi thuộc tính trong model.

#### **Edit**

Nó tạo ra một HTML form để sửa model. Nó tạo ra một label và một input cho mỗi thuộc tính của model.

#### **Empty**

Tạo ra một view trống

#### **List**

Tạo ra một HTML Table hiển thị danh sách model. Nó tạo ra một cột cho mỗi thuộc tính của Model. Bạn cần trả về một IEnumerable<Model> cho View. View cũng chứa danh sách các hành động để thực hiện như create/edit/delete.

### **Model class**

Dropdown hiển thị tất cả các Model class trong project. Tùy chọn này được loại bỏ nếu bạn chọn Empty project.

### **Tạo một Partial View**

Tùy chọn này là Create a Partial View. Partial View tạo một phần của view và không phải view hoàn chỉnh. PartialViewResult sử dụng model để tạo ra một phần của View. Kết quả của partial view nhìn giống hư một view bình thường ngoại trừ nó không có thẻ <html> hoặc không có thẻ <head> ở trên của View. Chọn tùy chọn này chỉ ra view cảu bạn sẽ tạo không phải view đầy đủ vì thế tùy chọn Layout sẽ bị bỏ đi.

### **Thư viện Scripts**

Chọn tùy chọn này sẽ thêm jquery.validate.min.js và jquery.validate.unobtrusive.min.json vào thưu viện JavaScript. Các thư viện này cần thiết cho triển khai client-side validation. CÁc thư viện này đòi hỏi khi tạo view chứa một đầu vào cho dữ liệu như một Form Edit view hoặc một form Create.

### **Sử dụng Layout Page**

Tùy chọn này cho phép bạn chọn một Layout page cho View. Layout page được dùng để chia sẻ các thành phần dùng chung trong trang của bạn và cung cấp một giao diện đồng nhất trong toàn bộ hệ thống.

Ví dụ chọn tên view là About, và template là Empty (without model) không chọn vào Create a partial view và Use a layout page. View được tạo sẽ nằm trong thưu mục Views/Home/About.cshtml. Mở file About.cshtml ra và thêm dòng <h1>About Us</h1> vào sau thẻ tiêu đề:

@{

Layout = null;

}

<!DOCTYPE html>

<**html**>

<**head**>

<**meta** name="viewport" content="width=device-width" />

<**title**>About</**title**>

<**h1**>About Us</**h1**>

</**head**>

<**body**>

</**body**>

</**html**>

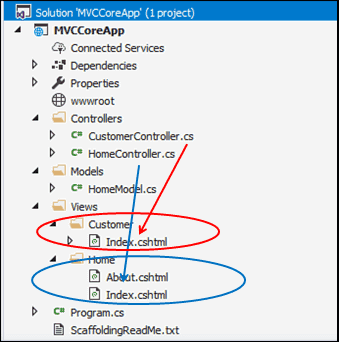
Chạy ứng dụng và bạn nhìn thấy dòng chữ "About us" trên ứng dụng.

## **Vị trí của View**

Theo quy tắc thì tất cả View được lưu trong thư mục Views của ứng dụng. Mỗi Controller sẽ có một thư mục con trong thư mục Views với tên trùng tên controller nhưng sẽ không có hậu tố Controller. Vì thế sẽ có một **HomeController**thư mục tên Home trong thư mục Views.

Mỗi Action method trong Controller sẽ lấy một file cho nó, tên trùng với tên của Action method. Vì thế phương thức Index của HomeController có một file với tên Index.cshtml trong Views/Home.

Dưới đây là hình minh họa:



Mỗi Action method trong **HomeController**sẽ có một file View tương ứng như **About.cshtml** và **Index.cshtml** và lưu trong thưc mục **Views/Home**. Tương tự như thế, action method của **CustomerController**sẽ có thư mục **Views/Customer**.

## **ASP.NET Core tìm View như thế nào?**

Project Example có một Controller tên là **HomeController**. Giờ hãy thêm **CustomerController**vào. Chọn thư mục **Controllers**chuột phải và add thêm Controller. Tên của nó là **CustomerController**.

**using** Microsoft.AspNetCore.Mvc;

**namespace** **MVCCoreApp.Controllers**

{

**public** **class** **CustomerController** : **Controller**

{

**public** IActionResult **Index**()

{

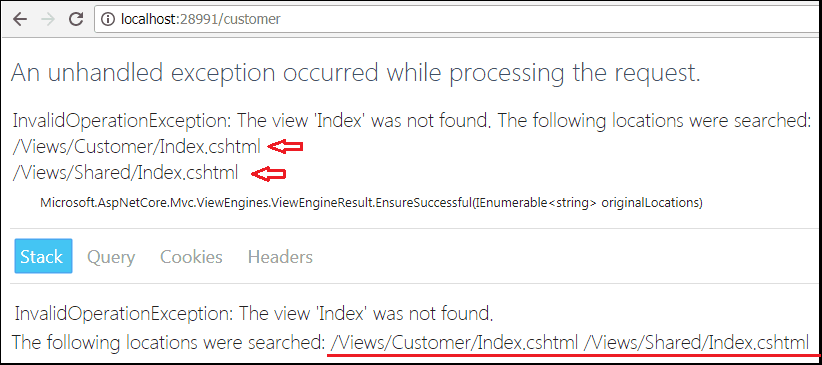
**return** View();

}

}

}

Giờ chúng ta không tạo View thay vào đó chạy luôn URL /Customer. Chúng ta nhìn lỗi như sau:



Màn hình ở trên hiển thị chính xác lỗi khi ASP.NET Core MVC middleware tìm kiếm View trong 2 thư mục:

* /Views/Customer/Index.cshtml
* /Views/Shared/Index.cshtml

Đầu tiên là thư mục Customer với tên của controller. Nếu không thấy nó sẽ tìm tiếp trong thư mục Shared trước khi đưa ra lỗi InvalidOperationException.

Thư mục Shared là thư mục đặc biệt chứa Views, Layouts hoặc Partial view dùng chung cho nhiều view.

Mở CustomerController và click bất cứ chỗ nào trong Index method. Chuột phải chọn Create an Empty View. File Index.cshtml sẽ tự động tạo ra trong thư mục Views/Customer. Giờ hãy mở Index.cshtml:

@{

Layout = null;

}

<!DOCTYPE html>

<**html**>

<**head**>

<**meta** name="viewport" content="width=device-width" />

<**title**>Index</**title**>

<**h1**>Customer List</**h1**>

</**head**>

<**body**>

</**body**>

</**html**>

Giờ hãy chạy ứng dụng và chúng ta không thấy lỗi này nữa.

1. **Razor**

# Razor View Engine trong ASP.NET Core MVC

## Razor View Engine là gì?

Razor View Engine là View Engine mặc định của ASP.NET Core. Nó lấy mã Razor trong file View và chuyển sang HTML response.

## Razor Markup

Controller trong MVC gọi View bằng cách gán dữ liệu để tạo giao diện. View phải có khả năng xử lý dữ liệu và tạo response. Điều này được xử lý bằng cách dùng Razor, nó cho chúng ta sử dụng C# code trong file HTML. Razor View Engine xử lý các lệnh này và tạo ra HTML.

Các file chứa Razor có đuôi .cshtml. Cú pháp Razor thường ngắn hơn và đơn giản hơn cũng dễ học như C# hoặc VB. Visual Studio IntelliSense cũng hỗ trợ cú pháp Razor.

## Ví dụ

Để hiểu Razor làm việc ra sao? Hãy tạo Empty Project và thêm MVC Middleware vào. Bạn có thể tham khảo: [Xây dựng ứng dụng ASP.NET Core MVC đầu tiên](https://tedu.com.vn/lap-trinh-aspnet-core/xay-dung-ung-dung-aspnet-core-mvc-dau-tien-225.html). Bạn có thể [download mã nguồn](https://github.com/teduinternational/aspnetcore_samples) từ Github.

Xem Index method của HomeController:

**public** IActionResult **Index**()

{

**return** View();

}

Mở Index.cshtml và copy:

<**h1**>Razor Example</**h1**>

Chạy ứng dụng và bạn nhìn thấy ví dụ: Giờ bạn hãy mở thư mục Models ra và tạo một class Customer.cs

**namespace** **MVCCoreApp.Models**

{

**public** **class** **Customer**

{

**public** **string** name { **get**; **set**; }

**public** **string** address { **get**; **set**; }

}

}

## Cú pháp Razor

Razor sử dụng ký tự @ để chuyển HTML sang C#.  Nos cos 2 cacsh ddeer khai baso:

* Sử dụng Razor expression
* Sử dụng khối lệnh Razor

Các biểu thức này được xử lý bởi Razor View Engine và được gán vào response.

## Khối lệnh

Khối lệnh Razor bắt đầu bởi @ và nằm trong cặp ngoặc nhọn. Bạn có thể sử dụng khối lệnh bất cứ đâu trong file. Một khối Razor có thể sử dụng để thao tác Model, khai báo biến, đặt thuộc tính của View. Tuy nhiên nó không nên sử dụng cho việc xử lý logic.

Mở file Index.cshtml ra và copy đoạn này:

<h3>Code Block</h3>

@{

**var** greeting = "Welcome to our site!";

**var** weekDay = DateTime.Now.DayOfWeek;

}

@{

**var** cust = **new** MVCCoreApp.Models.Customer()

{

name = "Rahul Dravid",

address = "Bangalore"

};

}

Đầu tiên chúng ta tạo một khối lệnh Razor bắt đầu với @ và cặp {}. Trong cặp ngoặc này chúng ta có một đoạn C# thông thường với lệnh khai báo hai biến greeting và weekDay

## Biểu thức Razor

Biểu thức Razor bắt đầu với @ và theo sau là code C#. Biểu thức này có thể là ngầm định hoặc tường minh.

### Implicit Razor Expressions

Implicit Razor Expressions không cho phép dấu cách vì nó dùng để kết tuhcs biểu thức. Các biểu thức được xử lý bởi Razor View Engine và kết quả được thêm ngay vào vị trí nó đặt.

<**h3**>Code Expression</**h3**>

<**p**>@greeting</**p**>

<**p**>@DateTime.Now</**p**>

<**p**>Today is : @WeekDay thank you </**p**>

Đoạn Razor dưới đây hiển thị tên và địa chỉ trong thẻ <p>

<**p**>Name : @cust.name</**p**>

<**p**>Address : @cust.address</**p**>

### Explicit Razor Expressions

Explicit Razor Expressions bắt đầu bằng dấu @ và theo sau bởi cặp (). Bất cứ nội dung nào trong cặp ngoặc đơn này đều được xử lý bởi Razor và tạo ra output.

@{ var ISBNNo = "10001200"; }

<**p**>ISBN : @ISBNNo</**p**>

@\*//<**p**>ISBN@ISBNNo</**p**> //Does not work\*@

<**p**>ISBN@(ISBNNo)</**p**>

## Hỗ trợ nhận diện cú pháp

Hình dưới đây hiển thị cách Razor làm việc được sử dụng trong HTML và Intellisense hỗ trợ từ Visual Studio:

### https://tedu.com.vn/uploaded/images/072019/Intellisense-Support-for-Razor-in-Visual-Studio.png

### Sử dụng Directive

Khai báo @using làm việc tương tự như C# vậy. Nó cho phép chúng ta import namespace. Ứng dụng có thể import namespace MVCCoreApp.Models theo cách sau:

@**using** MVCCoreApp.Models;

Và bạn đơn giản là dùng nó var cust=new Customer() thay vì var cust = new MVCCoreApp.Models.Customer()

### Khai báo biến

Các biến được khai báo sử dụng từ khóa var hoặc sử dụng kiểu dữ liệu C#. CÁc kiểu như int, float, decimal, bool, DateTime hay string có thể được dùng để lưu trữ kiểu dữ liệu tương ứng. Để khai báo biến bạn vẫn có thể dùng dấu @ trực tiếp:

<**h3**>Variables </**h3**>

<!-- Storing a string -->

@{ var message = "Welcome to our website"; }

<!-- Storing a date -->

@{ DateTime date = DateTime.Now; }

<**p**>@message</**p**>

<**p**> The current date is @date</**p**>

Các chuỗi được bao trong một cặp nháy kép. Để sử dụng nháy kép trong chuỗi, bạn cần phải nhân đôi dấu nháy.

@{ var helloWorld = @"Hello ""World"""; }

<p>@helloWorld</p>

Tương tự như vậy thì dấu gạch chéo ngược cũng được làm như thế:

@{ **var** Path = @"C:\Windows\"; }<p>

The path **is**: @Path</p>

Bạn có thể in ký tự @ trong HTML bằng cách lặp 2 ký tự:

@{ var symbol = "You can print @ in html"; }

<p>The @@symbol is: @symbol</p>

Dấu @ trong email thì bạn có thể chỉ đích danh nó và không cần phải double:

<**a** href="mailto:admin@tektutorialsHub.com">admin@tektutorialsHub.com</**a**>

### Comment

Sử dụng @\*\*@ để đặt comment:

@\*This **is** comment\*@

### HTML trong khối lệnh

Bất cứ HTML Element nào trong khối lệnh Razor được nhận dạng như bình thường:

@{

<**p**>Hello from the Code block</**p**>

}

### Dòng lệnh đơn

Bạn có thể xuất ra giá trị mà không cần thẻ HTML bằng dấu @:

@{

@:Hello from the Code block

}

### Đa dòng lệnh

Đối với nhiều dòng bạn cần phỉa có thẻ <text></text>

@{

<**text**>Hello from the multiline text </**text**>

}

### Lệnh điều kiện

Razor cho phép xử lý lệnh điều kiện như if hoặc switch.

#### If else

Điều kiện if được dùng để tạo ra một nội dung dựa trên điều kiện đó:

@{**int** **value** = 200;}

@**if** (**value** > 100)

{

<p>Value **is** greater than 100.</p>

}

**else**

{

<p>Value **is** less than 100.</p>

}

Hoặc đoạn code:

@{

**var** **value** = 200;

**if** (**value** > 100)

{

<p>The **value** **is** greater than 100 </p>

}

**else**

{

<p>This **value** **is** less than 100.</p>

}

}

Và lệnh if else if:

@**if** (SomeCondition)

{

}

**else** **if**(SomeOtherCondition)

{

}

**else**

{

}

#### Switch

Một lệnh switch có thể thêm nội dung vào HTML trên một số điều kiện:

@**switch** (value)

{

**case** 0:

@: value is Zero

**break**;

**case** 100:

<**p**>Value is 100 </**p**>

**break**;

**case** 200:

<**p**>Value is @value </**p**>

**break**;

**case** 300:

<**text**>Value is 300</**text**>

**break**;

**default**:

<**p**>Invalid Value </**p**>

**break**;

}

### Vòng lặp

#### Lặp foreach

Vòng lặp được sử dụng để lặp khối lệnh

@for (**int** i = 0; i < 5; i++)

{

<span> @i </span>

}

Trường hợp tốt nhất là dùng lặp qua các tập hợp đối tượng và hiển thị danh sách  trong bảng:

@{

var custList = new List<**Customer**>()

{

new Customer() { name = "Rahul", address = "Bangalore" },

new Customer() { name = "Sachin", address = "Mumbai" }

};

}

<**table**>

<**thead**>

<**tr**><**td**>Name</**td**><**td**>Address</**td**></**tr**>

</**thead**>

@foreach (Customer custvar in custList)

{

<**tr**>

<**td**>@custvar.name</**td**>

<**td**>@custvar.address</**td**>

</**tr**>

}

</**table**>

#### Vòng lặp While

<h3>While **loop**</h3>

@{

var r = 0;

**while** (r < 5)

{

r += 1;

<span> @r</span>

}

}

@{ var s = 0; }

@while (s < 5)

{

s += 1;

<span> @s</span>

}

}

### Mã hóa HTML

Tất cả biểu thức Razor được tự động mã hóa HTML

Ví dụ:

@{

string encodedMessage = "<**script**>alert('You are under cross-site script injection attack');</**script**>";

}

<**span**>@encodedMessage </**span**>

Đoạn code trên sẽ không có kết quả ra một alert mà thay vào đó bạn sẽ thấy dòng chữ: “<script>alert(‘You are under cross-site script injection attack’);</script>” trên trình duyệt.

Razor sẽ tự động mã hóa < thành &lt; và > thành &gt;

Nếu bạn nhìn thấy HTML Response sẽ như sau:

<span>&**lt**;script&**gt**;alert(&#x27; You are under cross-site script injection attack&#x27;) ;&lt;/script&gt; </span>

Nếu bạn muốn script thực thi, bạn bạn cần sử dụng @Html.Raw để int chuỗi không mã hóa ra.

<**span**>@**Html**.**Raw**(**encodedMessage**)</**span**>

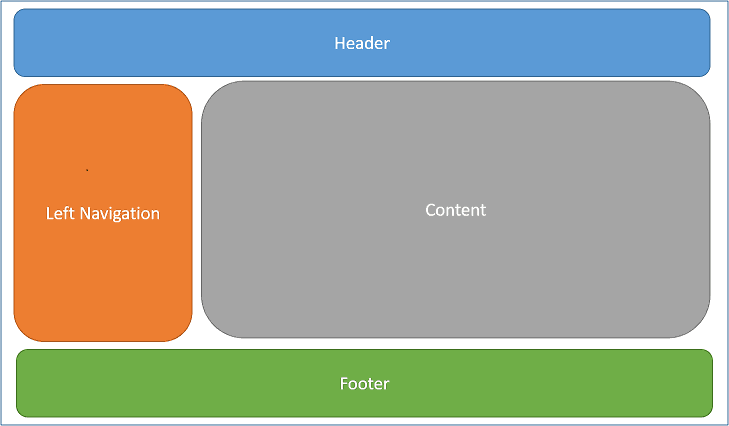
# Layouts và Section

# Layouts và Section trong ASP.NET MVC Core giúp chúng ta đảm bảo sự thống nhất giữa các trang trong toàn bộ các view của ứng dụng. Trong bài này chúng ta sẽ tìm hiểu cách tạo trang Layout và chia sẻ nó cho các view. Chúng ta sẽ sử dụng phương thức ****RenderBody****để tạo ra View. Chúng ta cũng được biết làm thế nào định nghĩa các section và sử dụng nó với ****RenderSection****. Cuối cùng chúng ta sẽ tìm hiểu về ****\_ViewStart****trong việc định nghĩa Layout

## Layout của trang web

Thiết kế của hầu hết các website bao gồm một menu, một phần header, footer và phần thanh sidebar. Như bạn thấy đi từ một trang này đến trang kia trong website thì chỉ có nội dung ở giữa là thay đổi. Nhưng menu, header, footer hay màu sắc toàn cục vẫn giữ nguyên. Điều này giúp website có cái nhìn thống nhất.

Một ứng dụng web nhìn cơ bản sẽ như sau:



## Layout trong ASP.NET Core là gì?

Views trong ASP.NET Core được tạo ra từ file **.cshtml** được đặt trong thư mục **Views**. Để giữ cho việc đồng nhất giữa các trang, chúng ta cần thêm header, footer và thanh điều hướng ở tất cả các view. Tuy nhiên điều này thường vướng víu và dễ sai sót đặc biệt nếu ta có nhiều view. Trang layout trong ASP.NET Core giúp chúng ta định nghĩa một giao diện có các phần tử dùng chung như header, footer, navigation menu trên trang ở một vị trí mà có thể dùng cho mọi nơi.

## Project ví dụ

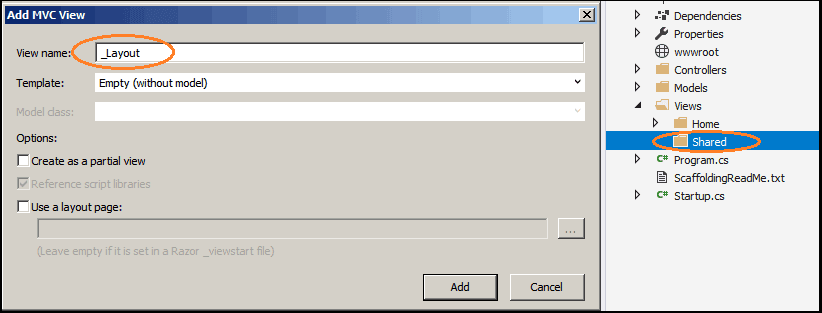
Mở một ứng dụng ASP.NET Core, chúng ta sử dụng kiến thức ở bài [Bắt đầu khởi tạo ứng dụng ASP.NET Core](https://tedu.com.vn/lap-trinh-aspnet-core/bat-dau-khoi-tao-ung-dung-aspnet-core-206.html).

