Tài liệu tham khảo ASP.Net Core 8+

Nội dung chính

[**1. HTTP [MVC so với API]** 3](#_Toc189428897)

[**2. Phần mềm trung gian [MVC vs API]** 21](#_Toc189428898)

[**3. Định tuyến [MVC vs API]** 37](#_Toc189428899)

[**4. Bộ điều khiển và IActionResult [MVC]** 59](#_Toc189428900)

[**5. Liên kết và xác thực mô hình [MVC vs API]** 82](#_Toc189428901)

[**6. Chế độ xem dao cạo [MVC]** 119](#_Toc189428902)

[**7. Chế độ xem bố cục [MVC]** 156](#_Toc189428903)

[**8. Chế độ xem một phần [MVC]** 170](#_Toc189428904)

[**9. Xem các thành phần [MVC]** 181](#_Toc189428905)

[**10. DI [MVC so với API]** 195](#_Toc189428906)

[**11. Môi trường [MVC vs API]** 223](#_Toc189428907)

[**12. Cấu hình [MVC vs API]** 239](#_Toc189428908)

[**13. xUnit [MVC so với API]** 270](#_Toc189428909)

[**14. Hoạt động CRUD [MVC]** 270](#_Toc189428910)

[**15. Trình trợ giúp thẻ [MVC]** 270](#_Toc189428911)

[**16. Entity Framework core [MVC vs API]** 270](#_Toc189428912)

[**17. Kiểm thử đơn vị nâng cao - Moq và mẫu kho lưu trữ [MVC vs API]** 270](#_Toc189428913)

[**18. Ghi nhật ký và Serilog [MVC vs API]** 271](#_Toc189428914)

[**19. Bộ lọc [MVC so với API]** 271](#_Toc189428915)

[**20. Xử lý lỗi [MVC vs API]** 271](#_Toc189428916)

[**21. Nguyên tắc SOLID [MVC vs API]** 271](#_Toc189428917)

[**22. Kiến trúc sạch [MVC vs API]** 271](#_Toc189428918)

[**23. Danh tính, Ủy quyền, Bảo mật [MVC vs API]** 271](#_Toc189428919)

[**24. Asp.net API Web cốt lõi [API]** 271](#_Toc189428920)

[**25. Swagger và OpenAPI [API]** 271](#_Toc189428921)

[**26. Angular và CORS [API]** 271](#_Toc189428922)

[**27. Xác thực JWT và Web API [API]** 271](#_Toc189428923)

[**28. API tối thiểu [API]** 271](#_Toc189428924)

# **1. HTTP [MVC so với API]**

**Tổng quan:**

* HTTP (Giao thức truyền siêu văn bản) là một giao thức được sử dụng để truyền siêu văn bản (ví dụ: HTML) qua internet.
* Nó hoạt động trên mô hình máy khách-máy chủ, trong đó máy khách (thường là trình duyệt web) đưa ra yêu cầu đến máy chủ, sau đó phản hồi với các tài nguyên được yêu cầu hoặc thông báo lỗi.
* **Giao thức không trạng thái (Trạng thái):** Mỗi yêu cầu HTTP độc lập với các yêu cầu khác; máy chủ không giữ lại thông tin từ các yêu cầu trước đó.

**Mô hình yêu cầu / phản hồi:**

* **Yêu cầu máy khách**: Máy khách gửi yêu cầu HTTP đến máy chủ.
* **Phản hồi máy chủ**: Máy chủ xử lý yêu cầu và gửi lại phản hồi HTTP.

Máy chủ HTTP

**Định nghĩa:**

* Máy chủ HTTP là phần mềm xử lý các yêu cầu HTTP từ máy khách và phục vụ phản hồi. Nó xử lý các yêu cầu đến, thực hiện logic cần thiết (ví dụ: truy cập cơ sở dữ liệu, tạo HTML) và trả về phản hồi thích hợp.

**Ví dụ:**

* Apache HTTP Server, Nginx, Microsoft IIS, Kestrel (được sử dụng với ASP.NET Core).

**Chim cắt:**

* Kestrel là một máy chủ web đa nền tảng đi kèm với ASP.NET Core.
* Nó nhẹ, hiệu suất cao và phù hợp để chạy cả các ứng dụng web nội bộ và công cộng.

Quy trình yêu cầu và phản hồi với Kestrel

1. **Khách hàng gửi yêu cầu:**
   * Máy khách (ví dụ: trình duyệt web) gửi yêu cầu HTTP đến máy chủ.
2. **Kestrel nhận được yêu cầu:**
   * Kestrel nhận yêu cầu và chuyển nó qua quy trình phần mềm trung gian ASP.NET Core.
3. **Xử lý yêu cầu:**
   * Các thành phần phần mềm trung gian xử lý yêu cầu và cuối cùng chuyển nó đến logic xử lý yêu cầu của ứng dụng.
4. **Tạo phản hồi:**
   * Ứng dụng tạo phản hồi HTTP và gửi lại thông qua đường ống phần mềm trung gian.
5. **Kestrel gửi phản hồi:**
   * Kestrel gửi phản hồi HTTP trở lại máy khách.