## Vị trí của trang Layout

Theo quy ước của ASP.NET Core MVC thì các view được lưu trong thư mục Views. Mỗi Controller sẽ có một thư mục con trong thư mục Views tương ứng với tên trung với tên controller nhưng không có hậu tố "**Controller**". Tất cả các view được liên kết với controller trong thư mục này. Thư mục **Views**cũng có chứa thư mục tên là **Shared**, nơi chứa các view chia sẻ cho nhiều view khác dùng chung. Vì thế Layouts được đặt trong thư mục này:**Views/Shared.**

## Làm sao để tạo Layout

Tạo một thư mục tên là Shared bên trong thư mục Views. Chuột phải vào thư mục Shared chọn Add View. Và đặt tên View là **\_Layout**. Chọn **Empty Template** và không chọn vào cả 2 checkbox C**reate as a partial view** và **Use a layout page** vì chính thằng này không phải partial view cũng không cần sử dụng một layout page nào vì nó là layout to nhất rồi. Trong tương lai vẫn có những layout là con của 1 layout khác chúng ta sẽ tìm hiểu sau.



Click vào Add để tạo View. Nó sẽ tạo ra file **\_Layout.cshtml** trong thư mục **Views/Shared**. Mở nó ra và dán đoạn nội dung này vào.

<!DOCTYPE html>

<**html**>

<**head**>

<**meta** name="viewport" content="width=device-width" />

<**title**>Layout Example</**title**>

</**head**>

<**body**>

<**div** id="header">

<**h1**>Layout example in HTML</**h1**>

</**div**>

<**div** id="content">

@RenderBody()

</**div**>

<**div** id="Footer">

<**p**>This is Footer</**p**>

</**div**>

</**body**>

</**html**>

Đoạn HTML này có 3 phần là header, content và footer. Content là phần mà bạn muốn thêm nội dung các view khác vào.

## RenderBody

**RenderBody**là một phương thức đặc biệt đánh dấu vị trí nơi mà các view sử dụng Layout này sẽ được đặt vào đó khi chạy. Cơ bản thì nó là một vùng định trước của các View sẽ hiển thị ở đó.

@RenderBody()

Chỉ có một **RenderBody**được gọi trong một trang Layout.

## Chỉ ra Layout được sử dụng trong View

Thay đổi nội dung **Index**Action method trong **HomeController**như sau:

**public** IActionResult **Index**()

{

**return** View();

}

Mở Index.cshtml ra từ Views/Home và copy dòng:

@{

Layout = "\_Layout";

}

<**p**>Here goes the content</**p**>

Index view trở nên đơn giản hơn. Chúng ta sẽ loại bỏ các thẻ <html>, <head> và <body> . Các thẻ này giờ đây chuyển sang trang layout. Dòng đầu tiên của code là khối Razor, chúng ta định nghĩa trang layout sẽ được dùng cho view này. Chạy ứng dụng và nhìn kết quả.



Giờ hãy thêm một Action method khác tên **AboutUs**trong **HomeController**.

**public** IActionResult **AboutUs**()

{

**return** View();

}

Giờ hãy thêm một View và chọn Layout:

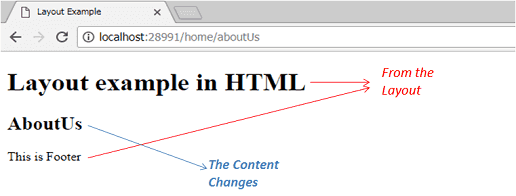
@{

Layout = "~/Views/Shared/\_Layout.cshtml";

}

<**h2**>About Us</**h2**>

Giờ hãy chạy ứng dụng và vào URL:**/Home/AboutUs**. Bạn sẽ thấy nội dung đã thay đổi khi header và footer đã được thêm vào từ Layout



## Đường dẫn Layout

Chúng ta sử dụng đường dẫn đầy đủ**"~/Views/Shared/\_Layout.cshtml"** trong view **AboutUs**, trong khi Index action lại sử dụng đường dẫn là **"\_Layout"**. Khi một đường dẫn partial được cung cấp cho Razor view engine tìm kiếm file layout trong thư mục **Views/{Controllers}** trước tiên, và theo sau đó là thư mục Shared. Quá trình này được gọi là tìm kiếm View.

## Section

ASP.NET Core cho phép bạn tạo một hoặc nhiều section cho layout. Section cần một cái tên và một thuộc tính chỉ ra xem nó có bắt buộc phải có trong các view con không. Phương thức **RenderSection(string name, bool required:false)** định nghĩa tên section.

@RenderSection("footer", required: false)

## Thêm Section vào Layout

Mở layout ra và dán dòng lệnh sau vào sau nội dung:

<div>

@RenderSection("Test", required: false)

</div>

Chúng ta định nghĩa một section tên **"Test"** với tham số required là false tức là không yêu cầu bắt buộc phải có section này trong view con.

## Định nghĩa Section trong view

Giờ hãy vào trang View của Index và dán đoạn sau:

@{

Layout = "\_Layout";

}

<**p**>Here goes the content</**p**>

@section Test

{

<**p**>Test Section</**p**>

}

Các section được định nghĩa với một khối lệnh Razor là **@section** bằng tên nó đi kèm. Chạy ứng dụng và kiểm tra xem các section đó có hiển thị sau content.

## Tạo section bắt buộc

Thay đổi tham số **required:true** khi định nghĩa section trong layout như dưới đây:

@RenderSection("Test", required: true)

Giờ hãy chạy ứng dụng và vào URL /Home/AboutUs sau đó bạn nhìn thấy lỗi sau:

InvalidOperationException: The layout page ‘/Views/Shared/\_Layout.cshtml’ cannot find the section ‘Test’ in the content page ‘/Views/Home/AboutUs.cshtml’.

Điều này xảy ra vì chúng ta không định nghĩa section **"Test"** trong view AboutUs

## \_ViewStart

Trong ví dụ trên, chúng ta định nghĩa layout và dùng nó cho các view. Định nghĩa layout trong tất cả các view thật khó để bảo trì. Cách đơn giải nhất là định nghĩa layout trong **\_ViewStart.cshtml.**

Mục đích của file này là cài đặt giá trị mặc định cho các View khác trong folder và folder con của nó. Giờ hãy đến thư mục Views và tạo **\_ViewStart.cshtml** trong thư mục Views và dán đoạn sau:

@{

Layout = "\_Layout";

}

Hãy bỏ hoặc comment đoạn code trên trong view **AboutUs**. Chạy ứng dụng và kiểm tra lại.

## Loại bỏ Layout từ view

Vì chúng ta đã thêm file **\_ViewStart.cshtml**vào thư mục Views nên layout giờ đã có thể áp dụng cho tất cả các View. Trong trường hợp bạn không muốn sử dụng layout cho view nào bạn chỉ cần đặt giá trị Layout là null.

@{

Layout = null;

}

hoặc đặt giá trị cho Layout là một Layout khác:

@{

Layout = "\_someOtherLayout";

}

Bạn có thể tạo một file **\_ViewStart** tách biệt trong thư mục Controller, vậy là bạn có thể ghi đè file **\_ViewStart** từ thư mục cha.

# ViewBag và ViewData

View cần lấy dữ liệu từ Controller. Một trong những cách truyền dữ liệu sang View là sử dụng đối tượng **ViewData**hoặc **ViewBag**. Bài này chúng ta sẽ tìm hiểu cách dùng **ViewBag**và **ViewData**.

## ViewData là gì?

**ViewData**là một thuộc tính của **Controller base class**, nó trả về một đối tượng **ViewDataDictionary**. **ViewDataDictionary**như tên của nó là một đối tượng dictionary cho phép lưu dữ liệu dạng key-value. Key phải là một chuỗi không phân biệt chữ hoa thường. Để truyền dữ liệu vào view bạn cần gán giá trị vào dictionary sử dụng key. Bạn có thể lưu bất kỳ số lượng **key-value** nào cần thiết trong **ViewData**.

**ViewData**truyền dữ liệu sang **View**từ **Controller**. Khi bạn gọi phương thức **View**trong Controller action, **ViewData**sẽ tự động gán vào View. Trong View bạn có thể truy cập giá trị được lưu trong **ViewData**cũng sử dụng key. Dữ liệu được lưu trong ViewData tồn tại chỉ trong request đó. Khi View được tạo xong cho client thì đối tượng ViewData đó cũng bị hủy.

## Sử dụng ViewData như thế nào?

Trong ví dụ dưới đây nêu cách dùng ViewData trong Controller action:

**public** IActionResult **SomeAction**()

{

ViewData["Greeting"] = "Hello World!";

**return** View();

}

Trong ví dụ trên, chúng ta thêm chữ **"Hello World"** vào ViewData sử dụng key là "Greeting". Bạn có thể gọi phương thức dưới đây trong View để nhận giá trị từ key "Greeting":

@ViewData["Greeting"]

## Gán đối tượng cho ViewData

Trong ví dụ trên, chúng ta lưu chuỗi dư liệu vào ViewData. Chuỗi dữ liệu có thể được dùng trực tiếp mà không cần chuyển kiểu. Bạn có thể lưu bất cứ kiểu nào của dữu liệu như kiểu số nguyên, kiểu logic hay đối tượng trong ViewData.

Để sử dụng như một kiểu dữ liệu khác thì bạn cần phải chuyển kiểu giá trị từ ViewData sang kiểu tương ứng khi bạn dùng nó. Ví dụ dưới đây nêu cách dùng ViewData để gán đối tượng từ Controller sang View:

**public** IActionResult **Index**()

{

ViewData["Greeting"] = "Hello World";

ViewData["Product"] = **new** ProductModel()

{

ProductID=1,

Name = "Samsung galaxy Note",

Brand = "Samsung",

Price = 19000

};

**return** View();

}

Trong View, chúng ta lấy sản phẩm từ ViewData và chuyển sang kiểu **ProductModel**và sử dụng nó:

@{

// Since Product isn't a string, it requires a cast.

var product = ViewData["Product"] as ProductModel;

}

@ViewData["Greeting"]!

@product.ProductID<br>

@product.Name<br>

@product.Brand<br>

@product.Price<br>

Bạn có thể dùng ViewData để gán dữ liệu từ Controller vào View bao gồm cả Partial View và Layout.

## ViewBag là gì?

Bạn đã thấy từ ví dụ trước chúng ta có thể lưu bất cứ thứ gì trong **ViewDataDictionary**, nhưng để truy cập được bất cứ thứ gì trong ViewData chúng ta cần phải chuyển kiểu dữ liệu.

**ViewBag**sử dụng kiểu động (dynamic) mà chúng ta đã có trong phiên bản C# 4.0. Nó là một vỏ bọc của ViewData và cung cấp thuộc tính động cho ViewData.

**ViewBag**có thể tiện dụng hơn để làm việc mà không cần chuyển kiểu. Sử dụng **ViewBag**như sau:

**public** IActionResult **SomeAction**()

{

ViewBag.Greeting = "Hello";

ViewBag.Product = **new** ProductModel()

{

ProductID = 1,

Name = "Samsung galaxy Note",

Brand = "Samsung",

Price = 19000

};

**return** View();

}

Sau đó bạn sử dụng trong View:

@ViewBag.Greeting World <br>

@ViewBag.product.ProductID<br>

@ViewBag.product.Name<br>

@ViewBag.product.Brand<br>

@ViewBag.product.Price<br>

## ViewBag và ViewData

Cả ViewBag và ViewData đều sử dụng **ViewDataDictionary**bên dưới. Vì thế bạn có thể sử dụng cả hai hoặc kết hợp chúng khi đọc hoặc ghi chúng. Ví dụ:

**ViewData**["Greeting"] = "Hello World";

Và nhận giá trị ở view sử dụng ViewBag:

@**ViewBag**.**Greeting**

## Sự khác nhau giữa ViewData và ViewBag

ViewData sử dụng cú pháp Dictionary để truy cập giá trị trong khi **ViewBag**sử dụng cú pháp truy cập giống truy cập thuộc tính của đối tượng. ViewData dẫn xuất từ **ViewDataDictionary**, nó có thuộc tính của dictionary như **ContainsKey**, **Add**, **Remove**và **Clear**.

ViewBag thì nhận từ DynamicViewData và nó cho phép tạo động các thuộc tính sử dụng dấu chấm (@ViewBag.SomeKey = <giá trị>) và không cần chuyển kiểu. Cú pháp của ViewBag giúp thêm giá trị nhanh hơn trong Controller và view.

ViewData cho phép sử dụng khoảng trắng trong Key vì nó là một chuỗi. Ví dụ V**iewData["Some Key With Namespace"]**. Nhưng ViewBag thì không thể.

ViewData cần phải chuyển kiểu dữ liệu khi không phải là một chuỗi. Nó cũng cần phải kiểm tra giá trị null để tránh lỗi. Check null đơn giản hơn trong ViewBag. Ví dụ: @ViewBag.Person?.Name.

## Khi nào sử dụng ViewBag và ViewData

**ViewData**và **ViewBag**đều là các tùy chọn hữu ích khi bạn muốn gán lượng dữ liệu nhỏ từ Controller sang View. Chọn cái nào thường hay tùy thuộc thói quen. Điểm yếu của cả **ViewBag**và **ViewData**là chúng giải quyết vấn đề động ở thời điểm runtime. Nó không kiểm tra kiểu ở lúc biên dịch (complie time). Vì thế tăng khả năng gây lỗi.

**ViewData**và **ViewBag**có thể truyền dữ liệu từ **Controller**sang **View**. Nó không thể truyền ngang từ Controller này sang Controller kia được.

# Model và ViewModel

Bài này mình sẽ kể cho bạn nghe tổng quan về Model trong ASP.NET  Core. Model có nghĩa rộng, nó là bất cứ cái gì tùy thuộc vào bạn muốn nó làm gì?. Trong ngữ cảnh của ASP.NET MVC thì Model có thể là một Domain Model, View Model hay là một Edit model. Chúng ta sẽ cùng tìm hiểu các khái niệm này và cách sử dụng chúng trong bài viết nhé.

## Model là gì?

Model là tập hợp các đối tượng chứa dữ liệu của ứng dụng có thể chứa thêm cả các logic nữa. Model chia làm một số loại dựa trên công dụng và nơi chúng sử dụng. Có 3 loại mục đích chính:

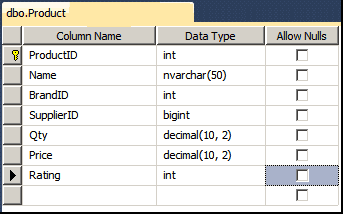
1. Domain Model
2. View Model
3. Edit Model

## Domain Model là gì?

Một Domain Model thể hiện một đối tượng trong database. Domain model thường có một mối quan hệ 1-1 với một bảng  trong cơ sở dữ liệu. Domain Model liên quan đến tầng truy cập dữ liệu (DAL) trong ứng dụng. Nó nhận từ cơ sở dữ liệu hoặc một nơi nào đó lưu dữ liệu bởi tầng truy cập dữ liệu. (DAL). Domain Model cũng được hiểu như entity model hay data model.

### Ví dụ về Domain Model

Ví dụ chúng ta có bảng Product như sau:



Nó được đại điện với Product Model:

**public** **class** **Product**

{

**public** **int** ProductId { **get**; **set**; }

**public** **string** Name { **get**; **set**; }

**public** Decimal Price { **get**; **set**; }

**public** **int** Rating { **get**; **set**; }

**public** Brand Brand { **get**; **set**; }

**public** Supplier Supplier { **get**; **set**; }

}

## ViewModel là gì?

ViewModel được tham chiếu đến các đối tượng chứa dữ liệu cần cho việc hiển thị cho người dùng. ViewModel liên quan đến tầng hiển thị của ứng dụng. Nó được định nghĩa dựa trên cách thức dữ liệu được hiển thị cho người dùng hơn là cách chúng được lưu trữ ra sao?

### Ví dụ về ViewModel

Ví dụ trong Product Model ở trên, người dùng cần hiển thị Brand Name và Supplier Name thay vì Brand ID và Supplier ID. Vì thế ViewModel trở thành:

**public** **class** **ProductViewModel**

{

**public** **int** ProductId { **get**; **set**; }

**public** **string** Name { **get**; **set**; }

**public** Decimal Price { **get**; **set**; }

**public** **int** Rating { **get**; **set**; }

**public** **string** BrandName { **get**; **set**; }

**public** **string** SupplierName { **get**; **set**; }

**public** **string** **getRating**()

{

**if** (Rating == 10)

{

**return** "\*\*\*\*\*";

}

**else** **if** (Rating >=8 )

{

**return** "\*\*\*\*";

}

**else** **if** (Rating >= 6)

{

**return** "\*\*\*";

}

**else** **if** (Rating >= 4)

{

**return** "\*\*";

}

**else**

{

**return** "\*";

}

}

}

Chúng ta bỏ Brand và Supplier thay vào đó trả về Brand Name và Supplier Name. View Model cũng có thể có View liên quan đến logic như là hiển thị Rating cho người dùng. Bạn không nên đặt bất cứ logic nào khác ngoài logic hiển thị trên view vào View Model.

## Edit Model là gì?

Edit Model hoặc Input Model đại điện dữ liệu cần để người dùng thay đổi hoặc thêm mới. Yêu cầu UI của Product cần chỉnh sửa khác với yêu cầu xem.

### Ví dụ về Edit Model

Ví dụ trong Product Model ở trên, người dùng cần hiển thị danh sách Brand và Supplier, trong khi thêm mới hay chỉnh sửa sản phẩm. Vì thế model trở thành:

**public** **class** **ProductEditModel**

{

**public** **int** ProductId { **get**; **set**; }

[Required(ErrorMessage = "Product Name is Required")]

[Display(Name = "Product Name")]

**public** **string** Name { **get**; **set**; }

**public** Decimal Price { **get**; **set**; }

**public** **int** Rating { **get**; **set**; }

**public** List<Brand> Brands { **get**; **set**; }

**public** List<Supplier> Suppliers { **get**; **set**; }

**public** **int** BrandID { **get**; **set**; }

**public** **int** SupplierID { **get**; **set**; }

}

## Model trong MVC Design Pattern

MVC Design Pattern là một pattern cho tầng hiển thị. Model trong MVC Design Pattern viết tắt của View Model và Edit Model. Hầu hết mọi người sử dụng từ View Model để chỉ cả 2 thằng này: View Model và Edit Model.

## Lợi ích của View Model

ViewModel rất hữu dụng khi bạn có một UI phức tạp, khi mà dữ liệu cần lấy ra từ vài Domain Model. Vì View Model được độc lập với Domain Model, nên việc này rất mềm dẻo và linh hoạt cho việc sử dụng nó. ViewModel giúp ứng dụng bảo mật hơn vì bạn không phải chìa các thuộc tính nhạy cảm và bí mật từ Domain Model ra như UserRole,  IsAdmin...

## Best practice với ViewModel

### Giữ Domain Model và View Model tách bạch

Tránh sử  dụng Domain model thay cho View Model. Bạn có thể chìa những thuộc tính nhạy cảm ra ngoài cho View. Domain Model thường gắn chặt vào database để sử dụng tầng DAL. Vì thế việc chìa các thuộc tính có thể sửa hay thêm mới vào database là nguy hiểm.

### Tạo Strongly Typed Views (View luôn có khai báo ViewModel tương ứng)

Trong Strongly Typed Views, bbanj hãy để View biết kiểu của viewModel được gán cho nó. Với strongly typed view, bạn sẽ có thể có sự trợ  giúp gợi ý thuộc tính từ Visual Studio và dễ tìm lỗi nếu có trong  quá trình phát triển.

### Sử dụng Data Annotation cho Validation

Sử dụng Data Annotation để khai báo cho thuộc tính của viewModel và giúp tận dụng cơ chế client validation trong ASP.NET Core.

### Chỉ đặt các dữ liệu cần thiết trong ViewModel

Giữ ViewModel nhỏ nhất có thể. Chỉ đặt các trường thực sự cần thiết cho việc hiển thị trong ViewModel.

### Sử dụng một Mapper để chuyển Model sang ViewModel

Model nhận từ cơ sở dữ liệu cần được map sang ViewModel. Bạn có thể sử dụng AutoMapper để thực hiện điều này.

### ViewModel có thể chứa các logic chỉ cho view

Về ý tưởng, ViewModel có thể chứa các dữ liệu và không có logic. Nhưng bạn có thể thêm một số logic đặc thù cho ViewModel.

### Sử dụng 1 ViewModel cho 1 View

Tạo một ViewModel cho một View. Sẽ dễ bảo trì và dễ tìm lỗi.

### Đồng nhất

Sử dụng ViewModel ngay cả cho các kịch bản đơn giản. Nó giúp dễ bảo trì và đảm bảo tính đồng nhất cho toàn ứng dụng.