Cách trình duyệt sử dụng HTTP

* Trình duyệt sử dụng HTTP để yêu cầu các tài nguyên như tài liệu HTML, hình ảnh, tệp CSS và tệp JavaScript từ máy chủ.
* Khi người dùng nhập URL hoặc nhấp vào liên kết, trình duyệt sẽ gửi yêu cầu HTTP đến máy chủ, sau đó máy chủ này phản hồi bằng tài nguyên được yêu cầu.

Quan sát các yêu cầu và phản hồi HTTP trong Chrome Dev Tools

1. **Mở Công cụ dành cho nhà phát triển Chrome:**
   * Nhấn F12 hoặc Ctrl+Shift+I (hoặc Cmd+Option+I trên máy Mac) để mở Chrome Dev Tools.
2. **Điều hướng đến tab Mạng:**
   * Nhấp vào tab Mạng để xem các yêu cầu và phản hồi HTTP.
3. **Kiểm tra yêu cầu:**
   * Nhấp vào bất kỳ yêu cầu nào trong danh sách để xem thông tin chi tiết:
     + **Tiêu đề**: Xem tiêu đề yêu cầu và phản hồi.
     + **Xem trước/Phản hồi**: Xem nội dung phản hồi.
     + **Thời gian**: Xem chi tiết thời gian của yêu cầu.

Định dạng thông báo phản hồi HTTP

**Định dạng tin nhắn phản hồi:**

* **Dòng bắt đầu**: Chứa phiên bản HTTP, mã trạng thái và thông báo trạng thái.
* **Tiêu đề**: Các cặp khóa-giá trị cung cấp thông tin về phản hồi.
* **Nội dung**: Không bắt buộc, chứa dữ liệu thực tế (ví dụ: HTML, JSON).

**Ví dụ:**

1. HTTP/1.1 200 OK
2. Loại nội dung: văn bản / html
3. Độ dài nội dung: 137
5. <html>
6. <thân>
7. <h1>Xin chào, thế giới!</h1>
8. < / cơ thể>
9. </html>

**Tiêu đề phản hồi thường được sử dụng:**

* Content-Type: Chỉ định loại phương tiện của tài nguyên.
* Content-Length: Kích thước của nội dung phản hồi tính bằng byte.
* Máy chủ: Cung cấp thông tin về máy chủ xử lý yêu cầu.
* Set-Cookie: Đặt cookie được lưu trữ bởi khách hàng.
* Cache-Control: Chỉ thị cho các cơ chế bộ nhớ đệm trong cả yêu cầu và phản hồi.

Tiêu đề phản hồi mặc định trong Kestrel

* Content-Type: Thường mặc định là text/html hoặc application/json tùy thuộc vào nội dung được phân phối.
* Máy chủ: Cho biết phần mềm máy chủ (ví dụ: Kestrel).
* Ngày: Ngày và giờ khi phản hồi được tạo.

Mã trạng thái HTTP

**Tổng quan:**

* Mã trạng thái được máy chủ phát hành để đáp ứng yêu cầu của khách hàng để chỉ ra kết quả của yêu cầu.
* Các danh mục bao gồm:
  + **1xx Thông tin**: Yêu cầu đã nhận, tiếp tục quá trình.
  + **2xx Thành công**: Yêu cầu đã được tiếp nhận, hiểu và chấp nhận thành công.
  + **Chuyển hướng 3xx**: Cần thực hiện thêm hành động để hoàn thành yêu cầu.
  + **Lỗi máy khách 4xx**: Yêu cầu chứa cú pháp sai hoặc không thể thực hiện được.
  + **Lỗi máy chủ 5xx**: Máy chủ không thực hiện được yêu cầu hợp lệ.

**Mã trạng thái phổ biến:**

* 200 OK: Yêu cầu thành công.
* 201 Đã tạo: Yêu cầu thành công và một tài nguyên mới đã được tạo.
* 204 Không có nội dung: Máy chủ đã xử lý thành công yêu cầu, nhưng không trả về bất kỳ nội dung nào.
* 400 Yêu cầu không tốt: Máy chủ không thể hiểu yêu cầu do cú pháp không hợp lệ.
* 401 Không được phép: Cần có xác thực.
* 403 Bị cấm: Khách hàng không có quyền truy cập vào nội dung.
* 404 Không tìm thấy: Máy chủ không thể tìm thấy tài nguyên được yêu cầu.
* Lỗi máy chủ nội bộ 500: Máy chủ gặp phải tình trạng không mong muốn.
* 502 Cổng xấu: Máy chủ đang hoạt động như một cổng hoặc proxy và nhận được phản hồi không hợp lệ từ máy chủ ngược dòng.
* 503 Dịch vụ không khả dụng: Máy chủ chưa sẵn sàng để xử lý yêu cầu.