## Tổng kết

Chúng ta đã tìm hiểu về Domain Model, View Model và Edit Model trong ứng dụng ASP.NET Core. Chúng ta cũng học được các best practice sử dụng View Model.

cách set up cho project nào chạy

# Strongly Typed View

Trong bài viết trước mình đã hướng dẫn các bạn xây dựng một HTML Form đơn giản. Chúng ta dùng **ViewModel**nhưng không gán nó cho View. ViewModel có thể được gán từ controller sang View dùng **ViewBag**hoặc **ViewData**. ASP.NET Core cung cấp khả năng có thể gán strongly typed view model hoặc một object sang view. Cách tiếp cận này giúp chúng ta dễ dàng bảo trì code và phát hiện lỗi từ lúc biên dịch. Cơ chế **Scaffolding**trong Visual Studio có thể được dùng để tạo một View dựa trên ViewModel.

## Strongly Typed View là gì?

View nào mà được kết hợp với một kiểu cụ thể của ViewModel thì được gọi là Strongly Typed View. Bằng cách chỉ ra model, Visual Studio cung cấp cơ chế gợi ý và kiểm tra kiểu lúc biên dịch. Chúng ta đã học cách truyền dữ liệu từ Controller sang View trước đây. Điều này thường được giải quyết với ViewBag hoặc ViewData. Trình biên dịch không biết gì về kiểu của model.

Trong Strongly typed view, chúng ta biết View sử dụng ViewModel nào sử dụng khai báo **@model**.

## Khai báo @model

Strongly typed view được tạo sử dụng khai báo **@model**. ViewData có một thuộc tính đặc biệt gọi là Model. Nó là một đối tượng kiểu dynamic. Nó cho phép chúng ta sử dụng ViewData.Model.Prop. Sử dụng Model cách này không giúp Visual phát hiện và kiểm tra thuộc tính của model mà phải chờ lúc chạy chương trình mới phát hiện lỗi. Nhờ thế thì việc kiểm tra kiểu lúc biên dịch cũng không có.

Vấn đề ở trên đã được giải quyết bằng cách chỉ cho View biết kiểu của model sẽ được găn vào ViewData.Model. Nó được giải quyết bằng khai báo @model, nó được đặt ở trên cùng của file View và chỉ ra kiểu của ViewModel được gán.

Khi bạn sử dụng khai báo @model, Razor engine sẽ gán kiểu choViewData.Model. Thuộc tính Model sẽ trả về kiểu được khai báo.

Ví dụ:

@model ProductEditModel

Được hiểu thành:

**ProductEditModel** Model;

Vì thế nên kiểu của Model giờ được biết trước ở lúc biên dịch, chúng ta sẽ tận dụng được trình gợi ý cú pháp (IntelliSense) và kiểm tra kiểu lúc biên dịch.

## Ví dụ về Strongly Typed View

Chúng ta tiếp tục đi từ điểm cuối của bài trước. Cập nhật code để thêm ProductEditModel vào đầu của View:

@model ProductEditModel

Và bạn sẽ dùng thuộc tính Model để truy cập đến thuộc tính của nó:

<form action="/home/create" method="post">

<**label** for="@Model.Name">Name</**label**>

<input type="text" name="@Model.Name" />

<**label** for="@Model.Rate">Rate</**label**>

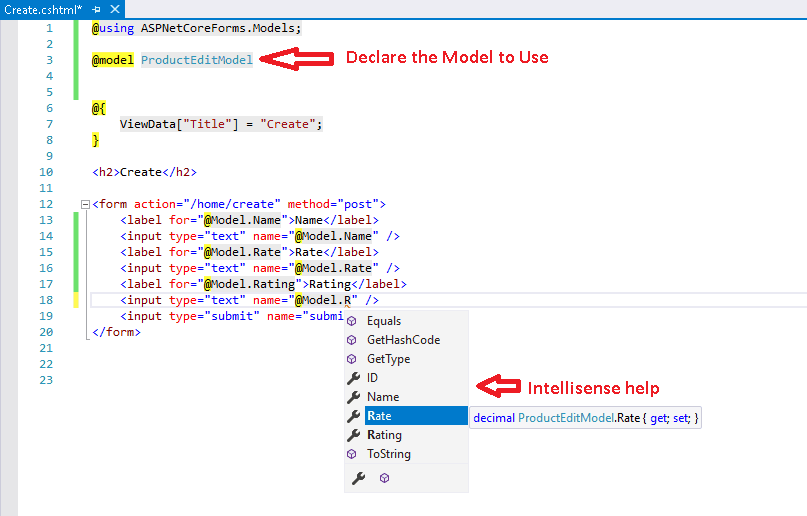
<input type="text" name="@Model.Rate" />

<**label** for="@Model.Rating">Rating</**label**>

<input type="text" name="@Model.Rating" />

<**input** type="submit" name="submit" />

</**form**>



## Gán Model từ Controller sang View

Bước tiếp theo chúng ta sẽ gán thể hiện của ProductEditModel sang View. Mở HomeController.cs và đến phương thức Create:

[HttpGet]

**public** IActionResult **Create**()

{

ProductEditModel model = **new** ProductEditModel();

**return** View(model);

}

Giờ hãy chạy ứng dụng

## Lợi ích của Strongly Typed View

1. Hỗ trợ IntelliSense
2. Kiểm tra kiểu lúc biên dịch
3. Không phải chuyển kiểu

Vì thế chỉ có một thuộc tính Model, bạn chỉ có thể có 1 ViewModel trên View.

## Model và model

Rất dễ nhầm giữa Model và model. Khai báo model được dùng để khai báo kiểu của ViewModel. Còn Model là biến được dùng để truy cập vào ViewModel. Kiểu của Model được khai báo bằng từ khóa @model.

## Sử dụng Scaffolding để tạo ra Strongly Typed View

Visual Studio Scaffolding System cho phép chúng ta tạo nhanh một view. Mở HomeController.cs và tạo một Action method là Edit.

[HttpGet]

**public** IActionResult **Edit**()

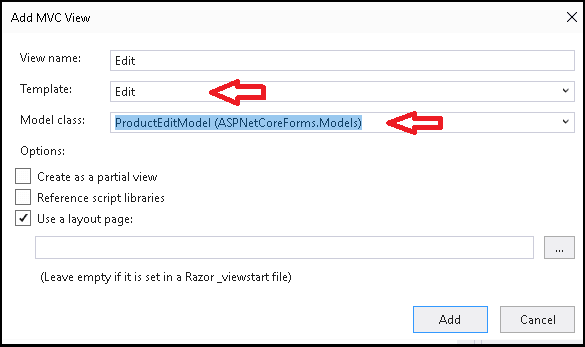
{

ProductEditModel model = **new** ProductEditModel();

**return** View(model);

}

Chọn template là Edit. Chọn Model class là ProductEditModel. Click vào Add để tạo ra một View. Sau đó mở View ra:



@model ASPNetCoreForms.Models.ProductEditModel

@{

ViewData["Title"] = "Create";

}

<h2>Create</h2>

<h4>ProductEditModel</h4>

<hr />

<div **class**="row">

<div **class**="col-md-4">

<form asp-action="Create" method="post">

<div asp-validation-summary="ModelOnly" **class**="text-danger"></div>

<div **class**="form-group">

<label asp-**for**="ID" **class**="control-label"></label>

<input asp-**for**="ID" **class**="form-control" />

<span asp-validation-**for**="ID" **class**="text-danger"></span>

</div>

<div **class**="form-group">

<label asp-**for**="Name" **class**="control-label"></label>

<input asp-**for**="Name" **class**="form-control" />

<span asp-validation-**for**="Name" **class**="text-danger"></span>

</div>

<div **class**="form-group">

<label asp-**for**="Rate" **class**="control-label"></label>

<input asp-**for**="Rate" **class**="form-control" />

<span asp-validation-**for**="Rate" **class**="text-danger"></span>

</div>

<div **class**="form-group">

<label asp-**for**="Rating" **class**="control-label"></label>

<input asp-**for**="Rating" **class**="form-control" />

<span asp-validation-**for**="Rating" **class**="text-danger"></span>

</div>

<div **class**="form-group">

<input type="submit" **value**="Create" **class**="btn btn-default" />

</div>

</form>

</div>

</div>

<div><a asp-action="Index">Back to List</a></div>

## Tổng kết

Chúng ta đã kết hợp view vào ViewModel để tạo ra Strongly typed view. Chúng ta cũng học cách sử dụng Scaffolding để tạo ra Strongly typed view.

# Tag Helpers

Tag Helpers là tính năng mới của ASP.NET Core, nó giúp chúng ta thêm code phía server vào HTML dễ dàng. Trong bài này chúng ta sẽ sử dụng nó trong HTML Form mà chúng ta tạo trong Strongly Typed View trước.

## Tag Helper là gì?

Tag Helper giúp chúng ta viết phần tử HTML trong Razor sử dụng cú pháp thân thiện với HTML. Nó nhìn như là HTML chuẩn vậy nhưng code được xử lý bởi Razor Engine trên server và nó tận dụng đươc các ưu điểm của việc xử lý phía server.

Razor được tạo sử dụng Tag Helper nhìn như phần tử HTML thuần. Nó thao tác với các phần tử HTML như thêm mới phần tử HTML hay thay thế các nội dung có sẵn bằng một cái mới.

Ví dụ, sử dụng thẻ Form Tag Helper, chúng ta có thể tạo ra thẻ <form> như dưới đây. Với các thuộc tính asp-action và asp-controller của Form Tag Helper:

<**form** asp-action="create" asp-controller="home">

Sẽ được gen ra HTML:

<**form** action="/home/create" method="post">

## Mục đích của Tag Helpers

Bạn có thể tạo form mà không cần dùng Tag Helper (hoặc HTML Helper) như bài trước. Tuy nhiên Tag Helper sẽ giúp tạo ra view HTML đơn giản hơn dựa trên dữ liệu từ Model gắn vào nó. Ví dụ Label Tag Helper sẽ tạo ra tiêu đề dựa trên attribute Data Annotation trong View Model. Tương tự như thế thì Input Tag Helper sẽ tạo ra id, name, type của phần tử HTML dựa  trên kiểu dữ liệu của Model và thuộc tính Data Annotation.

## Sử dụng Tag Helper?

ASP.NET Core Tag Helper nằm trong thư viện [**Microsoft.AspNetCore.Mvc.TagHelpers**](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.taghelpers?view=aspnetcore-2.1)bạn cần import thư viện này để sử dụng Tag Helper.

### Thêm Tag Helper sử dụng @addTagHelper

Để sử dụng Tag Helper bạn cần thêm khai báo @addTagHelper vào view, nơi mà bạn muốn sử dụng.

@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers

Đoạn code trên sử dụng wildcard ("\*") để chỉ ra tất cả Tag Helper được thêm vào từ thư viện Microsoft.AspNetCore.Mvc.TagHelpers.

### Thêm Tag Helper toàn bộ các View

Thêm @addTagHelper vào một view nào đó chỉ có tác dụng trên view đó. Bạn có thể thêm @addTagHelper vào **\_ViewImports.cshtm**l để sử dụng Tag Helper trên toàn bộ các view của ứng dụng.

### Bỏ Tag Helper

Đoạn code dưới đây loại bỏ tất cả tag helper từ assembly **Microsoft.AspNetCore.Mvc.TagHelpers** từ một view cụ thể:

@removeTagHelper "\*, Microsoft.AspNetCore.Mvc.TagHelpers"

### Thêm một số Tag Helper chỉ định

Thay vì add tất cả tag helper thì bạn có thể chọn ra một số cái mà bạn muốn dùng thôi:

@addTagHelper "Microsoft.AspNetCore.Mvc.TagHelpers.InputTagHelper, Microsoft.AspNetCore.Mvc.TagHelpers"

### Tắt Tag Helper với !

Bằng cách sử dụng ký tự ! trước mỗi phần tử HTML, bạn sẽ có thể vô hiệu hóa tag helper cho phần tử đó:

<!label asp-**for**="Name"></**!label**>

Tag helper label này được vô hiệu hóa với đoạn code trên. Bạn phải áp dụng ký tự ! cho cả thẻ đóng và thẻ mở.

### Sử dụng @tagHelperPrefix để bật Tag Helper

Thay vì vô hiệu hóa tag helper sử dụng ký tự !, bạn có thể sử dụng @tagHelperPrefix

@tagHelperPrefix th:

Giờ thì tiền tố **th:** phải được chỉ ra cho tất cả các tag helper trên view, để bật tag helper cho nó:

<th:label asp-**for**="Name"></th:label> //Tag helper **is** enabled

<label asp-**for**="Address"></label> //Tag helper **is** disabled

## Ví dụ về Tag Helper

Chúng ta xây dựng một Form đơn giản như bài trước. Hãy dùng Tag Helper. Đầu tiên mở **HomeController.cs**ra và thay đổi action method **Create**.

[HttpGet]

**public** IActionResult **Create**()

{

ProductEditModel model = **new** ProductEditModel();

**return** View(model);

}

Thể hiện của **ProductEditModel**được gán vào View để tạo [Strongly Typed View](https://tedu.com.vn/lap-trinh-aspnet-core/strongly-typed-view-trong-aspnet-core-248.html). Mở **Create.csthml** từ thư mục**/Views/Home**.

### Form Tag Helper

Form Tag Helper được bao bởi thẻ <form>. Form Tag Helper cung cấp một số thuộc tính phía server giúp chúng ta thao tác để tạo ra HTML. Một số thuộc tính đó là:

* asp-controller: Chỉ ra tên Controller sử dụng
* asp-action: Chỉ ra tên action method sử dụng
* asp-area: Chỉ ra tên Area sử dụng

Ví dụ:

<**form** asp-controller="Home" asp-action="Create">

Đoạn code trên sẽ biên dịch ra HTML thuần là:

<form method="post" action="/Home/Create">

<**input** name="\_\_RequestVerificationToken" type="hidden" value="CfDJ8PlIso5McDBOjgPkVg9O4mnNiAE8U0HkVlA9e-Mtc76u7fSjCnoy909Co49eGlbyJxpp-nYphF\_XkOrPo0tTGdygc2H8nCtZCcGURMZ9Uf01fPOg5jRARxTHXnb8N6yYADtdQSnJItXtYsir8GCWqZM" />

</**form**>

Chú ý là Form Tag Helper nó sẽ tự động thêm Antiy-Forgery Token vào HTML tạo ra.

### Label Tag Helper

Label Tag Helper được áp dụng cho phần tử label. Nó có một thuộc tính là asp-for. Sử dụng như sau:

<label asp-**for**="@Model.Name"></**label**>

Nó sẽ dịch ra:

<**label** for="Name">Name</**label**>

Tên của trường đó sẽ được lấy từ tên của thuộc tính trong Model hoặc từ Data Annotation của thuộc tính trong Model. Sử dụng từ khóa @Model là không bắt buộc. Bạn có thể chỉ ra tên thuộc tính trong Model luôn:

<label asp-**for**="Name"></**label**>

Cái này bạn có thể đọc Strongly Typed View ở bài trước.

### Input Tag Helper

Tương tự, Input Tag Helper cũng được áp dụng cho phần tử input:

<input asp-**for**="Name" />

Nó sẽ dịch ra:

<input type="text" id="Name" name="Name" value="" />

Thuộc tính type, id và name tự động lấy từ tên và kiểu dữ liệu của trường đó trong Model. Cuối cùng form nhìn như sau:

<form asp-controller="Home" asp-action="Create">

<**label** asp-for="Name"></**label**>

<input asp-**for**="Name" />

<**label** asp-for="Rate"></**label**>

<input asp-**for**="Rate" />

<**label** asp-for="Rating"></**label**>

<input asp-**for**="Rating" />

<**input** type="submit" name="submit" />

</**form**>

Khi chạy nó sẽ tạo ra HTML như này:

<**h2**>Create</**h2**>

<**form** action="/Home/Create" method="post">

<**label** for="Name">Name</**label**>

<**input** type="text" id="Name" name="Name" value="" />

<**label** for="Rate">Rate</**label**>

<**input** type="text" data-val="true" data-val-number="The field Rate must be a number." data-val-required="The Rate field is required." id="Rate" name="Rate" value="0.00" />

<**label** for="Rating">Rating</**label**>

<**input** type="number" data-val="true" data-val-required="The Rating field is required." id="Rating" name="Rating" value="0" />

<**input** type="submit" name="submit" />

<**input** name="\_\_RequestVerificationToken" type="hidden" value="CfDJ8PlIso5McDBOjgPkVg9O4mlYQIZDgGzfiyx\_ZiUck9A5E6DqBMIPnDCjFTyw5As2ALJT34MG\_lmaAanTwCeq1ugZ1r7w7qBsQCIgmO7Zw1C6CFvJNj6y4kxrSq0PT0Lk7XXqPv9NDsTL7C-6aB85Mjo" />

</**form**>

## Lợi ích của Tag Helper

### Thân thiện với cú pháp HTML

Tag Helper nhìn như là phần tử HTML chuẩn. Các Front end Developer không cần học cú pháp C# hay Razor để thêm các phần tử này vào View. Vì thế nó dễ dàng đạt được tính chất chia để trị. Bạn có thể dễ dàng thêm CSS hoặc bất cứ thuộc tính HTML nào vào Tag Helper như là với HTML.

### Được hỗ trợ bởi IntelliSense

Tag Helper cung cấp sự hỗ trợ bởi cơ chế gợi ý thông minh của Visual Studio. Ví dụ dưới khi chúng ta thấy gợi ý cho label của thuộc tính Rating.

### https://tedu.com.vn/uploaded/images/072019/Intellisense-Help-Tag-Helpers.png

### Code sạch hơn

Code sẽ sạch và rõ ràng hơn sử dụng HTML Helper cũ. Không cần phải sử dụng ký tự @ để chuyển giữa C# và HTML.

### Dễ mở rộng hơn

ASP.NET Core MVC cung cấp nhiều tag helper có sẵn giúp chúng ta tạo view. Nhưng nếu không có tag helper nào phù hợp với nhu cầu. Bạn cũng có thể tạo ra Tag Helper của riêng mình bằng cách mở rộng các Tag Helper có sẵn. Trong tương lai mình sẽ có bài viết về việc này.

### Danh sách các Tag Helper có sẵn

Thư viện Microsoft.AspNetCore.Mvc.TagHelpers chứa nhiều các Tag Helper có sẵn cho các công việc thường dùng như tạo form, validate form, label, link...

| **TAG HELPER** | **TẠO RA** | **THUỘC TÍNH** |
| --- | --- | --- |
| Form Tag Helper | <form> | asp-action, asp-all-route-data, asp-area, asp-controller, asp-fragment, asp-host, asp-page, asp-page-handler,asp-protocol,asp-route, asp-route- |
| Anchor Tag Helpers | <a> | asp-action, asp-all-route-data, asp-area, asp-controller, asp-Fragment, asp-host, asp-page, asp-page-handler, asp-Protocol, asp-route, asp-route- |
| Cache Tag Helper | <cache> | enabled1,expires-after2,expires-on3,expires-sliding4,priority5,vary-by6 |
| Environment Tag Helper | <environment> | names, include, exclude |
| Image Tag Helper | <img> | append-version |
| Input Tag Helper | <input> | for |
| Label Tag Helper | <label> | for |
| Link Tag Helper | <link> | href-include, href-exclude, fallback-href, fallback-href-include, fallback-href-exclude, fallback-test-class, fallback-test-value, fallback-test-property, fallback-test-value, append-version |
| Options Tag Helper | <select> | asp-for, asp-items |
| Partial Tag Helper | <partial> | name,model,for,view-data |
| Script Tag Helper | <script> | src-include, src-exclude, fallback-src, fallback-src-include, fallback-src-exclude fallback-test, append-version |
| Select Tag Helper | <select> | for, items |
| Textarea Tag Helper | <textarea> | for |
| Validation Message Tag Helper | <span> | validation-for |
| Validation Summary Tag Helper | <div> | validation-summary |

# Input Tag Helper

Input Tag Helper tạo ra phần tử HTML tương ứng với thuộc tính của Model là input. Thuộc tính model được kết hợp với tag helper sử dụng thuộc tính asp-for. Input tag helper tạo ra các thuộc tính HTML tương ứng như type, name và id dựa trên các thông tin của thuộc tính model như kiểu dữ liệu và data annotation được áp dụng vào thuộc tính của ViewModel. ViewModel phải gắn vào một strongly typed view. Nó cũng cung cấp các validation cho thuộc tính giúp có thể hỗ trợ validate phía client. Thuộc tính asp-format giúp định dạng các dữ liệu nhập vào.

## Input tag helper

Input tag helper áp dụng trên phần tử HTML là <input. Nó chỉ có 2 thuộc tính server-side là:

* asp-for
* asp-format

### asp-for

Thuộc tính này kết hợp với thuộc tính tương ứng trong View Model và tạo ra HTML dựa trên thông tin thuộc tính đó như kiểu, tên, và data annotation (chỉ dẫn trên thuộc tính của ViewModel).

<input asp-**for**=”<Expression Name>” />

### Ví dụ về asp-for

Bạn có một model có thuộc tính Name:

**public** **string** Name { **get**; **set**; }

Ở view sẽ có một phần tử input:

<input asp-**for**="Name" /><br />

Nó sẽ tạo ra một phần tử HTML như sau khi thực thi:

<**input** type="text" id="Name" name="Name" value="" /><**br** />

## Các thuộc tính của HTML Input

Thuộc tính asp-for tự động tạo ra phần tử HTML dựa trên các thông số sau đây:

* Kiểu chỉ ra trong HTML
* Data annotation được áp dụng cho thuộc tính model
* Kiểu dữ liệu của thuộc tính đó trong .NET được dùng

### Kiểu chỉ ra trong HTML

Type là thuộc tính được định nghĩa trong HTML và nó sẽ không bị ghi đè:

[EmailAddress]

**public** **string** Email { **get**; **set**; }

View sẽ là:

<**input** type=”text” asp-for="Email" /><**br** />

HTML sẽ tạo ra một input text vì cái data annotation là [EmailAddress] nó không ảnh hưởng đến kết quả HTML tạo ra. Nó vẫn tạo ra như sau:

<**input** type="text" id="Email" name="Email" value="" /><**br** />

### Dựa trên thuộc tính Data Annotaion

Thuộc tính Data Annotation được áp dụng cho model để tạo ra thuộc tính type. Ví dụ ta có data annotation EmailAddress nó sẽ tự dịch ra type="email":

[EmailAddress]

**public** **string** Email { **get**; **set**; }

Code view:

<input asp-**for**="Email" /><br />

HTML tạo ra sẽ là:

<**input** type="email" data-val="true" data-val-email="The Email field is not a valid e-mail address." id="Email" name="Email" value="" /><**br** />

Đây là danh sách các Data Annotation và các kiểu của input tạo ra tương ứng nếu sử dụng chúng:

| **ATTRIBUTE** | **INPUT TYPE** |
| --- | --- |
| [EmailAddress] | type=”email” |
| [Url] | type=”url” |
| [HiddenInput] | type=”hidden” |
| [Phone] | type=”tel” |
| [DataType(DataType.Password)] | type=”password” |
| [DataType(DataType.Date)] | type=”date” |
| [DataType(DataType.Time)] | type=”time” |

### Dựa trên .NET Type

Nếu thuộc tính Data Annotation không được chỉ ra, thì Input Tag Helper sẽ sử dụng kiểu .NET của thộc tính đó để xác định kiểu của HTML tạo ra. Ví dụ:

Model:

**public** DateTime DateOfJoining { **get**; **set**; }

View

<input asp-**for**="DateOfJoining" />

HTML:

<**input** type="datetime-local" data-val="true" data-val-required="The DateOfJoining field is required." id="DateOfJoining" name="DateOfJoining" value="0001-01-01T00:00:00.000" /><**br** />

Hoặc là kiểu boolean:

Model

**public** **bool** isActive { **get**; **set**; }

View:

<input asp-**for**="isActive" /><br />

HTML:

<**input** type="checkbox" data-val="true" data-val-required="The isActive field is required." id="isActive" name="isActive" value="true" /><**br** />

Đây là danh sách các kiểu .NET hay dùng và input tạo ra tương ứng:

| **NET TYPE** | **INPUT TYPE** |
| --- | --- |
| Bool | type=”checkbox” |
| String | type=”text” |
| DateTime | type=”datetime-local” |
| Byte, int, Single, Double | type=”number” |
| decimal, double, float | type=”text” |

### Thuộc tính id và name trong HTML

Thuộc tính id và name trong HTML nhận từ biểu thức tên của thuộc tính qua asp-for

[EmailAddress]

**public** **string** Email { **get**; **set**; }

<**input** type=”text” asp-for="Email" /><**br** />

Nó sẽ tạo ra HTML như sau. Thuộc tính id và name giống như giá trị được điền trong asp-for

<**input** type="text" id="Email" name="Email" value="" /><**br** />

### Các thuộc tính con

Hãy xem xét model person dưới đây có chứa thuộc tính Address và một mảng màu:

**public** **class** **person**

{

**public** List<**string**> colors { **get**; **set**; }

**public** Address Address { **get**; **set**; }

**public** **person**()

{

Address = **new** Address();

colors = **new** List<**string**>(){"red","blue"};

}

}

**public** **class** **Address**

{

**public** **string** address1 { **get**; **set**; }

}

Bạn có thể tham chiếu đến address1:

<input asp-**for**="Address.address1" /><br />

Đoạn code ở trên tạo ra HTML như bên dưới. Chú ý rằng thuộc tính name và id giống với biểu thức trong asp-for nhé. Thuộc tính id sử dụng dấu gạch dưới thay cho dấu chấm:

<**input** type="text" id="Address\_address1" name="Address.address1" value="" /><**br** />

Danh sách màu:

<input asp-**for**="colors[0]" /><br />

Nó sẽ tạo ra HTML và chỉ ra tên của phần tử dựa trên dấu [] và chỉ số ở trong

<**input** type="text" id="colors\_0\_" name="colors[0]" value="red" /><**br** />

### Các thuộc tính Validation

Input Tag Helper cũng tạo ra các thuộc tính validate dữ liệu trong HTML. Các thuộc tính validate dữ liệu bắt đầu bằng data-val-\*. Chúng chứa các thông tin validation như giá trị lớn nhất nhỏ nhất, bắt buộc nhập hay không, miền giá trị cho phép, biểu thức quy tắc hay thông báo lỗi...

ASP.NET Core Unobtrusive Client Validation framework sử dụng các thuộc tính data-val-\* này để validate dữ liệu trên phía client. Input tag helper xác định các thuộc tính validation bằng cách phân tích các kiểu trong .NET và data annotation áp dụng cho thuộc tính model.

Ví dụ áp dụng thuộc tính [Required] cho kiểu string, thuộc tính data-val và data-val-required sẽ được thêm vào thẻ input HTML:

[Required]

**public** **string** Name { **get**; **set**; }

HTML tạo ra:

<**input** asp-for="@Model.Name" /><**br** />

<**input** type="text" data-val="true" data-val-required="The Name field is required." id="Name" name="Name" value="" /><**br** />

Thêm thuộc tính ErrorMessage vào Required:

[Required(ErrorMessage ="Please enter the name")]

**public** **string** Name { **get**; **set**; }

View:

<input asp-**for**="Name" /><br />

HTML tạo ra sẽ là:

<**input** type="text" data-val="true" data-val-required="Please enter the name" id="Name" name="Name" value="" /><**br** />

Kiểu DataTime tự động tạo ra data-val-required, ngay cả khi thuộc tính [Required] không được chỉ ra:

**public** DateTime DateOfJoining { **get**; **set**; }

<input asp-**for**="@Model.DateOfJoining" />

HTML tạo ra:

<**input** type="datetime-local" data-val="true" data-val-required="The DateOfJoining field is required." id="DateOfJoining" name="DateOfJoining" value="0001-01-01T00:00:00.000" /><**br** />

Bạn có thể bỏ việc này bằng cách đặt kiểu DateTime là nullable:

**public** DateTime? DateOfJoining { **get**; **set**; }

<**label** for="DateOfJoining">DateOfJoining</**label**>

HTML tạo ra:

<**input** type="datetime-local" id="DateOfJoining" name="DateOfJoining" value="" /><**br** />

Danh sách một số kiểu của data annotation sẽ tạo ra thuộc tính validation tương ứng:

| **ATTRIBUTE** | **MÔ TẢ** |
| --- | --- |
| Compare | Chỉ ra field khác cần so sánh giá trị với field này |
| MaxLength | Chỉ ra số ký tự dài nhất có thể được chấp nhận |
| MinLength | Chỉ ra số ký tự ít nhất có thể được chấp nhận |
| Range | Chỉ ra miền giá trị được chấp nhận |
| RegularExpression | Giá trị phải tuân theo regular expression |
| Remote | Bật tính năng validation phía client đến 1 server remote, ví dụ kiểm tra giá trị username có tồn tại hay không. |
| Required | Giá trị bắt buộc phải nhập. Chú ý là các kiểu DateTime hay kiểu số mặc định sẽ có cái này, bạn cần sử dụng nullable để bỏ nó đi |
| StringLength | Chỉ ra độ dài tối đa của chuỗi được chấp nhận |

## asp-format

Thuộc tính asp-format định dạng chuỗi cho thuộc tính input. Ví dụ bạn format số với 2 số thập phân đằng sau:

**public** **decimal** Balance { **get**; **set**; }

<input asp-**for**="Balance" asp-format="{0:N2}" /><br />

HTML tạo ra:

<**input** type="text" data-val="true" data-val-number="The field Balance must be a number." data-val-required="The Balance field is required." id="Balance" name="Balance" value="0.00" /><**br** />

Đây là [danh sách các format type](https://docs.microsoft.com/en-us/dotnet/standard/base-types/formatting-types)dùng cho as-format <https://docs.microsoft.com/en-us/dotnet/standard/base-types/formatting-types>

# Environment Tag Helper

Environment Tag Helper hỗ trợ tạo ra các nội dung phụ thuộc vào biến quy định môi trường trong ASP.NET Core. Bài này chúng ta sẽ tìm hiểu về nó nhé.

## Environment Tag Helper

Environment Tag Helper áp dụng sử dụng thẻ <environment>. Nó xác định giá trị hiện tại của biến môi trường **ASPNETCORE\_ENVIRONMENT**để render ra nội dung dựa trên giá trị của biến. Giá trị của **ASPNETCORE\_ENVIRONMENT**được truy cập bởi **IHostingEnvironment.EnvironmentName**. Biến môi trường này đọc tại thời điểm ứng dụng khởi động và giá trị được đọc vào IHostingEnvironment.EnvironmentName. Theo quy ước thì có 3 giá trị tương ứng với 3 môi trường được hỗ trợ: **Development**, **Staging**và **Production**. Tuy nhiên bạn có thể đặt bất cứ tên môi trường nào tùy thích.

## Thuộc tính Environment Tag Helper

Environment Tag Helper hỗ trợ 3 thuộc tính.

### include

Dấu hỏi được dùng để ngăn cách các giá trị là tên môi trường mà nội dung bên trong cặp thẻ sẽ tạo ra ở các môi trường đó.

<**environment** include="Development">

<**h1**>This only renders in the Development Environment</**h1**>

</**environment**>

<**environment** include="Staging,Production">

<**h1**>This only renders in the Staging and Production Environments!</**h1**>

</**environment**>

### exclude

Một danh sách các tên môi trường cũng được chia tách trong giá trị của thuộc tính exclude. Thuộc tính này quy định nội dung được hiển thị ở tất cả các môi trường trừ các môi trường trong danh sách này. Exclude cho ASP.NET Core biết rằng sẽ không render ra nội dung nếu tên môi trường hiện tại đúng với môi trường đã chỉ ra:

<**environment** exclude="Development">

<**h1**>This content is not rendered in the Development Environments!</**h1**>

</**environment**>

Cả include và exclude được áp dụng trùng một tên môi trường thì exclude luôn được ưu tiên.

### names

Một dấu phẩy cũng được dùng để chia tách tên môi trường mà nội dung được render. Nếu tên môi trường hiện tại cung nằm trong danh sách exclude thì nội dung vẫn không được render. Vì exclude được ưu tiên cao nhất.

<**environment** names="Development">

<**h1**>This only renders in the Development Environment</**h1**>

</**environment**>

<**environment** names="Development,Production">

<**h1**>This renders in Development and Production Environments</**h1**>

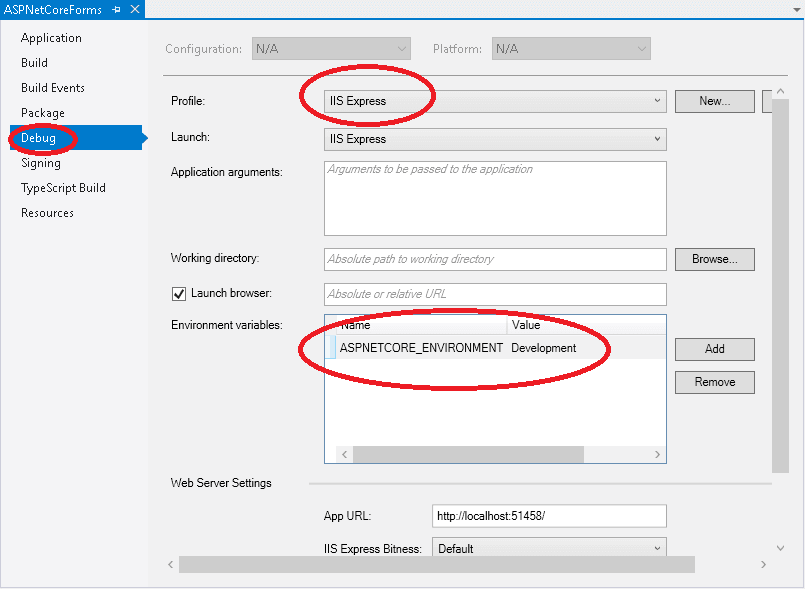
</**environment**>

## Chỉ ra biến môi trường

Bạn có thể chỉ ra biến môi trường từ Properties của Project hoặc qua file lauchSettings.json.

### Project Properties

Mở Project Properties ra và chọn tab Debug. Trong phần tên biến môi trường chọn giá trị biến **ASPNETCORE\_ENVIRONMENT**và thay đổi giá trị. Nếu tên biến là **ASPNETCORE\_ENVIRONMENT**không tồn tại hãy click vào nút thêm mới.



### lauchSettings.json

Cách khác là thay đổi biến môi trường sử dụng **lauchSettings.json** như dưới đây:



## Ví dụ sử dụng Environment Tag Helper

Hầu hết các trường hợp sử dụng Environment Tag Helper là sử dụng để nhúng các file CSS hoặc JavaScript cho các môi trường khác nhau như **Development** hay **Production**. Ví dụ bạn cần load version full của CSS trong môi trường **Development**để debug tuy nhiên với **Production**bạn chỉ cần nhúng file đã bundle tức là file đã nén để tăng tốc độ và giảm băng thông truyền dữ liệu.

Ví dụ dưới đây cho phép nhúng file đã nén vào của **style.css** trên tất cả các môi trường trừ môi trường **Development**.

<environment include="Development">

<**link** rel="stylesheet" href="~/css/style.css" />

</**environment**>

<environment exclude="Development">

<**link** rel="stylesheet" href="~/css/style.min.css" />

</**environment**>

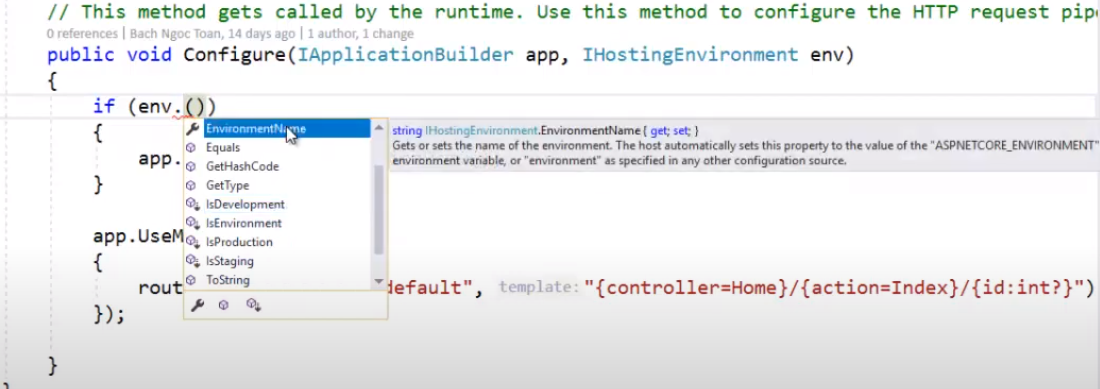
## Exclude được ưu tiên

Nếu bạn sử dụng các thuộc tính include và exclude trong cùng 1 tag helper thì exclude luôn được ưu tiên. Ví dụ đoạn code dưới đây, môi trường Development là một giá trị quy định trong cả exclude và include. Trong trường hợp này, nội dung sẽ không được render ra cho môi trường Development do danh sách exclude được ghi đè danh sách inlude hoặc danh sách names.

<environment include="Development, Staging" exclude="Development">

<h1>This will **not** show **in** Development **or** Production, but it will show **in** Staging.</h1>

</environment>

trong code thì ta sử dụng IhostingEnviroment

# Cơ chế Model Binding: Truyền dữ liệu từ View lên Controller

Trong cơ chế **Model Binding** của ASP.NET Core chúng ta sẽ học cách làm sao để truyền dữ liệu từ View lên Controller. Chúng ta cũng sẽ tìm hiểu về **Model Binding** và cơ chế hoạt động của nó. ASP.NET Core cho phép chúng ta bind dữ liệu từ nhiều nguồn khác nhau như HTML Form sử dụng **[FromForm]**, từ giá trị route **[FromRoute]**, từ query string **[FromQuery]**, từ  body của request**[FromBody]** và từ Header của request **[FromHeader]**.

Một số bài hướng dẫn trước đây có thể tham khảo:

* [Model và ViewModel trong ASP.NET Core MVC](https://tedu.com.vn/lap-trinh-aspnet-core/model-va-viewmodel-trong-aspnet-core-mvc-245.html)
* [Truyền dữ liệu từ Controller sang View trong ASP.NET Core](https://tedu.com.vn/lap-trinh-aspnet-core/truyen-du-lieu-tu-controller-sang-view-trong-aspnet-core-246.html)
* [Xây dựng HTML Form trong ASP.NET Core](https://tedu.com.vn/lap-trinh-aspnet-core/xay-dung-html-form-trong-aspnet-core-247.html)
* [Strongly Typed View trong ASP.NET Core](https://tedu.com.vn/lap-trinh-aspnet-core/strongly-typed-view-trong-aspnet-core-248.html)
* [Tag Helpers trong ASP.NET Core MVC](https://tedu.com.vn/lap-trinh-aspnet-core/tag-helpers-trong-aspnet-core-mvc-249.html)

## Model Binding là gì?

Model Binding là cơ chế map dữ liệu được gửi qua HTTP Request vào các tham số của action method trong Controller. HTTP Request có thể chứa dữ liệu từ nhiều định dạng. Dữ liệu có thể chứa trong HTML Form. Nó có thể là một phần của route value hoặc trên query string hay có thể là một body của request.

Cơ chế ASP.NET Core model binding cho phép chúng ta dễ dàng bind các giá trị này vào các tham số của action method. Các tham số này có thể là kiểu nguyên thủy hoặc kiểu đối tượng phức tạp.

## Lấy dữ liệu từ Form Data trong Controller

Trong bài [Tag Helper](http://tedu.com.vn/lap-trinh-aspnet-core/tag-helpers-trong-aspnet-core-mvc-249.html), chúng ta đã tạo một form cơ bản cho phép nhận một đối tượng **Product**. Khi người dùng click nút **Submit**thì dữ liệu sẽ được post lên phương thức **Create**trên Controller. Trong project đó chúng ta cũng tạo một **ProductEditModel**class chứa chi tiết của sản phẩm cần được tạo hoặc chỉnh sửa:

**public** **class** **ProductEditModel**

{

**public** **int** ID{ **get**; **set**; }

**public** **string** Name { **get**; **set**; }

**public** **decimal** Rate { **get**; **set**; }

**public** **int** Rating { **get**; **set**; }

}

Một form được tạo chứa 3 field: **Name**, **Rate**và **Rating**:

<form action="/home/Create" method="post">

<**label** for="Name">Name</**label**>

<input type="text" name="Name" />

<**label** for="Rate">Rate</**label**>

<input type="text" name="Rate" />

<**label** for="Rating">Rating</**label**>

<input type="text" name="Rating" />

<**input** type="submit" name="submit" />

</**form**>

Action method **Create**trong **HomeController**:

[HttpPost]

**public** IActionResult **Create**(ProductEditModel model)

{

string message = "";

**if** (ModelState.IsValid)

{

message = "product " + model.Name + " created successfully" ;

}

**else**

{

message = "Failed to create the product. Please try again";

}

**return** Content(message);

}

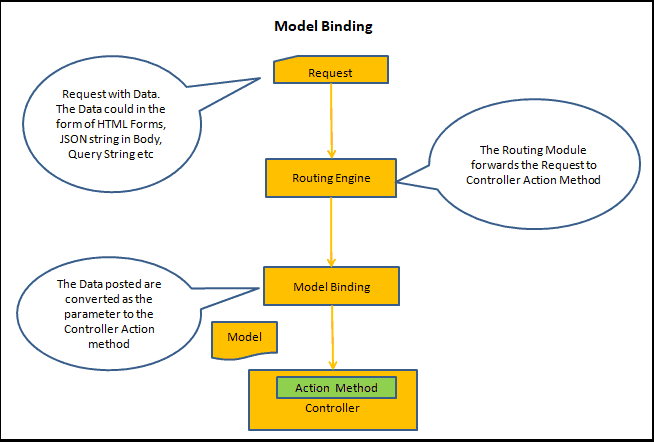
Một submit của form trên, các giá trị trong form sẽ tự động được map vào đối tượng **ProductEditModel**trong Action method của controller:

public IActionResult **Create**(ProductEditModel **model**)

Cơ chế này tự động xảy ra đằng sau được gọi là Model Binding. Model Binder sẽ tìm các trường tương ứng giữa tham số **ProductEditModel**với trường trong form, route value hoặc query string...