Đặt mã trạng thái và tiêu đề phản hồi trong ASP.NET Core

**Mã ví dụ 1:**

1. var builder = WebApplication.CreateBuilder(args);
2. var app = trình tạo. Xây dựng();
4. App. Run(async (ngữ cảnh HttpContext) =>
5. {
6. ngữ cảnh. Response.Headers["MyKey"] = "giá trị của tôi";
7. ngữ cảnh. Response.Headers["Máy chủ"] = "Máy chủ của tôi";
8. ngữ cảnh. Response.Headers["Content-Type"] = "văn bản/html";
9. chờ ngữ cảnh. Response.WriteAsync("<h1>Xin chào</h1>");
10. chờ ngữ cảnh. Response.WriteAsync("<h2>World</h2>");
11. });
13. App. Chạy();

**Lời giải thích:**

* ngữ cảnh. Response.Headers["MyKey"] = "giá trị của tôi";: Thêm tiêu đề tùy chỉnh vào phản hồi.
* ngữ cảnh. Response.Headers["Server"] = "My server";: Sửa đổi tiêu đề Server.
* ngữ cảnh. Response.Headers["Content-Type"] = "text/html";: Đặt tiêu đề Content-Type thành text/html.
* chờ ngữ cảnh. Response.WriteAsync("<h1>Hello</h1>");: Ghi phần đầu tiên của nội dung phản hồi.
* chờ ngữ cảnh. Response.WriteAsync("<h2>World</h2>");: Ghi phần thứ hai của nội dung phản hồi.

**Mã ví dụ 2:**

1. csharpCopy trình tạo codevar = WebApplication.CreateBuilder(args);
2. var app = trình tạo. Xây dựng();
4. App. Run(async (ngữ cảnh HttpContext) =>
5. {
6. nếu (1 == 1)
7. {
8. ngữ cảnh. Response.StatusCode = 200;
9. }
10. khác
11. {
12. ngữ cảnh. Response.StatusCode = 400;
13. }
14. chờ ngữ cảnh. Response.WriteAsync("Xin chào");
15. chờ ngữ cảnh. Response.WriteAsync(" thế giới");
16. });
18. App. Chạy();

**Lời giải thích:**

* ngữ cảnh. Response.StatusCode = 200;: Đặt mã trạng thái thành 200 OK.
* ngữ cảnh. Response.StatusCode = 400;: Đặt mã trạng thái thành 400 Yêu cầu không tốt (dòng này sẽ không được thực thi do điều kiện).
* chờ ngữ cảnh. Response.WriteAsync("Hello");: Viết phần đầu tiên của nội dung phản hồi.
* chờ ngữ cảnh. Response.WriteAsync(" World");: Ghi phần thứ hai của nội dung phản hồi.

Tóm tắt

* **Giao thức HTTP:** Một giao thức cơ bản cho giao tiếp web, tuân theo mô hình yêu cầu/phản hồi và hoạt động không trạng thái.
* **Máy chủ HTTP**: Phần mềm xử lý các yêu cầu và phản hồi HTTP, chẳng hạn như Kestrel.
* **Luồng yêu cầu / phản hồi**: Từ yêu cầu khách hàng đến phản hồi máy chủ, liên quan đến xử lý phần mềm trung gian trong Kestrel.
* **Sử dụng trình duyệt**: Trình duyệt yêu cầu tài nguyên qua HTTP, sau đó được xử lý và hiển thị.
* **Công cụ dành cho nhà phát triển**: Công cụ dành cho nhà phát triển của Chrome có thể kiểm tra chi tiết lưu lượng truy cập HTTP.
* **Định dạng tin nhắn**: Yêu cầu và phản hồi HTTP bao gồm dòng bắt đầu, tiêu đề và nội dung tùy chọn.
* **Tiêu đề**: Các cặp khóa-giá trị cung cấp thông tin bổ sung về các yêu cầu và phản hồi.
* **Mã trạng thái**: Cho biết kết quả của các yêu cầu HTTP, được phân loại thành mã thông tin, thành công, chuyển hướng, lỗi máy khách và mã lỗi máy chủ.
* **Đặt mã trạng thái và tiêu đề**: ASP.NET Core cho phép tùy chỉnh phản hồi bằng mã, cho phép cài đặt mã trạng thái và tiêu đề như đã minh họa.

Yêu cầu HTTP

Trong thế giới của các ứng dụng web, yêu cầu HTTP là cách của khách hàng để nói, "Này máy chủ, tôi cần một cái gì đó." "Một cái gì đó" này có thể là một trang web, một hình ảnh, dữ liệu từ cơ sở dữ liệu hoặc kết quả của một số tính toán phía máy chủ. Máy khách, thường là trình duyệt web, gửi yêu cầu này đến máy chủ, máy chủ này xử lý nó và trả về phản hồi.

**Cấu trúc của một yêu cầu HTTP**

Yêu cầu HTTP bao gồm một số phần:

1. **Dòng bắt đầu:** Đây là dòng đầu tiên của yêu cầu và nó chứa ba phần thông tin quan trọng:
   * **Phương pháp:** Điều này cho biết hành động mà máy khách muốn máy chủ thực hiện. Các phương pháp phổ biến bao gồm:
     + GET: Truy xuất dữ liệu từ máy chủ.
     + ĐĂNG: Gửi dữ liệu đến máy chủ (ví dụ: dữ liệu biểu mẫu).
     + PUT: Cập nhật tài nguyên hiện có trên máy chủ.
     + XÓA: Xóa tài nguyên khỏi máy chủ.
   * **URI yêu cầu (Mã định danh tài nguyên thống nhất):** Đây là đường dẫn đến tài nguyên trên máy chủ mà máy khách đang yêu cầu.
   * **Phiên bản HTTP:** Điều này chỉ định phiên bản của giao thức HTTP đang được sử dụng (ví dụ: HTTP/1.1 hoặc HTTP/2).
2. **Tiêu đề:** Chúng cung cấp thông tin bổ sung về yêu cầu, chẳng hạn như:
   * Tác nhân người dùng: Trình duyệt hoặc ứng dụng của khách hàng.
   * Chấp nhận: Các loại nội dung mà khách hàng có thể hiểu (ví dụ: HTML, JSON).
   * Máy chủ: Tên miền của máy chủ.
   * Content-Type: Loại dữ liệu được gửi trong nội dung yêu cầu (nếu có).
   * Ủy quyền: Thông tin đăng nhập để xác thực (nếu được yêu cầu).
3. **Dòng trống:** Thao tác này tách các tiêu đề khỏi nội dung của yêu cầu.
4. **Nội dung (Tùy chọn):** Phần này của yêu cầu chứa dữ liệu mà máy khách đang gửi đến máy chủ. Ví dụ: yêu cầu POST có thể bao gồm dữ liệu biểu mẫu hoặc dữ liệu JSON.

**Chuỗi truy vấn: Truyền tham số trong URL**

Chuỗi truy vấn là một cách để chuyển các tham số đến máy chủ trong chính URL. Nó bắt đầu bằng dấu chấm hỏi (?) và đi theo đường dẫn trong URL. Mỗi tham số là một cặp khóa-giá trị, được phân tách bằng dấu bằng (=) và nhiều tham số được phân tách bằng dấu hiệu và (&).

**Ví dụ:**

1. https://example.com/products?category=electronics&brand=apple

Trong ví dụ này, category=electronics và brand=apple là các tham số được chuyển đến máy chủ.

**Đối tượng yêu cầu trong ASP.NET Core**

ASP.NET Core cung cấp một đối tượng HttpRequest cho phép bạn truy cập vào tất cả thông tin trong một yêu cầu đến. Đối tượng này có các thuộc tính như:

* Phương thức: Phương thức HTTP (GET, POST, v.v.).
* Đường dẫn: Đường dẫn URI do máy khách yêu cầu.
* Truy vấn: Một tập hợp các tham số chuỗi truy vấn.
* Tiêu đề: Tập hợp các tiêu đề yêu cầu.
* Nội dung: Một luồng đại diện cho nội dung yêu cầu (nếu có).

**Mã 1: Hiển thị đường dẫn và phương thức yêu cầu**

1. var builder = WebApplication.CreateBuilder(args);
2. var app = trình tạo. Xây dựng();
4. App. Run(async (ngữ cảnh HttpContext) =>
5. {
6. đường dẫn chuỗi = ngữ cảnh. Yêu cầu.Đường dẫn;
7. string method = ngữ cảnh. Yêu cầu.Phương pháp;
9. ngữ cảnh. Response.Headers["Loại nội dung"] = "văn bản/html";
10. chờ ngữ cảnh. Response.WriteAsync($"<p>{path}</p>");
11. chờ ngữ cảnh. Response.WriteAsync($"<p>{method}</p>");
12. });
14. App. Chạy();

Mã này xác định một thành phần phần mềm trung gian đơn giản (sử dụng app. Chạy) rằng:

1. Trích xuất Path và Method từ đối tượng Request.
2. Đặt tiêu đề phản hồi Content-type thành text/html.
3. Viết đường dẫn và phương thức được trích xuất vào nội dung phản hồi dưới dạng đoạn HTML.