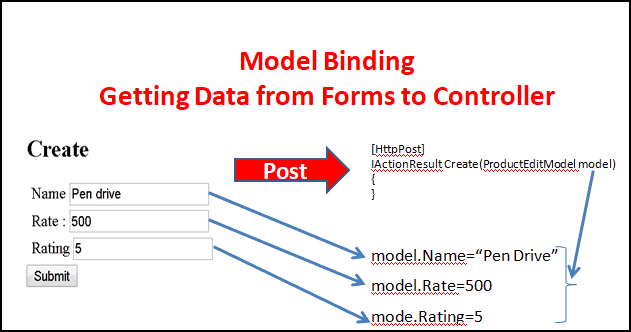
## Cơ chế Model Binding làm việc như thế nào?

Hình dưới đây minh họa cơ chế làm việc của Model binding:



Khi người dùng click vào nút Submit thì một request Post được gửi lên server với Form Data, QueryString, Route Parameter...MVCRouteHandler của Routing Engine sẽ xử lý request đến và có trách nhiệm gọi action method tương ứng. Model Bindler sẽ được kích hoạt trước khi action method được gọi. Nó tìm dữ liệu thỏa mãn trong form data, querys tring và request parameter trong HTTP Request. Sau đso nó sẽ binding các giá trị vào tham số của action method qua tên.

Ví dụ, trường "**name**" trong form sẽ được map vào thuộc tính "**Name**" trong **ProductEditModel**. Rate trong form sẽ được map vào thuộc tính Rate...



Để Model binding làm việc đúng:

* Thuộc tính Name phải match với Request Data
* Cấc thuộc tính phải đặt public set

## Model Binder

Model Binder có trách nhiệm gán dữ liệu vào các tham số của action method. Model Binder được tạo mởi model binder provider. Model binder phải được implement inteface **IModelBinderProvider**. Nghĩa là bạn có thể tạo một Model Binder của riêng mình hoặc mở rộng nó bằng cách triển khai inteface **IModelBinderProvider**. Custom model binder phải được đăng ký trong **ModelBinderProviders**trong **Startup.cs**.

services.AddMvc(options =>

{

options.ModelBinderProviders.Add(**new** CustomModelBinderProvider());

});

### ModelState

Nếu Model binder không thành công trong việc bind dữ liệu từ Request vào thuộc tính model tương ứng, nó sẽ không đưa ra bất cứ thông báo lỗi nào. Nhưng nó sẽ update đối tượng ModelState với danh sách lỗi và set thuộc tính IsValid là false.

Vì thế kiểm tra **ModelState.IsValid** sẽ cho chúng ta thấy quá trình binding có thành công hay không.

Ví dụ: Trong ví dụ trên khi click nút **submit**mà không nhập bất cứ dữ liệu gì trên form thì kết quả sẽ ra validate thất bại và vì thế **ModelState.IsValid** sẽ là **false**.

## Nếu không dùng model binding thì sao?

Trước khi chúng ta tìm hiểu sâu hơn về Model binding, chúng ta cần hiểu nếu không có model binding thì chúng ta sẽ truy cập đến dữ liệu từ request kiểu gì? Xem lại code mà chúng ta đã tạo trong phần trước. Thêm mới action method **NoModelBinding**:

[HttpPost]

**public** IActionResult **NoModelBinding**()

{

ProductEditModel model = **new** ProductEditModel();

string message = "";

model.Name = Request.Form["Name"].ToString();

model.Rate = Convert.ToDecimal( Request.Form["Rate"]);

model.Rating =Convert.ToInt32( Request.Form["Rateing"]);

message = "product " + model.Name + " created successfully";

**return** Content(message);

}

Và thay đổi thành:

<**form** action="/home/NoModelBinding" method="post">

### Truy cập trực tiếp đến query string

Tương tự như thế, bạn có thể truy cập đến các giá trị trên query string sử dụng**Request.Query**. Lấy các giá trị trên query string.

Ví dụ,**Request.Query["id"].ToString()** trả về giá trị của **id**trên query string. Sử dụng **Request.QueryString.HasValue** sẽ cho bạn biết nếu có giá trị query string trên URL hiện tại hay không và **Request.QueryString.Valu**e sẽ trả về giá trị thô của query string.

### Truy cập trực tiếp đến Request Headers

Tương tự như thế, bạn có thể sử dụng **Request.Headers**để truy cập các giá trị được gửi lên thông qua HTTP Header.

### Truy cập đến Route Data

Để truy cập đến route bạn cần ghi đè phương thức OnActionExecuting:

**using** Microsoft.AspNetCore.Mvc.Filters;

**public** **override** **void** **OnActionExecuting**(ActionExecutingContext context)

{

**string** id = context.RouteData.Values["id"].ToString();

**base**.OnActionExecuting(context);

}

Bạn thấy rằng có rất nhiều code để lấy giá trị được post lên HTTP Request. ASP.NET Core model binding sẽ làm giúp bạn các việc này mà dùng ít code hơn.

## Các nguồn cho Model binding

Như đã nhắc đến trước đây, model binder có thể lấy dữ liệu từ rất nhiều nơi khác nhau. Đây là danh sách các nguồn dữ liệu theo thứ tự mà model binding sẽ tìm:

* HTML Form Value
* Route Value
* Query String

 Model binder cũng có thể tìm dữ liệu từ các nguồn sau, nhưng chúng ta cần chỉ ra nguồn nào cần lấy một cách tường minh:

* Request Body
* Request Header
* Services

### Lấy dữ liệu từ Form và Query String

Hãy thử binding action parameter với cả form và query string. Phương thức **FormAndQuery**sẽ như sau:

[HttpGet]

**public** IActionResult **FormAndQuery**()

{

**return** View();

}

[HttpPost]

**public** IActionResult **FormAndQuery**(string name,ProductEditModel model)

{

string message = "";

**if** (ModelState.IsValid)

{

message = "Query string "+ name + " product " + model.Name + " Rate " + model.Rate + " Rating " + model.Rating ;

}

**else**

{

message = "Failed to create the product. Please try again";

}

**return** Content(message);

}

Chú ý rằng action method **FormAndQuery**có hai tham số **name**và **ProductEditModel**.

**public** IActionResult **FormAndQuery**(string name,ProductEditModel model)

Tiếp theo, chúng ta tạo view **FormAndQuery**như sau:

<form action="/home/FormAndQuery/?name=Test" method="post">

<**label** for="Name">Name</**label**>

<input type="text" name="Name" />

<**label** for="Rate">Rate</**label**>

<input type="text" name="Rate" />

<**label** for="Rating">Rating</**label**>

<input type="text" name="Rating" />

<**input** type="submit" name="submit" />

</**form**>

Form có tên trường là "**name**". Chúng ta cũng gửi **"name=test"** qua query string đến controller action:

<**form** action="/home/FormAndQuery/?name=Test" method="post">

Trong ví dụ trên, tham số**"name"** xuất hiện 2 lần như là một phần của query string. Khi form được submit, tham số "name" luôn map đến trường trong form chứ không phải query string. Bởi vì model binder luôn sử dụng thứ tự map dữ liệu theo thứ tự:

1. Form Values
2. Route Values
3. Query Strings

Vì thế các giá trị trong form có trường name, tham số name luôn được gán giá trị. Chúng ta có thể thay đổi hành vi này bằng cách thêm thuộc tính **[FromQuery]**. Sử dụng **[FromQuery]** như sau:

**public** IActionResult **FormAndQuery**([FromQuery] string name,ProductEditModel model)

Giờ nếu submit form thì tham số name sẽ lấy giá trị từ query string, trong khi **ProductEditModel**sẽ lấy giá trị từ form.

## Điều khiển Binding Source

Trong ví dụ trước, chúng ta sử dụng**[FromQuery]** để bắt buộc model binder đổi hành vi mặc định và sử dụng query string làm nguồn cho binding. ASP.NET Core cung cấp cho chúng ta một số thuộc tính điều khiển và chọn nguồn nào sẽ được nhận khi binding:

1. [FromForm]
2. [FromRoute]
3. [FromQuery]
4. [FromBody]
5. [FromHeader]
6. [FromServices]

### [FromForm]

 [FromForm] ép model binder bind tham số vào các trường của HTML Form.

[HttpPost]

**public** **IActionResult** **Create**([FromForm] **ProductEditModel** **model**)

{

}

### [FromRoute]

[FromRoute] ép model binder bind tham số vào route data từ request.

Ví dụ: Tạo một action method FromRoute, cho phép một giá trị **id**và **ProductEditModel**. Chúng ta có 2 tham số id. MVC mặc định sẽ có tham số Id qua route và nó là tuỳ chọn. **ProductEditModel**cũng có thuộc tính id.

[HttpGet]

**public** IActionResult **FromRoute**()

{

**return** View();

}

[HttpPost]

**public** IActionResult **FromRoute**(string id, ProductEditModel model)

{

string message = "";

**if** (ModelState.IsValid)

{

message = "Route " + id + " Product id " + model.id + " product " + model.Name + " Rate " + model.Rate + " Rating " + model.Rating;

}

**else**

{

message = "Failed to create the product. Please try again";

}

**return** Content(message);

}

Tạo ra view **FromRoute**

<form action="/home/FromRoute/Test" method="post">

<**label** for="id">id</**label**>

<input type="text" name="id" />

<**label** for="Name">Name</**label**>

<input type="text" name="Name" />

<**label** for="Rate">Rate</**label**>

<input type="text" name="Rate" />

<**label** for="Rating">Rating</**label**>

<input type="text" name="Rating" />

<**input** type="submit" name="submit" />

</**form**>

Chú ý là chúng ta đang gọi controller action bằng cách sử dụng **"test"** như là giá trị route:

<**form** action="/home/FromRoute/Test" method="post">

Giờ khi submit form thì tham số id luôn map vào id từ form thay vì từ route value. Giờ hãy mở **HomeController**ra và áp dụng [FromRoute] trên tham số **id**.

**public** IActionResult **FromRoute**([FromRoute] string id, ProductEditModel model)

Khi submit form thì id sẽ được nhận là **"test"**

### Binding query string sử dụng [FromQuery]

[FromQuery] ép model binder bind giá trị vào tham số từ giá trị lấy từ query string.

### Binding đến Request body sử dụng [FromBody]

[FromBody] ép model binder bind dữ liệu từ request body. Formatter sẽ chọn dựa trên **content-type** của request. Dữ liệu trong request body có nhiều format khác nhau như JSON, XML...Model binder sẽ tìm thuộc tính **content-type** trên header và chọn **formatter**để chuyển dữ liệu về kiểu mong muốn.

Ví dụ, khi content-type là **'application/json",** model binder sử dụng **JsonInputFormatter**class để chuyển request body và map vào tham số.

### Binding từ Request Header sử dụng [FromHeader]

[FromHeader] map các giá trị từ request header vào tham số của action:

**public** IActionResult **FromHeader**( [FromHeader] string Host)

{

**return** Content(Host);

}

### Vô hiệu hoá binding với [BindNever]

Để cho model binder biết rằng không bind cho thuộc tính nào đó.

[BindNever]

**public** **int** Rating { **get**; **set**; }

Giờ thì model binder sẽ bỏ qua thuộc tính Rating ngay cả khi form có trường Rating.

### Bắt buộc Binding với [BindRequuired]

Cái này ngược hẳn với [BindNever]. Trường nào được đánh dấu là BindRequired phải luôn hiển thị trên form và binding phải thực hiện nếu không thì ModelState.IsValid sẽ false.

# Model Validation

ài viết này chúng ta sẽ học về cơ chế Model Validation. Thường thì dữ liệu được nhập bởi người dùng sẽ không hợp lệ và không thể đưa vào cơ sở dữ liệu. Các dữ liệu được nhập có thể chứa một số lỗi chính tả hoặc người dùng cố tính nhập các dữ liệu không phù hợp. Vì thế chúng ta cần phải kiểm tra dữ liệu người dùng nhập vào trước khi lưu trữ vào cơ sở dữ liệu. ASP.NET Core cung cấp chó chúng ta thành phần gọi là Model Validator, nó dùng các attribute để kiểm tra dữ liệu trong model dễ dàng hơn. Chúng ta cũng tìm hiểu về ModelState và cách sử dụng nó.

## Giới thiệu về Model Validation

Form Data post dữ liệu lên Controller action tự động được map vào các tham số của action bởi Model Binder như chúng ta đã tìm hiểu ở bài trước. Model cần kiểm tra dữ liệu đầu vào xem có hợp lệ không. Quá trình kiểm tra này có thể được hoàn thành bởi client trước khi gửi lên server hoặc server kiểm tra sau khi nhận được từ client. Cơ chế validation phía client (client-side validation) rất quan trọng vì nó giúp tăng trải nghiệm người dùng khi kiểm tra dữ liệu mà không cần chờ đến server nhưng phía server lại nên đảm bảo một lần nữa để các dữ liệu không hợp lệ không thể đưa vào hệ thống.

## Sự quan trọng của client-side validation

* Giúp tăng sự trải nghiệm
* Vì việc kiểm tra tiến hành phía trình duyệt client nên phản hồi nhanh hơn và gần như là ngay lập tức
* Tiết kiệm tài nguyên server như là băng thông bằng cách giảm truy vấn đến server.

## Sự quan trọng của server-side validation

Client-side validation cung cấp trải nghiệm người dùng tốt hơn nhưng không tin cậy. Nó có thể lỗi do một trong các lý do sau:

* Javascript có thể bị tắt ở trình duyệt
* Người dùng có thể gửi trực tiếp dữ liệu đến người dùng mà không sử dụng ứng dụng hoặc sử dụng một số các trình chỉnh sửa request có hại.
* Khi Javascript có lỗi thì kết quả là dữ liệu được đưa vào hệ thống mà có thể không hợp lệ

Vì thế điều quan trọng là kiểm tra dữ liệu phải được thực hiện ở cả phía server, ngay cả bạn đã validate ở phía client.

## Kiểm tra model một cách tường minh

Một khi bạn nhận model trong controller, bạn có thể kiểm tra model đó bằng cách viết code như sau:

**if** (string.IsNullOrEmpty(model.Name))

{

//Validation Failed

//Send the list of errors to client

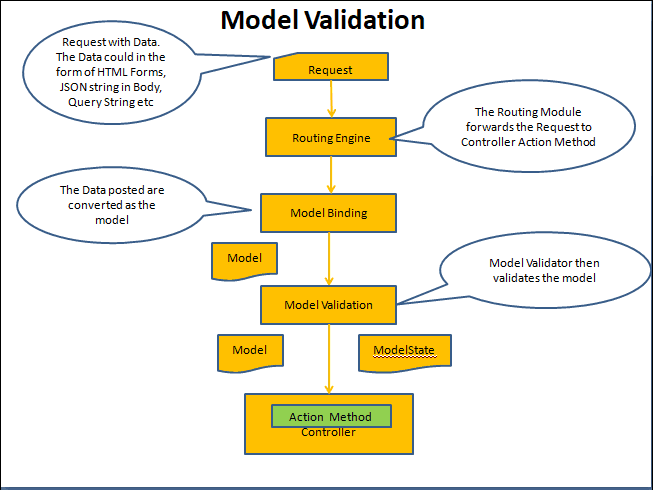
}

Đoạn code trên đơn thuần chỉ kiểm tra xem thuộc tính name của model có rỗng hay null không. Đoạn code trên làm việc tốt, nhưng bạn sẽ phải làm nhiều lần tương tự nếu có nhiều thuộc tính trong model, khi đó bạn cần kiểm tra với các đoạn code giống nhau. Ở đây bạn cần xem làm sao để gửi các báo lỗi về client để hiển thị cho người dùng. Model Validator sẽ làm điều này cho bạn mà không cần viết các đoạn code thừa thãi.

## Model Validation làm việc ra sao?

Chúng đã tìm hiểu cơ chế Model Binding làm việc ra sao trong ASP.NET Core rồi đúng không? Vậy khi HTTP Request được đưa tới Model Binder thì nó sẽ được gọi trước khi truyền tham số vào Controller action. Model Binder sẽ không chỉ map dư liệu vào action method mà nó còn kiểm tra chúng sử dụng Model Validator.

Model Validator chạy sau model binding và chạy một loạt các logic kiểm tra trên mỗi thuộc tính của model dựa trên các attribute bạn đặt cho các thuộc tính đó. Các attribute này gọi là Validation Attribute và chứa code sử dụng bởi Model Validator.



Tất cả các logic kiểm tra đều chạy phía server. ASP.NET Core có nhiều các attribute viết sẵn, bạn có thể thêm nó vào thuộc tính của model để cài đặt quy tắc kiểm tra. Các quy tắc này cũng được gọi là **DataAnnotations**và nằm trong namespace: **System.ComponentModel.DataAnnotations**. Bạn cần import nó vào **ViewModel**và sử dụng.

Ví dụ, bạn đặt trường bắt buộc phải nhập sẽ dùng **[Required]**

[Required]

Public **string** Name {**get**;**set**;}

Model Binder không bắn ra bất cứ lỗi nào nếu quá trình kiểm tra không hợp lệ. Nhưng nó sẽ cập nhật đối tượng **ModelState**với danh sách lỗi và đặt thuộc tính **IsValid**là **false**trước khi gọi action method. Chúng ta cần kiểm tra **ModelState.IsValid** để biết xem quá trình kiểm tra có hợp lệ hay không để action method có thể trả về danh sách lỗi nếu cần.

**if** (ModelState.IsValid)

{

//Model is valid. Call Service layer for further processing of data

} **else** {

//Validation failed. Return the model to the user with the relevant error messages

}

## Cách sử dụng Validation Attributes

### Cập nhật Model với Data Annotaion

Trong Model, thuộc tính thêm vào Data Annotation attribute được hiển thị bên dưới đây. Code sẽ được thêm vào attribute cho thuộc tính **Name**. Nó cũng được cài đặt thông báo trong trường hợp kiểm tra dữ liệu không hợp lệ với attribute đó:

[Required(AllowEmptyStrings =false,ErrorMessage ="Please enter the name")]

[StringLength(maximumLength:25,MinimumLength =10,ErrorMessage ="Length must be between 10 to 25")]

**public** **string** Name { **get**; **set**; }

### Hiển thị danh sách lỗi Validations

Trong view, sử dùng Validation Tag Helper để hiển thị lỗi cho người dùng. Tag helper **asp-validation-summary** giúp hiển thị danh sách các thông báo lỗi cho riêng Model validation trên form. Nó được gắn vào thẻ div và đặt trên cùng của form. **asp-validation-for** hiển thị danh sách lỗi cho thuộc tính **Name**ở phía bên phải của nó. Nó được gắn vào thẻ span và đặt ngay cạnh thuộc tính.

Validation tag helper thêm class field-validation-error và validation-summary-errors vào HTML nếu lỗi tìm thấy. Vì thế chúng a thêm các style CSS vào để hiển thị lỗi màu đỏ cho các class này. Bạn có thể đọc thêm phần Validation Tag Helper ở bài sau:

@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers

@model ASPNetCoreForms.Models.ProductEditModel

@{

ViewData["Title"] = "Create";

}

<**style**>

.field-validation-error {

**color**: red

}

.validation-summary-errors {

**color**: red

}

</**style**>

<**h2**>Create</**h2**>

<**form** action="/home/create" method="post">

<**div** asp-validation-summary="ModelOnly">

<**span**>Please correct the following errors</**span**>

</**div**>

<**label** asp-for="Name">Name</**label**>

<**input** asp-for="Name" />

<**span** asp-validation-for="Name"></**span**>

<**br** />

<**input** type="submit" name="submit" />

</**form**>

### Kiểm tra ModelState.IsValid trong Controller Action

Cuối cùng trong Controller action method chúng ta kiểm tra nếu **ModelState.IsValid** nếu có bất cứ lỗi nào thì chúng ta trả về cho người dùng. Chúng ta trả về model để hiển thị các giá trị được nhập đồng thời trả kèm cả danh sách lỗi được hiển thị bởi Validation tag helper:

[HttpPost]

**public** IActionResult **Create**(ProductEditModel model)

{

string message = "";

**if** (ModelState.IsValid)

{

message = "product " + model.Name + " created successfully";

}

**else**

{

**return** View(model);

}

**return** Content(message);

}

Đoạn code trên sẽ hiển thị như sau:



## ModelState

Model Binder cập nhật đối tượng ModelState với kết quả của Model Binding và validation. ModelState sẽ lưu chi tiết của các giá trị được cập nhật lên model và cả các thông tin lỗi trong quá trình validation xuất hiện trong mỗi thuộc tính. ModelState là thuộc tính của ControllerBase và thuộc kiểu **ModelStateDictionary**.