**Mã 2: Xử lý yêu cầu GET với tham số truy vấn**

1. App. Run(async (ngữ cảnh HttpContext) =>
2. {
3. ngữ cảnh. Response.Headers["Loại nội dung"] = "văn bản/html";
4. if (ngữ cảnh. Request.Method == "GET")
5. {
6. if (ngữ cảnh. Request.Query.ContainsKey("id"))
7. {
8. ID chuỗi = ngữ cảnh. Request.Query["id"];
9. chờ ngữ cảnh. Response.WriteAsync($"<p>{id}</p>");
10. }
11. }
12. });

Mã này tập trung vào các yêu cầu GET:

1. Nó đặt tiêu đề phản hồi Content-type.
2. Nó kiểm tra xem phương thức yêu cầu có phải là GET hay không.
3. Nếu có, nó sẽ kiểm tra xem tham số truy vấn có tên "id" có tồn tại hay không.
4. Nếu tìm thấy, nó sẽ trích xuất giá trị của tham số "id" và hiển thị nó.

**Mã 3: Giải nén tiêu đề User-Agent**

1. App. Run(async (ngữ cảnh HttpContext) =>
2. {
3. ngữ cảnh. Response.Headers["Loại nội dung"] = "văn bản/html";
4. if (ngữ cảnh. Request.Headers.ContainsKey("Tác nhân người dùng"))
5. {
6. chuỗi userAgent = ngữ cảnh. Request.Headers["Tác nhân người dùng"];
7. chờ ngữ cảnh. Response.WriteAsync($"<p>{userAgent}</p>");
8. }
9. });

Mã này:

1. Đặt tiêu đề phản hồi Content-type.
2. Kiểm tra xem tiêu đề User-Agent có trong yêu cầu hay không.
3. Nếu tìm thấy, nó sẽ trích xuất giá trị của tiêu đề User-Agent và hiển thị nó, cho biết trình duyệt hoặc ứng dụng của máy khách.

**Tóm tắt về HTTP Request:**

Yêu cầu HTTP là các tin nhắn được gửi từ máy khách (như trình duyệt web) đến máy chủ để yêu cầu tài nguyên hoặc hành động. Chúng bao gồm một dòng bắt đầu (phương thức, URI, phiên bản HTTP), tiêu đề (thông tin bổ sung), một dòng trống và một nội dung tùy chọn chứa dữ liệu. Chuỗi truy vấn được sử dụng để truyền các tham số trong URL.

ASP.NET Core cung cấp đối tượng HttpRequest để truy cập chi tiết yêu cầu. Các mã ví dụ đã chứng minh:

1. Hiển thị đường dẫn được yêu cầu và phương thức HTTP.
2. Xử lý các yêu cầu GET và trích xuất các giá trị tham số truy vấn.
3. Truy xuất và hiển thị tiêu đề User-Agent từ một yêu cầu.

Phương thức HTTP

**GET: Truy xuất dữ liệu**

Phương thức GET chủ yếu được thiết kế để tìm nạp dữ liệu từ máy chủ. Hãy nghĩ về nó như yêu cầu máy chủ cung cấp một tài nguyên cụ thể, chẳng hạn như trang web, hình ảnh hoặc một số dữ liệu từ cơ sở dữ liệu. Đây là những gì đặc trưng cho các yêu cầu GET:

1. **Dữ liệu trong URL:** Các thông số được thêm vào URL dưới dạng chuỗi truy vấn. Điều này làm cho các tham số yêu cầu hiển thị trên thanh địa chỉ của trình duyệt.
2. **Kích thước dữ liệu bị giới hạn:** Kích thước dữ liệu có thể được gửi trong yêu cầu GET bị hạn chế do giới hạn về độ dài URL (trình duyệt và máy chủ có thể có giới hạn khác nhau).
3. **Idempotent:** Yêu cầu GET được coi là idempotent. Điều này có nghĩa là bạn có thể thực hiện cùng một yêu cầu GET nhiều lần và nó sẽ có tác dụng tương tự như thực hiện một lần (giả sử dữ liệu cơ bản không thay đổi).
4. **Bộ nhớ đệm:** Các yêu cầu GET có thể được lưu vào bộ nhớ đệm, có nghĩa là nếu một máy khách yêu cầu lại cùng một tài nguyên, trình duyệt có thể phục vụ phản hồi đã truy xuất trước đó từ bộ nhớ đệm của nó, cải thiện hiệu suất.
5. **Bảo mật:** Yêu cầu GET thường kém an toàn hơn các yêu cầu POST vì dữ liệu hiển thị trong URL. Tránh sử dụng GET cho thông tin nhạy cảm như mật khẩu hoặc số thẻ tín dụng.