### Các thuộc tính của ModelState

## Các phương thức của ModelState

### AddModelError

Thêm một message lỗi cụ thể vào danh sách Errors với 1 key.

### Clear

Xoá toàn bộ các key và value trong thể hiện của **ModelStateDictionary**

### ClearValidationState(string)

Xoá phần tử trong **ModelStateDictionary**đúng với key truyền vào

### GetFieldValidationState(string)

Trả về **ModelValidationState**với key truyền vào

### TryAddModelError(string, Exception, ModelMetadata)

Thêm một exception cụ thể vào Errors với một key

## Ví dụ sử dụng ModelState

### Lấy danh sách Errors

Lặp quá danh sách key và lấy danh sách Errors trong ModelState

**foreach** (**var** modelStateKey **in** ModelState.Keys)

{

**var** modelStateVal = ModelState[modelStateKey];

**foreach** (**var** error **in** modelStateVal.Errors)

{

**var** key = modelStateKey;

**var** errorMessage = error.ErrorMessage;

}

}

### Thêm một Custom Error Message

Không phải tất cả các thông báo lỗi đều được đưa ra bởi Model Validator. Ví dụ bạn nhận một Product Model hợp lệ nhưng Product lại tồn tại trong cơ sở dữ liệu. Vậy trường hợp này bạn cần thêm vào một custom error message cho ModelState và gửi nó cho người dùng:

**if** (ModelState.IsValid) {

If (someService.IsProductExists(model)) {

ModelState.AddModelError("", “Product already **exists**”);

**return** View(model);

} **else** {

**return** View(model);

}

}

## Danh sách Validation Attributes

Chúng ta nhìn 2 attribute **Required**và **StringLength**ở ví dụ trước. Namespace **System.ComponentModel.DataAnnotaions** chứa một số các thuộc tính. Đây là danh sách:

### CreditCard

Kiểm tra thuộc tính có định dạng credit card

[CreditCard(ErrorMessage ="Please enter a valid card No")]

**public** **string** creditCard { **get**; **set**; }

### Compare

So sánh giá trị với giá trị của thuộc tính khác

Ví dụ, áp dụng vào trường hợp cần xác nhận mật khẩu. Tham số vào của attribute này là **NewPassword**. Nếu xác nhận mật khẩu không đúng với **NewPassword**thì quá trình kiểm tra không hợp lệ.

[Required(AllowEmptyStrings =false,ErrorMessage ="Please enter a valid password")]

**public** **string** NewPassword { **get**; **set**; }

[Compare(otherProperty:"NewPassword", ErrorMessage ="New & confirm password does not match")]

**public** **string** ConfirmPassword { **get**; **set**; }

Cách khác là bạn có thể so sánh trong một phương thức get để kiểm tra:

[Required(AllowEmptyStrings = false, ErrorMessage = "Please enter a valid password")]

**public** **string** NewPassword { **get**; **set**; }

[Compare(otherProperty: "validatePassword", ErrorMessage = "New & confirm password does not match")]

**public** **string** ConfirmPassword { **get**; **set**; }

**public** **string** validatePassword

{

**get**

{

//Do Some calculations here

**return** **this**.NewPassword;

}

}

### EmailAddress

Kiểm tra thuộc tính có phải định dạng email không

[EmailAddress(ErrorMessage ="Please enter a valid email")]

**public** **string** EmailID {**get**;**set**;}

HTML được tạo ra:

<**label** for="EmailID">EmailID</**label**>

<**input** type="email" data-val="true" data-val-email="Please enter a valid email" id="EmailID" name="EmailID" value="" />

<**span** class="field-validation-valid" data-valmsg-for="EmailID" data-valmsg-replace="true"></**span**>

<**br** />

Như bạn thấy HTML, thuộc tính type của nó sẽ tạo ra là email. Trong trường hợp này thì trình duyệt sẽ hiển thị thông báo lỗi nếu có một email không hợp lệ.

### Phone

Kiểm tra định dạng số điện thoại

[Phone(ErrorMessage = "Please enter a valid Phone No")]

**public** **string** PhoneNo { **get**; **set**; }

<label asp-**for**="PhoneNo"></**label**>

<input asp-**for**="PhoneNo" />

<**span** asp-validation-for="PhoneNo"></**span**>

<**label** for="PhoneNo">PhoneNo</**label**>

<**input** type="tel" data-val="true" data-val-phone="Please enter a valid Phone No" id="PhoneNo" name="PhoneNo" value="" />

<**span** class="field-validation-valid" data-valmsg-for="PhoneNo" data-valmsg-replace="true"></**span**>

<**br** />

### Range

Cho phép kiểm tra xem giá trị có nằm trong một khoảng cho trước không

[Range(minimum:100,maximum:200, ErrorMessage = "Please enter a valid no between 100 & 200")]

**public** **int** Range { **get**; **set**; }

<label asp-**for**="Range"></**label**>

<input asp-**for**="Range" />

<**span** asp-validation-for="Range"></**span**>

<**label** for="Range">Range</**label**>

<**input** type="number" data-val="true" data-val-range="Please enter a valid no between 100 &amp; 200" data-val-range-max="200" data-val-range-min="100" data-val-required="The Range field is required." id="Range" name="Range" value="0" />

<**span** class="field-validation-valid" data-valmsg-for="Range" data-valmsg-replace="true"></**span**>

<**br** />

### RegularExpression

Kiểm tra dữ liệu có khớp với một biểu thức chính quy không

[RegularExpression(pattern: "^Mr\\..\*|^Mrs\\..\*|^Ms\\..\*|^Miss\\..\*", ErrorMessage ="Name must start with Mr./Mrs./Ms./Miss.")]

**public** **string** FullName { **get**; **set**; }

### Required

Yêu cầu một thuộc tính là bắt buộc. Các kiểu không chấp nhận giá trị null (decimal, int, float và DateTime) thường mặc định mà bắt buộc không cần dùng đến attribute này. Ứng dụng sẽ tự động kiểm tra phía server với các kiểu không cho phép null và tự đánh là Required.

### StringLength

Kiểm tra xem chuỗi có nằm trong vùng độ dài cho phép

[StringLength(MaximumLength: 25, MinimumLength = 10, ErrorMessage = "Length must be between 10 to 25")]

**public** **string** Name { **get**; **set**; }

### Url

Kiểm tra xem định dạng URL chuẩn không

[Url(ErrorMessage ="Please enter a valid URL")]

**public** **string** Url { **get**; **set**; }

<label asp-**for**="Url"></**label**>

<input asp-**for**="Url" />

<**span** asp-validation-for="Url"></**span**>

HTML tạo ra là

<**label** for="Url">Url</**label**>

<**input** type="url" data-val="true" data-val-url="Please enter a valid URL" id="Url" name="Url" value="" />

<**span** class="field-validation-valid" data-valmsg-for="Url" data-valmsg-replace="true"></**span**>

<**br** />

# Validation Tag Helper

ASP.NET cung cấp các Tag Helper liên quan đến hiển thị validation message cho người dùng. Chúng ta đã tìm hiểu cách sử dụng server-side model validation trong bài trước. Model Binder sẽ bind và kiểm tra dữ liệu nhận từ HTTP Request. Nó tạo một đối tượng ModelState chứa các danh sách thông báo lỗi được tạo ra bởi Model Binder. Validation Tag Helper sẽ tạo ra các phần tử HTML để hiển thị cho user trên view.

## Validation Tag Helpers

ASP.NET Core cung cấp 2 Tag Helper để hiển thị thông báo lỗi trên client:

1. Validation Message Tag Helper
2. Validation Summary Tag Helper

### Validation Message Tag Helper

Validation Tag Helper hướng đến phần tử <span> và sử dụng nó để thiển thị một thông báo lỗi cho thuộc tính cụ thể.

[Required]

Public **string** name {**get**;**set**;}

<**label** asp-for="Name">Name</**label**>

<**input** asp-for="Name" />

<**span** asp-validation-for="Name"></**span**>

<**br** />

Đoạn HTML sau được tạo ra. Chú ý là phần tử <span> rỗng và class field-validation-valid được gắn cho nó.

<label **for**="Name">Name</label>

<input type="text" data-val="true" data-val-required="The Name field is required." id="Name" name="Name" **value**="" />

<span **class**="field-validation-valid" data-valmsg-**for**="Name" data-valmsg-replace="true"></span>

Và khi bạn gửi form với một trường name bị rỗng. Đoạn HTML được tạo ra chú ý là phần tử <span> giờ đã có thông tin lỗi. Và class field-validation-error được gắn cho nó.

<label **for**="Name">Name</label>

<input type="text" **class**="input-validation-error" data-val="true" data-val-required="The Name field is required." id="Name" name="Name" **value**="" />

<span **class**="field-validation-error" data-valmsg-**for**="Name" data-valmsg-replace="true">The Name field **is** required.</span>

Class dưới đây hiển thị màu đỏ cho class lỗi:

<**style**>

.field-validation-error { **color**:red }

</**style**>

## Validation Summary Tag Helper

Validation summary tag helper hướng đến phần tử <div> và dùng nó để hiển thị tất cả các thông tin lỗi trên form ở một nơi. Mỗi một thông báo lỗi được hiển thị với một danh sách không được sắp xếp. Bạn cần chỉ ra 3 giá trị cho asp-validation-summary tag helper. **All**, **ModelOnly**hoặc **None**.

### All

Tuỳ chọn này hiển thị tất cả các thông báo lỗi

<div asp-validation-summary="All"></**div**>

Nó sẽ tạo ra đoạn HTML dưới đây khi không có lỗi nào được hiển thị

<div **class**="validation-summary-valid" data-valmsg-summary="true">

<**ul**><**li** style="display:none"></**li**></**ul**>

</div>

Và tạo ra HTML khi model lỗi:

<div **class**="validation-summary-errors" data-valmsg-summary="true">

<**ul**><**li**>The Name field is required.</**li**></**ul**>

</div>

Để hiển thị tiêu đề

<**div** asp-validation-summary="All">

<**span**>Please correct the following errors</**span**>

</**div**>

Và thêm style:

.validation-summary-valid **span** { **display**: none; }

### ModelOnly

Tuỳ chọn này hiển thị chỉ các lỗi liên quan đến validation mức Model. Các lỗi ở level thuộc tính sẽ không hiển thị. Sử dụng tuỳ chọn này nếu bạn đang hiển thị các lỗi validation liên quan đến thuộc tính sử dụng asp-validation-for.

<div asp-validation-summary="ModelOnly"></**div**>

Điều lạ nữa là nó không tạo ra bất cứ HTML nào khác biệt, khi **ModelState**hợp lệ không giống như tuỳ chọn **All**.

Bạn có thể thêm lỗi mức độ Model trực tiếp trong Model sử dụng

ModelState.AddModelError(string.Empty, "Errors occurred in the Model");

Đoạn HTML dưới được tạo ra khi lỗi xảy ra:

<div **class**="validation-summary-errors">

<**ul**><**li**>Errors occurred in the Model</**li**></**ul**>

</div>

### None

Tag Helper sẽ không hiển thị bất cứ lỗi nào, cái này giống như không có Tag Helper.

Chú ý là validation-summary-valid class được thêm vào khi model hợp lệ (không phải khi tuỳ chọn "**All**" được chọn) và nó được thay thế bởi validation-summary-errors khi model không hợp lệ.

Thêm các style để thay đổi màu chữ:

<**style**>

.validation-summary-errors { **color**:red }

</**style**>

Bạn có thể hiển thị tiêu đề của thông báo lỗi

<**div** asp-validation-summary="ModelOnly">

<**span**>Please correct the following errors</**span**>

</**div**>

Tiêu đề được hiển thị chỉ khi lỗi được hiển thị.

# Unobtrusive(không phô trương) Client Validation

Trong bài viết này chúng ta sẽ thực hiện validation phía client sử dụng Javascript. Chúng ta đã học cách thực hiện server-side validation ở bài Model Validation. Cơ chế Unobtrusive client-side validation trong ASP.NET Core sử dụng cùng các attribute như với Model Validator.

## Client-side validation

Hãy xem xét một ví dụ, khi bạn cần kiểm tra trường Name không được trống. Chúng ta dễ dàng làm với việc thêm attribute **Required** vào thuộc tính **Name** trong model.

[Required]

Public **string** Name {**get**;**set**;}

Model Validator sẽ thực hiện kiểm tra trên server và yêu cầu một request gửi tới server. Nó sẽ không tốt cho trải nghiệm người dùng vì phải tốn một khoảng thời gian chờ  trước khi người dùng nhận thông báo lỗi từ server. Nó cũng tốn tài nguyên server cho một kiểm tra đơn giản.

Đây là trường hợp đơn giản nhất của client-side validation.

Client side validation có thể được  thực hiện bằng nhiều cách:

1. Sử dụng các thư viện Javascript như Jquery validation và Javascript Unobtrusive
2. Sử dụng HTML 5 Validation
3. Viết Javascript của bạn

## Unobtrusive client-side validation

ASP.NET Core bao gồm các thư viện unobtrusive client-side validation, nó giúp dễ dàng thêm các client-side validation mà không cần viết code nhiều. Trong bài trước chúng ta sử dụng server-side validation, chúng ta cũng tìm hiểu cách sử dụng các attribute Annotation và được xử lý bởi Model Validator để kiểm tra Model. Thư viện unobtrusive client-side cũng sử dụng các attribute đó để kiểm tra các thuộc tính trên phía client. Sự khác nhau là nó sử dụng Javascript thay vì C#.

Unobtrusive (ngầm) validation sử dụng jquery.validate.unobtrusive.js. Thư viện này được xây dựng dựa trên jquery.validate.js, nó sử dụng Jquery. Vì thế chúng ta cần import tất cả nó vào view:

## Cách sử dụng unobtrusive client-side validation

### Load các thư viện Javascript bắt buộc

Đầu tiên chúng ta cần thêm **JQuery**,**jquery.validate** và **jquery.validate.unobtrusive** vào view. Các script này được thêm vào file Layout (\_Layout.cshtml) định nghĩa layout của ứng dụng và sử dụng chung cho tất cả các view.

<**script** src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.2.0.min.js"></**script**>

<**script** src="https://ajax.aspnetcdn.com/ajax/jquery.validate/1.16.0/jquery.validate.min.js"></**script**>

<**script** src="https://ajax.aspnetcdn.com/ajax/jquery.validation.unobtrusive/3.2.6/jquery.validate.unobtrusive.min.js"></**script**>

### Thêm Validation Attribute vào thuộc tính input

Tiếp theo chúng ta cần thêm Validation Attribute vào thuộc tính model, cái này đã học trong bài trước. Các attribute này chúng ta thêm khi cài đặt server-side validation cũng được dùng luôn cho client-side validation. Trong view nhớ chỉ ra file layout sử dụng bằng cách:

@{

Layout = "\_Layout";

}

Model

[Required(AllowEmptyStrings = false, ErrorMessage = "Please enter the name")]

[StringLength(maximumLength: 25, MinimumLength = 10, ErrorMessage = "Length must be between 10 to 25")]

**public** **string** Name { **get**; **set**; }

View

<form asp-controller="home" asp-action="create" method="post">

<**div** asp-validation-summary="ModelOnly">

<**span**>Please correct the following errors</**span**>

</**div**>

<label asp-**for**="Name"></**label**>

<input asp-**for**="Name"/>

<**span** asp-validation-for="Name"></**span**>

<br />

<**input** type="submit" name="submit" />

</**form**>

Controller

[HttpPost]

**public** IActionResult **Create**(ProductEditModel model, [FromQuery] string test)

{

string message = "";

**if** (ModelState.IsValid){

message = "product " + model.Name + " created successfully";

}

**else** {

**return** View(model);

}

**return** Content(message);

}

Chúng ta không sửa bất cứ code server nào, ngoại trừ thêm các thư viện Javascript

## Chúng làm việc ra sao?

Các input tag helper tạo ra HTML như sau:

<**label** for="Name">Name</**label**>

<**input** type="text" data-val="true" data-val-length="Length must be between 10 to 25" data-val-length-max="25" data-val-length-min="10" data-val-required="Please enter the name" id="Name" name="Name" value="" />

<**span** class="field-validation-valid" data-valmsg-for="Name" data-valmsg-replace="true"></**span**>

<**br** />

Bạn có thể thấy các thuộc tính được  thêm vào với ký tự bắt đầu là data-\*. Các thuộc tính data-\* là một phần của HTML 5 cho phép chúng ta thêm các thông tin mở rộng cho thẻ HTML.

Thư viện Javascript unobtrusive đọc các thuộc tính data-val và thực hiện kiểm tra phía client trên trình duyệt khi người dùng submit form. Các validation này hoàn thành trước khi form được gửi qua HTTP Request. Nếu có validation nào lỗi thì request sẽ không được gửi.

Click **F12** và mở cửa sổ **Chrome Developer console** và xem tab **Network**. Bạn có thể thấy không có request nào được gửi khi click nút **Submit** với các dữ liệu không hợp lệ.

# Cơ chế Dependency Injection

Dependency Injection giờ đã trở thành thành phần chính thức mặc định của ASP.NET Core. Nó giúp chúng ta đáp ứng tính chất lỏng lẻo (loosely couple), dễ đọc và bảo trì code. Trong bài viết này chúng ta sẽ học cơ bản về Dependency Injection trong việc xây dựng ứng dụng đơn giản.

## Giới thiệu về Dependency Injection trong ASP.NET Core

Dependency Injection (DI) giờ là một phần của ASP.NET Core. Tất cả các service của framework đều được inject khi chúng ta c ần. Nhưng trước khi đi xa hơn, chúng ta cần hiểu tại sao cần dependency injection. Hãy xem ví dụ sau đây trong controller action method khi chúng ta muốn lấy danh sách sản phẩm ra từ **ProductService**.

**public** **class** **HomeController** : **Controller**

{

**public** IActionResult **Index**()

{

\_productService = **new** ProductService();

**return** View(\_productService.getAll());

}

}

Action method **Index**có một phụ thuộc đến **ProductService**. Vì thế nó tạo ra một thể hiện và gọi phương thức phương thức **GetAll**để lấy danh sách sản phẩm. **ProductService**giờ đã bị gắn chặt (tightly couple) vào phương thức **Index**của **HomeController**.

Nếu chúng ta tạo **BetterProductService**và muốn thay thế **ProductService**cũ thì sao:

**public** IActionResult **Index**()

{

\_productService = **new** BetterProductService();

**return** View(\_productService.getAll());

}

Chúng ta cần làm điều đó với tất cả các Controller, service có sử dụng **ProductService**. Nếu chúng ta muốn sử dụng **TestProductService**và chỉ muốn dùng nó cho mục đích test còn với môi trường Production thì sẽ là **ProductService**. Vậy là không dễ dàng.

Hãy xem xét trường hợp khi **ProductService**phụ thuộc vào service khác, sau đó service đó lại phụ thuộc service khác nữa. Nó không thường xuyên nhưng việc một chuỗi các phụ thuộc như thế chắc chắn có tồn tại trong thực tế.

Depedency Injection sẽ giải quyết tất cả các vấn đề này.

## Depedency Injection là gì?

Dependency Injection (được biệt là DI) là một design pattern khi một đối tượng không được tạo trong các thành phần phụ thuộc vào nó mà yêu cầu nó. Hãy thay đổi **HomeController**một chút:

**public** **class** **HomeController** : **Controller**

{

**private** IProductService \_productService;

**public** **HomeController**(IProductService productService)

{

\_productService = productService;

}

**public** IActionResult **Index**()

{

\_productService = **new** ProductService();

**return** View(\_productService.All());

}

}

Sự khác nhau giữa đoạn code trên và đoạn code này là chúng ta không tạo ra thể hiện của **ProductService**trong Index action method. Chúng ta yêu cầu nó trong constructor của **HomeController**. Vấn đề đã được giải quyết chưa? Ai đó đã tạo thể hiện của **ProductService**và gán nó vào **HomeController**?

Đây là điểm mà ASP.NET Core Dependency Injection framework làm nhiệm vụ của nó. Trách nhiệm của nó là tạo ra thể hiện của ProductService và đối tượng này được gọi là **DI Container** hay **Ioc Container.**

Dependency Injection là một design pattern. Dependency injection framework triển khai design pattern này. Có nhiều framework như Autofac, Unity...bạn có thể sử dụng trong ASP.NET Core.

## DI Container

DI Container là một đối tượng có trách nhiệm tạo các phụ thuộc (**ProductService**) và gán nó cho đối tượng yêu cầu (**HomeController**) nó.

### Làm thế nào DI Container biết đối tượng nào được tạo?

Chúng ta cần cấu hình cho DI Container là class nào bạn muốn tạo. Chúng ta cần đặt trong class **Startup**với phương thức **ConfigureServices**.

**public** **void** **ConfigureServices**(IServiceCollection services)

{

services.AddMvc();

services.AddTransient<IProductService, ProductService>();

}

Dòng thứ 2 sẽ đăng ký **ProductService**với service collection sử dụng phương thức **AddTransient**. Có 2 phương thức khác là **AddSingleton**và **AddScoped**. Có 3 phương thức định nghĩa vòng đời của service mà chúng ta sẽ thảo luận trong bài sau.

## Ví dụ về hệ thống Dependency Injection

Tạo một project ASP.NET Core sử dụng empty template và đặt tên nó là **DependencyInjection**. Thêm HomeController với Index method. Bạn có thể tham khảo các bài viết trước.

### Tạo ViewModel

Tạo một folder **Models**và tạo ViewModel như sau:

**public** **class** **ProductViewModel**

{

**public** **int** Id { **get**; **set**; }

**public** **string** Name { **get**; **internal** **set**; }

}

### Thêm mới service

Tạo một thư mục **Services**và thêm mới một class với tên **ProductService.cs**

**using** DependencyInjection.Models;

**using** System.Collections.Generic;

**namespace** **DependencyInjection.Service**

{

**public** **interface** **IProductService**

{

List<ProductViewModel> **getAll**();

}

**public** **class** **ProductService** : **IProductService**

{

**public** List<ProductViewModel> **getAll**()

{

**return** **new** List<ProductViewModel>

{

**new** ProductViewModel {Id = 1, Name = "Pen Drive" },

**new** ProductViewModel {Id = 2, Name = "Memory Card" },

**new** ProductViewModel {Id = 3, Name = "Mobile Phone" },

**new** ProductViewModel {Id = 4, Name = "Tablet" },

**new** ProductViewModel {Id = 5, Name = "Desktop PC" } ,

};

}

}

}

Đầu tiên chúng ta thêm interface **IProductService**và thêm **ProductService**để triển khai interface này. **ProductService**sẽ trả về một danh sách sản phẩm. Danh sách sản phẩm được fix cứng trong code. Trong thực tế thì nó sẽ lấy từ database.

### Sử dụng Service trong Controller

Giờ hãy mở **HomeController**

**using** DependencyInjection.Service;

**using** Microsoft.AspNetCore.Mvc;

**namespace** **DependencyInjection.Controllers**

{

**public** **class** **HomeController** : **Controller**

{

**private** IProductService \_productService;

**public** **HomeController**(IProductService productService)

{

\_productService = productService;

}

**public** IActionResult **Index**()

{

**return** View(\_productService.getAll());

}

}

}

Constructor của **HomeController**yêu cầu một thể hiện của **ProductService** và lưu trữ nó trong biến local tên là **\_productService**. Phương thức Index gọi view với danh sách sản phẩm được lấy ra từ phương thức **GetAll**của **ProductService**.

### View

View chỉ hiển thị danh sách sản phẩm

@model List<DependencyInjection.Models.ProductViewModel>;

@{

ViewData["Title"] = "Index";

}

<h2>Index</h2>

@foreach (var product **in** Model)

{

<p>@product.Id @product.Name</p>

}

### Đăng ký service

Bước cuối cùng là đăng ký service với Dependency Injection container. Mở **Startup.cs**và đến phương thức **ConfigureServices**. Nơi mà tất cả các service được cấu hình cho DI.

**public** **void** **ConfigureServices**(IServiceCollection services)

{

services.AddMvc();

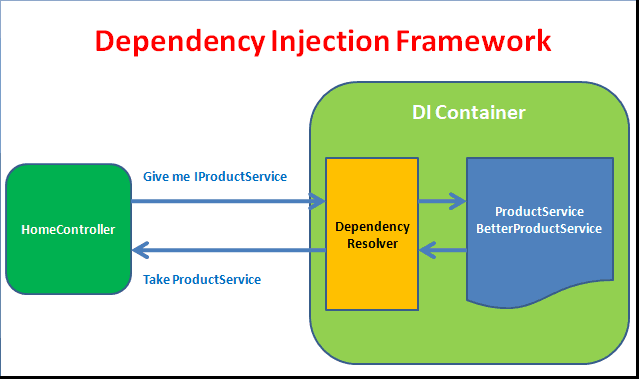
services.AddTransient<IProductService, ProductService>();

}

Giờ chúng ta sẽ đăng ký **ProductService**sử dụng phương thức **AddTransient**. Có 2 phương thức khác là **AddScoped** và **AddSingleton**. Chúng ta sẽ tìm hiểu sau nhé. Giờ hãy chạy ứng dụng và nhìn kết quả.

## Các thành phần phụ thuộc được inject như thế nào?

Hình dưới đây mô tả cách mà **ProductService** được inject vào **HomeController**



Khi một HomeController mới được yêu cầu, MVC sẽ yêu cầu DI framework cung cấp một thể hiện của HomeControlelr class. DI Container sẽ xem constructor của **HomeController**và xác định xem nó có những thành phần phụ thuộc nào (depedencies). Nó sẽ tìm các thành phần phụ thuộc trong danh sách được đăng ký của service collection và tìm service nào thoả mãn sau đó tạo thể hiện cho nó. Sau khi tạo **HomeController**và gán thể hiện của dependencies đó cho constructor.

## Tạo BetterProductService

Giờ chúng ta sẽ tạo mới service khác là **BetterProductService**và muốn sử dụng nó thay vì **ProductService**

**public** **class** **BetterProductService** : **IProductService**

{

**public** List<ProductViewModel> **getAll**()

{

**return** **new** List<ProductViewModel>

{

**new** ProductViewModel {Id = 1, Name = "Television" },

**new** ProductViewModel {Id = 2, Name = "Refrigerator" },

**new** ProductViewModel {Id = 3, Name = "IPhone" },

**new** ProductViewModel {Id = 4, Name = "Laptop" },

};

}

}

Tất cả bạn cần làm là vào **ConfigureServices**trong Startup thay đổi **ProductService**thành **BetterProductService**.

**public** **void** **ConfigureServices**(IServiceCollection services)

{

services.AddMvc();

services.AddTransient<IProductService, BetterProductService>();

}

Bạn không phải vào mỗi controller hay service để thay đổi. Bạn có thể quản lý nó ở một nơi duy nhất. Tương tự bạn có thể làm với **TestProductService**sử dụng chúng để test.

## Thay đổi Service dựa trên environment

Ví dụ, bạn có thể sử dụng môi trường và chuyển đổi giữa các service Trong constructor của **Startup**class yêu cầu **IHostingEnvironment**service:

**IHostingEnvironment** \_env;

**public** Startup(IHostingEnvironment env)

{

**\_env** = env;

}

Tiếp theo trong **ConfigureServices**chúng ta cấu hình code với **BetterProductService**trong **Production**và **ProductService**cho các môi trường khác.

**public** **void** **ConfigureServices**(IServiceCollection services)

{

services.AddMvc();

**if** (\_env.IsProduction())

{

services.AddTransient<IProductService, BetterProductService>();

}

**else**

{

services.AddTransient<IProductService, ProductService>();

}

}

Thay đổi môi trường chọn **Project**, chuột phải và chọn **Properties**. Chọn tab **Debug**. Thay đổi **ASPNETCORE\_ENVIRONMENT**thành môi trường gì mà bạn muốn.

## Constructor Injection và Action Injection

Bạn có thể inject service vào Controller theo 2 cách:

* Constructor Injection
* Action Injection

### Constructor Injection

Khi các thành phần được inject vào thông qua constructor sau đó cách này được gọi là constructor injection

**public** **class** **HomeController** : **Controller**

{

**private** IProductService \_productService;

**public** **HomeController**(IProductService productService)

{

\_productService = productService;

}

**public** IActionResult **Index**()

{

**return** View(\_productService.All());

}

}

### Action Injection

Nếu các dependencies được inject qua một action method thì cách này gọi là Action Injection. Action Injection được thực hiện sử dụng [**FromServices**] attribute vào một tham số của action method. Sử dụng method này nếu service chỉ được sử dụng trong một action method thôi.

**public** **class** **HomeController** : **Controller**

{

**public** **HomeController**()

{

}

**public** IActionResult **Index**( [FromServices] IProductService productService)

{

**return** View(\_productService.All());

}

}

## Lợi ích của Dependency Injection

1. Dependency injection giúp thực hiện kiến trúc lỏng lẻo (loose coupling) trong phần mềm.
2. Code sẽ sạch và dễ đọc hơn
3. Tăng khả năng có thể kiểm thử và bảo trì
4. Cho phép bạn thay đổi triển khai mà không phải thay đổi quá nhiều code

# Agile là gì? Scrum là gì? Các công cụ quản lý dự án theo Agile mà bạn nên biết

Published by

[**Nhat Nguyen**](https://topdev.vn/blog/author/nhat/)

17605

Có rất nhiều phương thức phát triển phần mềm theo quy chuẩn, và một trong số đó là phương thức phát triển phần mềm theo mô hình Scrum ([Scrum là gì](https://topdev.vn/blog/agile-la-gi-scrum-la-gi/#scrum-la-gi)?). Bài viết này sẽ giải thích các khái niệm cơ bản nhất cũng như những giá trị cốt lõi về Agile để bạn có thể nắm chắc được.

## **Agile là gì?**

**Agile**là một phương pháp phát triển phần mềm linh hoạt, là một hướng tiếp cận cụ thể cho việc quản lý dự án phần mềm. Nó gồm một quá trình làm việc tương tác và tích hợp để có thể đưa sản phẩm đến tay người dùng càng nhanh càng tốt.

*Trong các dự án phần mềm, đặc biệt là các dự án chúng ta sẽ gặp rất nhiều khó khăn trong việc thu thập đầy đủ và chính xác các requirements của product để lập plan tốt ngay từ đầu. Có quá nhiều vấn đề gây ảnh hưởng đến việc phát triển phần mềm mà chúng ta không lường trước được. Ví dụ như những vấn đề có thể đến từ những yếu tố như kinh doanh, kỹ thuật, con người, thời gian ra mắt ….*

Xem tiếp...

Những phương pháp phát triển phần mềm theo cách truyền thống ngày càng bộc lộ nhiều nhược điểm và tỷ lệ các dự án thất bại cao trong thời kỳ bùng phát của ngành công nghệ. Nhận ra vấn đề đó, một số cá nhân và công ty riêng lẻ đã đưa ra các phương pháp phát triển phần mềm hiện đại hơn và khác nhau để thích ứng với tình hình mới.

Những phương thức phát triển phần mềm này giúp phần nào giải quyết được một số vấn đề nhưng lại phát sinh vấn đề khác về sự cộng tác, kỹ thuật, công cụ, hướng phát triển, chia sẻ ….

Vào năm 2001, **bản tuyển ngôn Agile (**[**Agile Manifesto**](https://agilemanifesto.org/)**)** đã được thống nhất và ra đời bởi một nhóm người có uy tính trong phát triển phần mềm:

* **Individuals and interactions over processes and tools**: Cá nhân và sự tương tác hơn là quy trình và công cụ
* **Working software over comprehensive documentation**: Phần mềm chạy tốt hơn là tài liệu đầy đủ
* **Customer collaboration over contract negotiation**: Cộng tác với khách hàng hơn là đàm phán hợp đồng
* **Responding to change over following a plan**: Phản hồi với sự thay đổi hơn là bám theo kế hoạch

**[Hoàn thiện và đo vận tốc nhóm Agile Scrum của bạn](https://topdev.vn/blog/?p=7961" \t "_blank)**

### **1. Cá nhân và sự tương tác hơn là quy trình và công cụ**

Đặt trọng tâm vào con người và sự tương hỗ giữa những thành viên trong team. Nếu dự án có những thành viên có năng lực, chịu làm việc cùng nhau thì sẽ mang đến thành công cho dự án.

*Nếu dự án của bạn có quy trình làm việc tốt, được hỗ trợ bởi những công cụ tốt nhất nhưng những thành viên không thống nhất hoặc cùng nhìn về một hướng thì khả năng dự án thất bại là rất lớn. Nói điều này không có nghĩa là phủ nhận tầm quan trọng của quy trình và công cụ nhưng trong Agile nó được đặt sau yếu tố con người.*

Xem tiếp...

**Quy trình** là các thủ tục cần thiết để phát triển dự án như thiết kế, sau đó đến lập trình, rồi kiểm tra QA/QC. Hay để đưa ra một chức năng nào đó cần phải có sự đồng ý của bộ phận QA/QC …. Quy trình này do mỗi công ty quy định và bắt  buộc các nhân viên khi tham gia vào dự án phải tuân thủ.

**Công cụ** là phần mềm được sử dụng trong dự án như : Phần mềm quản lý công việc, phần mềm quản lý source code, phần mềm quản lý lỗi… Có rất nhiều công cụ được sử dụng để hỗ trợ một tổ chức vận hành.

### **2. Phần mềm chạy tốt hơn là tài liệu đầy đủ**

Trong một số [quy trình phát triển phần mềm](https://topdev.vn/blog/quy-trinh-phat-trien-phan-mem/), việc tạo ra và cập nhật các tài liệu về sản phẩm là bắt buộc. Nhóm [lập trình viên](https://topdev.vn/blog/lap-trinh-vien/) không thể hoặc không đồng ý tiến hành công việc nếu không có tài liệu đặc tả về yêu cầu, thiết kế hệ thống.

Nhóm kiểm thử thì yêu cầu tài liệu về sản phẩm để có thể viết trường hợp kiểm thử và kiểm thử được. Nhóm QA đòi tất cả các tài liệu phải được viết trước khi sản phẩm được giao cho khách hàng nếu không thì không đủ điều kiện, chuẩn để giao sản phẩm cho khách hàng.

Việc viết tài liệu thật ra rất mất nhiều thời gian và được cho là rất chán. Ý tưởng ở đây là tại sao mình phải tập trung quá nhiều cho việc không cần thiết mà không dành thời gian đó để trao đổi để hiểu thêm về công việc phải làm. Sau đó đúc kết và chỉ viết những gì mà mọi người cần đọc.

### **3. Cộng tác với khách hàng hơn là đàm phán hợp đồng**

Ta luôn nghe các câu này “Khách hàng là thượng đế” hay “khách hàng luôn luôn đúng”. Tuy nhiên thì khách hàng có nhiều dạng. Cách duy nhất để có thể làm việc tốt là phải cộng tác với khách hàng để hiểu được khách hàng muốn gì và cần gì để có thể tư vấn và điều chỉnh thay vì chỉ dựa vào những điều đã quy định trong hợp đồng.

Trao đổi và thảo luận với khách hàng về sự cần thiết có hay không của một chức năng trong sản phẩm, từ đó quyết định là có nên làm hay không. Tất nhiên để thuyết phục khách hàng thì cần có số liệu nghiên cứu cụ thể chẳng hạn.

### **4. Phản hồi với sự thay đổi hơn là bám theo kế hoạch**

Có một điểm chung là hầu hết những dự án đều có sự thay đổi điều chỉnh khi triển khai. Sự thay đổi đó có thể là thay đổi về requirements, thay đổi tech stack, thay đổi nhân sự, thay đổi deadline, thay đổi phương thức làm việc… mặc dù kế hoạch đã được định ra rõ ràng từ đầu.

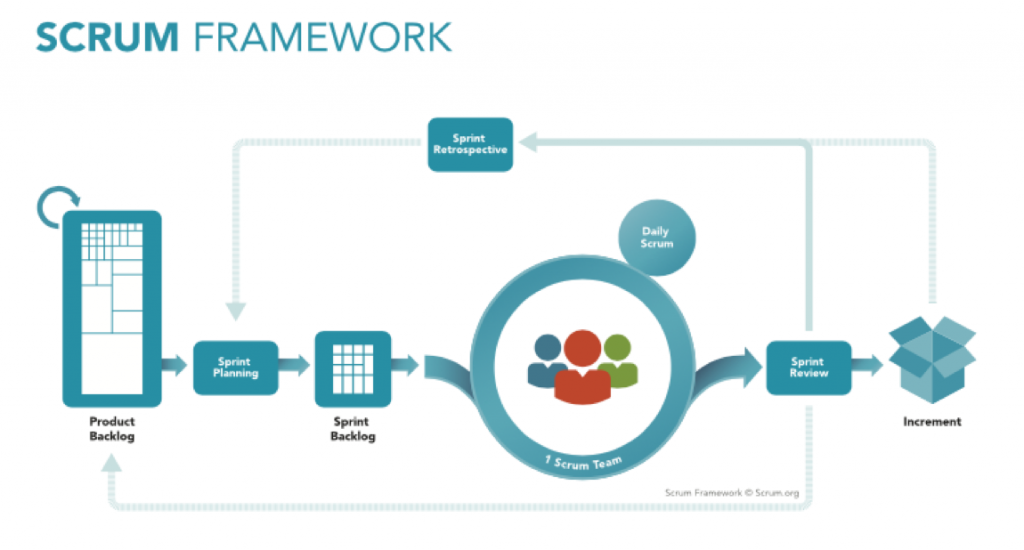
Agile không khuyến khích cho sự thay đổi nhưng khuyến khích chúng ta tập thích nghi với thay đổi.

Một trong các nguyên tắc cơ bản của agile là “phần mềm chạy tốt chính là thước đo của tiến độ”. Nguyên tắc này giúp nhóm dám loại bỏ đi các công việc dư thừa không trực tiếp mang lại giá trị cho sản phẩm.

Để vận hành được cơ chế “làm việc dựa trên giá trị”, nhóm agile thường làm việc trực tiếp và thường xuyên với khách hàng, cộng tác trực tiếp với họ để biết yêu cầu nào có **độ ưu tiên cao hơn**, mang lại giá trị hơn sớm nhất có thể cho dự án. Nhờ đó các dự án agile thường giúp khách hàng tối ưu hóa được giá trị của dự án. Một cách gần như trực tiếp, agile gia tăng đáng kể độ hài lòng của khách hàng.

## **Scrum là gì?**

**Scrum** là một “bộ khung làm việc” cơ bản để tiếp cận những công việc phức tạp. Dựa trên bộ khung này, nhóm làm việc có thể áp dụng những quy trình, kỹ thuật khác nhau cho công việc của mình… Nó là một thành viên của họ [Agile](https://topdev.vn/blog/agile-la-gi/).

Credit: Scrum.org

## **Scrum có ích gì cho phát triển phầm mềm hiện nay**

Nó giúp loại bỏ những công đoạn phức tạp và chỉ tập trung vào những công đoạn cần thiết đáp ứng được nhu cầu của khác hàng đưa ra. Ba yếu tố nòng cốt tạo thành một mô hình quản lý tiến trình thực nghiệm gồm: **sự minh bạch** (transparency), **thanh tra** (inspection) và **thích nghi** (adaptation).

## **Ba giá trị cốt lõi của Scrum**

### **1. Minh bạch**

Muốn áp dụng thành công Scrum, các thông tin liên quan đến quá trình phải mình bạch và thông suốt. Các thông tin có thể là tầm nhìn của sản phẩm, yêu cầu của khách hàng, tiến độ công việc, các rào cản khác…

Từ đó mọi thành viên ở vai trò khác nhau có đầy đủ thông tin cần có để tiến hành quyết định trong việc nâng cao hiệu quả công việc.

### **2. Thanh tra**

Phải thường xuyên thanh tra các hoạt động trong Scrum và tiến độ đến đích để phát hiện các bất thường không theo ý muốn. Tần suất thanh tra không nên quá dày để khỏi ảnh hưởng đến công việc. Công tác thanh tra khi được thực hiện bởi người có kĩ năng tại các điểm quan trọng của công việc sẽ giúp cải tiến liên tục trong Scrum.

### **3. Thích nghi**

Scrum mang lợi thế là tính linh hoạt rất cao, nhờ đó mang lại tính thích nghi cao. Dựa vào thông tin liên tục và minh bạch từ quá trình thanh tra và làm việc, Scrum có thể cho lại các thay đổi tích cực, nhờ đó mang lại thành công cho dự án.

## **Lợi ích mà Scrum mang lại**

Tính minh bạch, kiểm tra, và thích nghi là 3 nền tảng cơ bản của Scrum. Và dưới đây là những lý do tại sao nên dùng Scrum.