**Ví dụ về yêu cầu GET:**

1. GET /products?category=electronics&brand=apple HTTP/1.1
2. Người dẫn chương trình: example.com

**ĐĂNG: Gửi dữ liệu**

Phương thức POST chủ yếu được sử dụng để gửi dữ liệu đến máy chủ để xử lý. Dữ liệu này thường được bao gồm trong nội dung của yêu cầu và không hiển thị trong URL. Dưới đây là sự khác biệt của yêu cầu POST với GET:

1. **Dữ liệu trong Body:** Dữ liệu được gửi trong nội dung yêu cầu, phù hợp hơn để gửi một lượng lớn dữ liệu hoặc thông tin nhạy cảm.
2. **Không phải Idempotent:** Yêu cầu POST không phải là idempotent. Các yêu cầu POST lặp đi lặp lại có thể dẫn đến các kết quả khác nhau (ví dụ: tạo nhiều tài nguyên hoặc kích hoạt hành động nhiều lần).
3. **Không thể cachable:** Các yêu cầu POST thường không được lưu vào bộ nhớ đệm, vì chúng thường dẫn đến các thay đổi trên máy chủ.
4. **Bảo mật:** Yêu cầu POST được coi là an toàn hơn các yêu cầu GET vì dữ liệu không được hiển thị trong URL. Tuy nhiên, chúng vẫn dễ bị tấn công như giả mạo yêu cầu chéo trang web (CSRF), đòi hỏi các biện pháp bảo mật bổ sung.

**Ví dụ về yêu cầu POST:**

1. ĐĂNG / đăng nhập HTTP / 1.1
2. Người dẫn chương trình: example.com
3. Loại nội dung: application/x-www-form-urlencoded
5. username=john&password=bí mật

**Chọn giữa GET và POST**

* **Sử dụng GET khi:**
  + Bạn đang truy xuất dữ liệu từ máy chủ.
  + Bạn muốn yêu cầu có thể đánh dấu.
  + Dữ liệu được gửi nhỏ và không nhạy cảm.
* **Sử dụng POST khi:**
  + Bạn đang gửi dữ liệu đến máy chủ để xử lý.
  + Yêu cầu có thể gây ra thay đổi trên máy chủ.
  + Bạn đang gửi dữ liệu nhạy cảm hoặc một lượng lớn dữ liệu.

Người đưa thư

Postman là một công cụ phát triển và thử nghiệm API linh hoạt. Nó cho phép bạn dễ dàng tạo các yêu cầu HTTP, gửi chúng đến ứng dụng ASP.NET Core của bạn (hoặc bất kỳ API nào) và kiểm tra các phản hồi. Đó là một cách tuyệt vời để gỡ lỗi, thử nghiệm và khám phá các điểm cuối API của bạn.

**Cài đặt**

1. **Tải xuống:** Truy cập trang web chính thức của Postman (<https://www.postman.com/downloads/>) và tải xuống phiên bản phù hợp với hệ điều hành của bạn (Windows, macOS, Linux).
2. **Cài đặt:** Làm theo hướng dẫn trên màn hình để cài đặt Postman. Quá trình này thường đơn giản.

**Cách sử dụng: Thực hiện yêu cầu đến ứng dụng ASP.NET Core của bạn**

Giả sử ứng dụng ASP.NET Core của bạn đang chạy cục bộ tại https://localhost:7070 và có điểm cuối /api/products. Dưới đây là cách sử dụng Postman:

1. **Khởi chạy Postman:** Mở ứng dụng Postman.
2. **Tạo một yêu cầu mới:**
   * Nhấp vào nút "Mới" ở góc trên cùng bên trái.
   * Chọn "Yêu cầu" từ các tùy chọn.
3. **Đặt phương thức yêu cầu và URL:**
   * Trong trình tạo yêu cầu, chọn phương thức HTTP thích hợp (GET, POST, PUT, DELETE, v.v.) từ trình đơn thả xuống.
   * Nhập URL đầy đủ của điểm cuối ASP.NET Core của bạn (ví dụ: https://localhost:7070/api/products) vào thanh địa chỉ.
4. **(Tùy chọn) Thêm tiêu đề:**
   * Nếu điểm cuối của bạn yêu cầu các tiêu đề cụ thể (như Content-Type), hãy nhấp vào tab "Tiêu đề" và thêm chúng dưới dạng cặp khóa-giá trị.
5. **(Tùy chọn) Thêm nội dung yêu cầu:**
   * Nếu bạn đang gửi dữ liệu với yêu cầu (ví dụ: dữ liệu JSON cho yêu cầu POST), hãy nhấp vào tab "Nội dung".
   * Chọn định dạng (ví dụ: "thô" cho JSON) và nhập dữ liệu của bạn.
6. **Gửi yêu cầu:**
   * Nhấp vào nút "Gửi".
7. **Kiểm tra phản hồi:**
   * Phản hồi từ ứng dụng ASP.NET Core của bạn sẽ xuất hiện ở phần dưới của Postman. Bạn sẽ thấy:
     + Mã trạng thái (200 OK, 404 Not Found, v.v.)
     + Tiêu đề phản hồi
     + Nội dung phản hồi (nếu có)