1. Cải thiện chất lượng phần mềm, dễ học và dễ sử dụng.
2. Rút ngắn thời gian phát hành phần mềm, cho phép khách hàng sử dụng sản phẩm sớm hơn.
3. Nâng cao tinh thần đồng đội, tối ưu hóa hiệu quả và nỗ lực của đội phát triển.
4. Gia tăng tỷ suất hoàn vốn đầu tư (ROI)
5. Tăng mức độ hài lòng của khách hàng
6. Kiểm soát dự án tốt, cải tiến liên tục
7. Giảm thiểu rủi ro khi xây dựng sản phẩm

## **Các khái niệm cơ bản Scrum**

### **1. Scrum Team**

Scrum team chia làm 3 vai trò bao gồm những thành phần sau:

* **Product Owner**: Nhiệm vụ của Product Owner là đảm bảo việc quản lý những công việc còn tồn đọng (**Product backlog**) của việc phát triển sản phẩm phần mềm. Product Owner phải liên tục cập nhật thông tin cho các thành viên trong team để họ hiểu về yêu cầu hay các tính năng cần có của sản phẩm ngay cả khi họ không trực tiếp phát triển tính năng đó.
* **Development Team**: là những [lập trình viên](https://topdev.vn/blog/lap-trinh-vien/) sẽ tham gia vào việc phát triển từng tính năng cụ thể. Các lập trình viên này có thể sẽ có kỹ năng khác nhau và một số sẽ giỏi về những kỹ năng nhất định. Tuy nhiên khi sử dụng Scrum thì tất cả các thành viên của Development Team yêu cầu phải có khả năng làm việc thay thế vị trí của nhau và không ai chỉ chịu trách nhiệm phát triển một (hoặc một số) tính năng nhất định.
* **Scrum Master**: sẽ chịu trách nhiệm cho việc lên kế hoạch để phân công công việc, sắp xếp thứ tự ưu tiên giải quyết những công việc tồn đọng nào có trong **Backlog** trước, tổ chức các buổi họp với Product Owner để theo dõi tình hình và nắm thông tin cần thiết.

### **2. Sprint**

**Sprint**là mộ phân đoạn lặp đi lặp lại trong quy trình phát triển phần mềm, có khung thời gian thường là 1 tháng (từ 1 – 4 tuần) mà theo đó sản phẩm sẽ được release phiên bản mới. Khi một Sprint kết thúc thì Scrum Master cần phải chuyển trạng thái của nó sang Done.

Khi bắt đầu một Sprint thì Scrum Master cần đưa ra mục tiêu của Sprint đó và mục tiêu này không được phép thay đổi cho tới khi Sprint hoàn thành. Tuy nhiên Product Owner vẫn có quyền huỷ một Sprint trước thời hạn kết thúc của nó.

Mặc dù để làm điều này thì Product Owner cần sự đồng thuận của Development Team cũng như Scrum Master. Sau khi một Sprint kết thúc thì các bên sẽ dựa trên kết quả của Sprint đó để lên kế hoạch cho Sprint tiếp theo.

### **3. Sprint Planning**

Đây là bước đầu tiên cần phải thực hiện trước khi một Sprint bắt đầu. Development team họp với Product Owner để lên kế hoạch cho một sprint. Những công việc nào cần phải được hoàn thành trong Sprint này và làm sao để có thể hoàn thành những công việc này.

Sau khi thống nhất được số lượng công việc, thời gian hoàn thành thì chúng ta có thể bắt đầu Sprint. Trong khi thực hiện một Sprint chúng ta sẽ phải có những buổi họp được gọi là Daily Sprint hay Daily Meeting.

### **4. Daily Sprint**

Các buổi họp Daily Sprint thường kéo dài khoản 15 phút, trong buổi họp này tất cả các thành viên sẽ lần lượt báo cáo lại:

* Những gì họ đã làm được ngày hôm qua
* Những gì họ cần làm ngày hôm nay
* Những khó khăn mà họ gặp phải

Mỗi buổi họp này sẽ giúp việc dự kiến được kế hoạch đưa ra trong Sprint đang làm sẽ tiến triển ra sao và liệu có cần phải cập nhật lại bản kế hoạch đã đưa ra hay không. Tất nhiên cần nhớ rằng việc thay đổi kế hoạch này không bao gồm thay đổi mục tiêu đã đưa ra của Sprint.

Ví dụ bạn có thể tăng thêm thời gian để hoàn thành một chức năng và qua đó khiến Sprint phải kéo dài hơn dự kiến. Tuy nhiên mục tiêu của Sprint là cho phát hành một phiên bản mới cần được giữ nguyên.

### **5. Sprint Review**

Là công việc được thực hiện bởi nhóm phát triển và product owner ở cuối mối Sprint nhằm đánh giá lại kết quả thực hiện được. Từ lúc Sprint mới hoàn thành và qua đó đưa ra những chỉnh sửa, thay đổi cần thiết ở Sprint sau.

### **6. Sprint Restrospective**

Dưới sự trợ giúp của Scrum master, team phát triển sẽ tổng kết những kiến nghị và đánh giá từ bước Sprint Review ở trên để đưa ra những cải tiến nhằm nâng cao hiệu quả làm việc cũng như sản phẩm.

**[Những điểm khác biệt giữa Kanban và Scrum](https://topdev.vn/blog/?p=869" \t "_blank)**

**[Những điểm khác biệt giữa Kanban và Scrum](https://topdev.vn/blog/?p=869" \t "_blank)**

### **7. Các công cụ (artifacts) Scrum**

Scrum sử dụng các công cụ rất đơn giản nhưng hiệu quả để trợ giúp công việc.

**Product backlog**

Đây là danh sách ưu tiên các tính năng (feature) hoặc đầu ra khác của dự án. Có thể hiểu như là danh sách yêu cầu (requirement) của dự án.

Product Owner chịu trách nhiệm sắp xếp độ ưu tiên cho từng hạng mục (Product Backlog Item) trong Product Backlog dựa trên các giá trị do Product Owner định nghĩa (thường là giá trị thương mại – business value).

**Sprint backlog**

Đây là bản kế hoạch cho một Sprint; là kết quả của buổi họp lập kế hoạch (Sprint Planning).

Với sự kết hợp của Product Owner, nhóm sẽ phân tích các yêu cầu theo độ ưu tiên từ cao xuống thấp để hiện thực hóa các hạng mục trong Product Backlog dưới dạng danh sách công việc (TODO list).

**Burndown Chart**

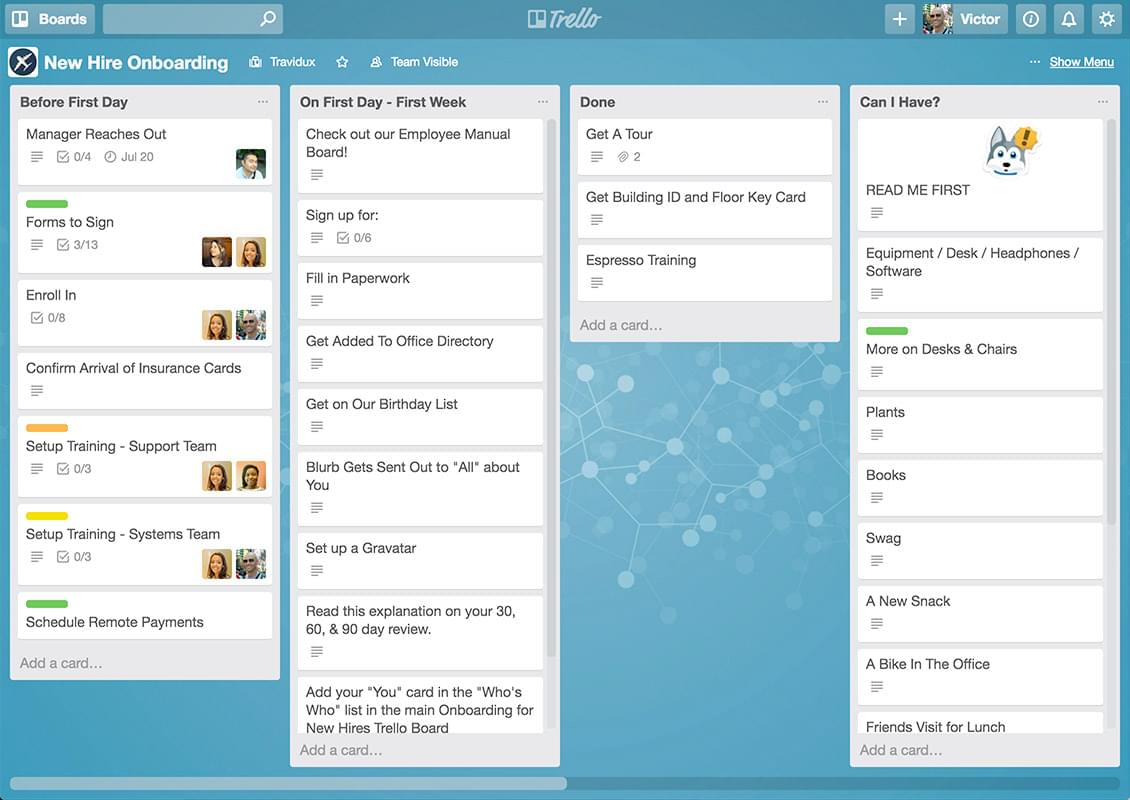
Đây là biểu đồ hiển thị xu hướng của dự án dựa trên lượng thời gian cần thiết còn lại để hoàn tất công việc.

Burndown Chart có thể được dùng để theo dõi tiến độ của Sprint (được gọi là Sprint Burndown Chart) hoặc của cả dự án (Project Burndown Chart).

Biểu đồ burndown không phải là một thành tố tiêu chuẩn của Scrum theo định nghĩa mới, nhưng vẫn được sử dụng rộng rãi do tính hữu ích của nó.

## **Các công cụ quản lý dự án theo Agile mà bạn nên biết**

### [Trello](https://trello.com/)

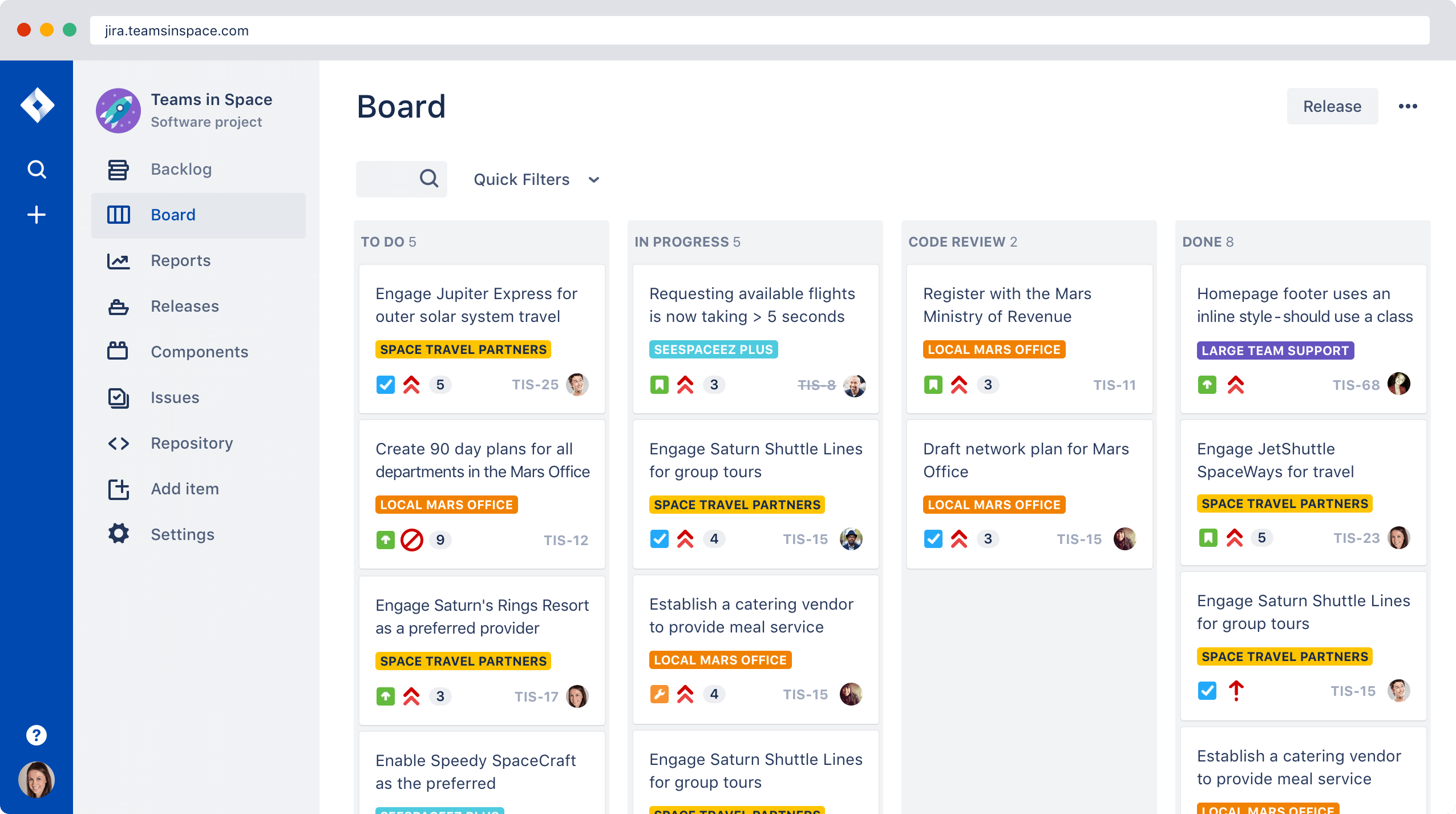


Đây là một trong những ứng dụng quản lý dự án nổi tiếng và được sử dụng nhiều nhất. Nó có cả tài khoản miễn phí và cao cấp mang đến cho bạn cơ hội tuyệt vời để sử dụng hầu hết các chức năng phổ biến.

Cấu trúc của Trello dựa trên phương pháp kanban. Tất cả các dự án được đại diện bởi các bảng, có chứa danh sách. Mọi danh sách đều có các thẻ lũy tiến mà bạn được tạo dưới dạng kéo và thả. Người dùng có liên quan đến bảng, có thể được gán cho thẻ.

Tóm lại, nó có nhiều tính năng hay, nhỏ nhưng không kém phần hữu ích: viết bình luận, chèn tệp đính kèm, ghi chú, ngày đáo hạn, danh sách kiểm tra, nhãn màu, tích hợp với các ứng dụng khác, v.v. Ngoài ra, Trello được hỗ trợ bởi tất cả các nền tảng di động. Trello là công cụ có thể được sử dụng cho cả công việc và các quy trình cá nhân.

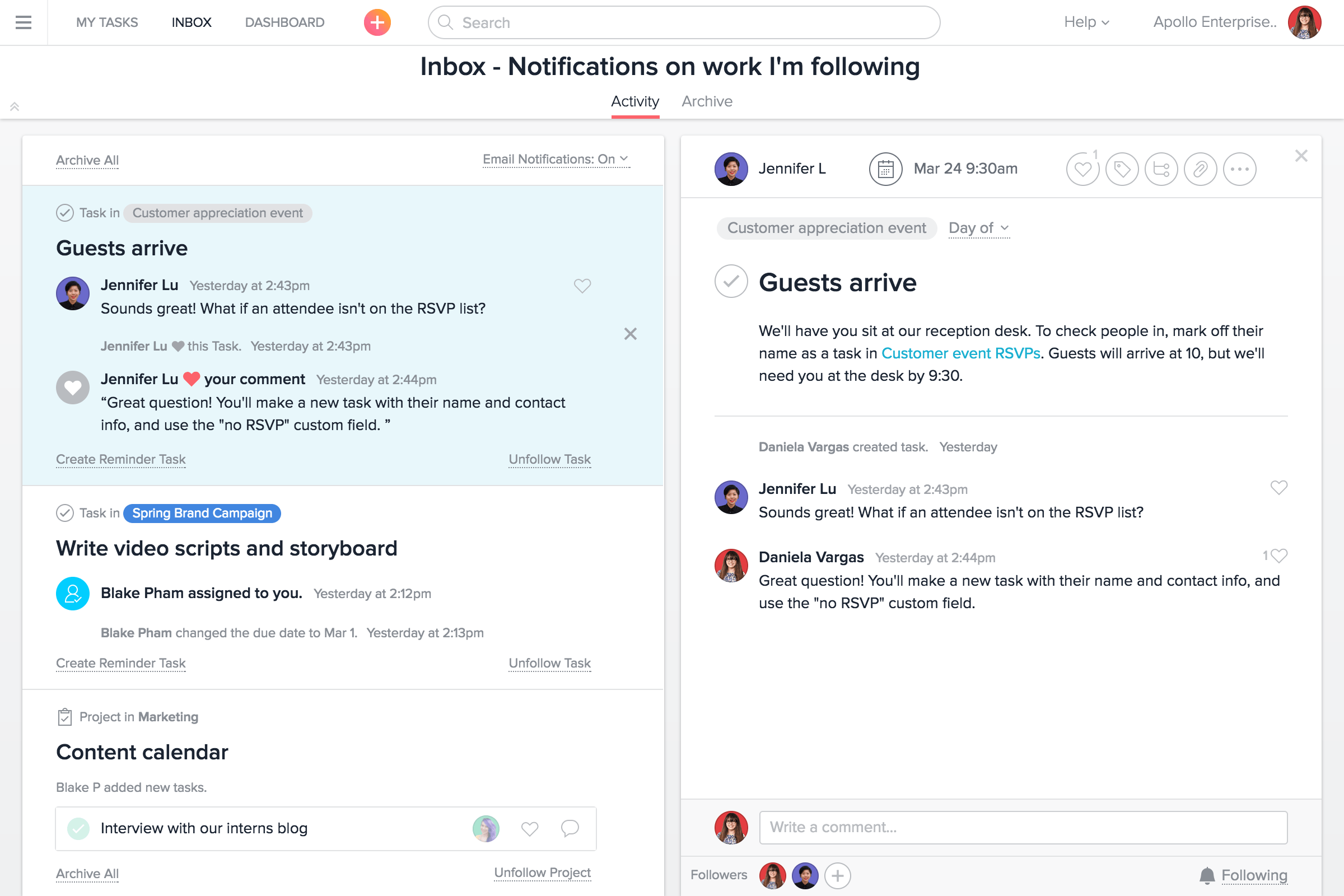
### [JIRA](https://jira.atlassian.com/secure/Dashboard.jspa)



JIRA là một công cụ được phát triển để theo dõi lỗi, theo dõi vấn đề và quản lý dự án cho các quy trình phát triển phần mềm và di động. Bảng điều khiển JIRA có nhiều chức năng & tính năng hữu ích có thể xử lý các vấn đề khác nhau một cách dễ dàng.

Một số tính năng và sự cố chính: loại sự cố, quy trình làm việc, màn hình, trường, thuộc tính vấn đề. Một số tính năng bạn sẽ không tìm thấy ở nơi khác. Bảng điều khiển trên JIRA có thể được tùy chỉnh để phù hợp với quy trình kinh doanh của bạn.

### [Asana](https://asana.com/)



Asana là công cụ quản lý công việc cho phép các nhóm chia sẻ, lập kế hoạch, tổ chức và theo dõi tiến trình của các nhiệm vụ mà mỗi thành viên đang thực hiện. Nó đơn giản, dễ sử dụng và miễn phí cho tối đa 30 người dùng trong một nhóm.

Như tất cả các nền tảng phần mềm quản lý dự án Agile trước đây với mục tiêu chính là cho phép quản lý các dự án và nhiệm vụ. Điều đáng chú ý là bạn không cần phải có email để sử dụng Asana. Mỗi nhóm có thể tạo nơi làm việc sẽ chứa các dự án và nhiệm vụ của dự án: mỗi tác vụ có thể có ghi chú, nhận xét, tệp đính kèm và thẻ.

Công cụ này có thể được sử dụng cho các quy trình nhỏ và cho các quy trình lớn mà không có bất kỳ giới hạn nào trong các ngành hoặc bộ phận.

## **Resource cho bạn tìm hiểu về Agile và Scrum:**

* [Việc làm Agile](https://topdev.vn/viec-lam-it?q=Product+Manager): Có rất nhiều tại TopDev với mức lương cực hấp dẫn.
* [Scrum.org](https://www.scrum.org/): đầy đủ kiến thức cơ bản, nâng cao về Scrum và các chứng chỉ Scrum.
* [Agile Manifesto](http://agilemanifesto.org/): cơ bản về Agile, tuyên ngôn Agile cho người mới bắt đầu.
* [Agile Vietnam Group](https://www.facebook.com/groups/agilevietnam/) và [Agile forum Vietnam](http://www.agilevietnam.org/): diễn đàn lớn nhất về Agile tại Việt Nam, cùng chia sẻ thông tin, kiến thức, sự kiện về Agile.

## **Tổng kết**

Bạn cần lưu ý rõ điều này Agile không phải là “một phương pháp” mà là tư duy, cách tiếp cận, là tập hợp những phương pháp, sự thực hành dựa trên những giá trị và nguyên tắc nêu ra trong bản tuyên ngôn Agile.