Tóm tắt

**HTTP (Giao thức truyền siêu văn bản):**

* **Nền tảng của Web:** HTTP là giao thức cung cấp năng lượng cho World Wide Web. Nó xác định cách máy khách (trình duyệt, ứng dụng) và máy chủ giao tiếp.
* **Chu kỳ yêu cầu-phản hồi:** Giao tiếp tuân theo mô hình yêu cầu-phản hồi. Máy khách gửi yêu cầu và máy chủ gửi lại phản hồi.
* **Stateless:** HTTP là stateless, có nghĩa là mỗi yêu cầu là độc lập. Máy chủ vốn không nhớ các tương tác trong quá khứ.
* **Phương thức:** Phương thức HTTP xác định các hành động (GET, POST, PUT, DELETE, v.v.).
* **Phiên bản:** HTTP/1.1 và HTTP/2 là những phiên bản được sử dụng phổ biến nhất.

**Yêu cầu HTTP:**

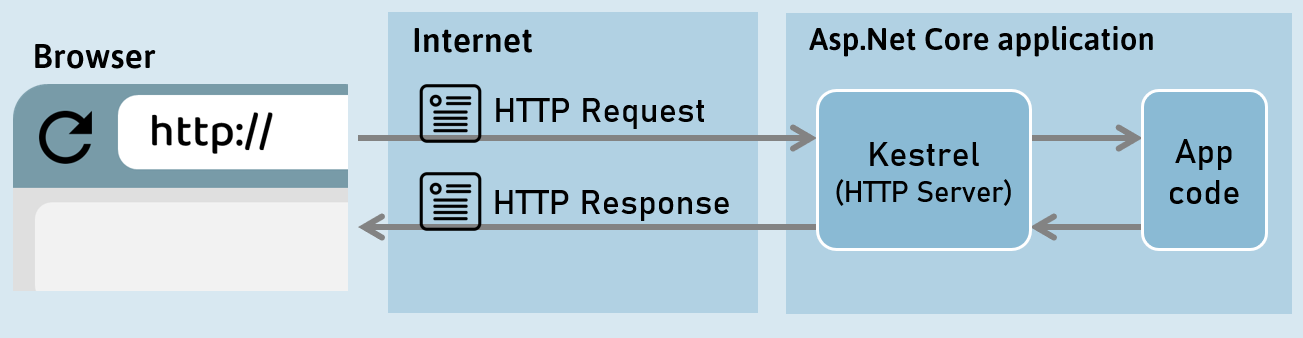
* **Mục đích:** Bắt đầu giao tiếp, yêu cầu tài nguyên hoặc hành động từ máy chủ.
* **Cấu trúc:** Dòng bắt đầu (phương thức, URI, phiên bản), tiêu đề, dòng trống, nội dung tùy chọn.
* **Phương pháp:**
  + GET: Tìm nạp dữ liệu, idempotent, cachable.
  + POST: Gửi dữ liệu, không phải idempotent, thường không được lưu vào bộ nhớ đệm.
  + PUT, DELETE: Cập nhật và xóa tài nguyên tương ứng.
* **Tiêu đề:** Cung cấp siêu dữ liệu như loại nội dung, tác nhân người dùng, xác thực.
* **Body:** Được sử dụng để gửi dữ liệu bằng POST, PUT, v.v.

**Phản hồi HTTP:**

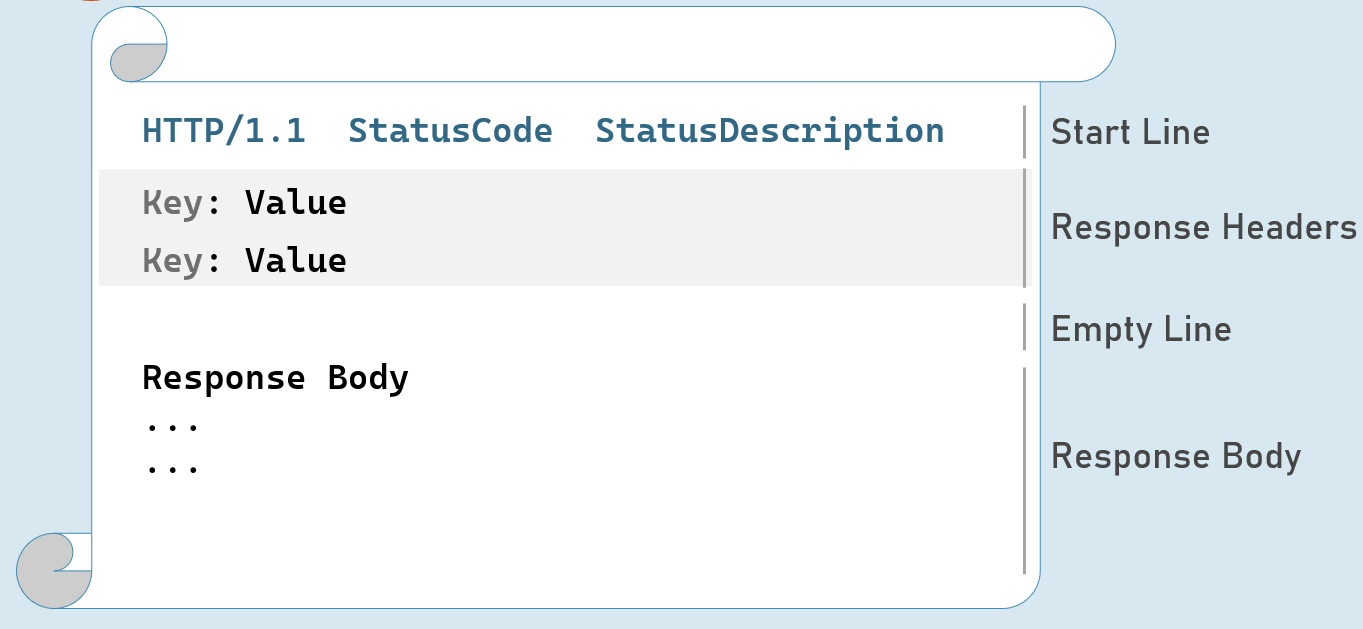
* **Mục đích:** Trả lời của máy chủ cho một yêu cầu.
* **Cấu trúc:** Dòng bắt đầu (phiên bản, mã trạng thái, cụm từ lý do), tiêu đề, dòng trống, nội dung tùy chọn.
* **Mã trạng thái:** Mã ba chữ số cho biết kết quả (200 OK, 404 Không tìm thấy, 500 Lỗi máy chủ nội bộ).
* **Tiêu đề:** Cung cấp siêu dữ liệu về phản hồi (loại nội dung, độ dài, bộ nhớ đệm).
* **Nội dung:** Chứa dữ liệu được yêu cầu (HTML, JSON, v.v.) hoặc thông báo lỗi.

HTTP là một giao thức ứng dụng xác định tập hợp các quy tắc để gửi yêu cầu từ trình duyệt này sang máy chủ khác và gửi phản hồi từ máy chủ này sang trình duyệt khác.

Ban đầu được phát triển bởi Tim Berners Lee, sau đó được tiêu chuẩn hóa bởi IETF (Lực lượng đặc nhiệm kỹ thuật Internet) và W3C (World Wide Web Consortium)



Phản hồi HTTP



Dòng bắt đầu phản hồi

Bao gồm phiên bản HTTP, mã trạng thái và mô tả trạng thái.

**Phiên bản HTTP:** 1/1 | 2 | 3

**Mã trạng thái:** 101 | 200 | 302 | 400 | 401 | 404 | 500

**Mô tả trạng thái:** Chuyển đổi giao thức | Được | Tìm thấy | Yêu cầu xấu | Trái phép | Không tìm thấy | Lỗi máy chủ nội bộ

Mã trạng thái phản hồi HTTP

**1xx | Thông tin**

101 Giao thức chuyển đổi

**2xx | Sự thành công**

200 đồng ý

**3xx | Redirection**

302 Tìm thấy

304 Không sửa đổi

**4xx | Lỗi khách hàng**

400 Yêu cầu xấu

401 Trái phép

404 Không tìm thấy

**5xx | Lỗi máy chủ**

500 Lỗi máy chủ nội bộ

Tiêu đề phản hồi HTTP

**Ngày**

Ngày và giờ phản hồi. ví dụ: Tue, 15 Nov 1994 08:12:31 GMT

**Máy chủ**

Tên của máy chủ.

ví dụ: Server=Kestrel

**Loại nội dung**

Loại MIME của nội dung phản hồi.

Ví dụ: text/plain, text/html, application/json, application/xml, v.v.

**Độ dài nội dung**

Độ dài (byte) của nội dung phản hồi.

Ví dụ: 100

**Kiểm soát bộ nhớ cache**

Cho biết số giây mà phản hồi có thể được lưu vào bộ nhớ đệm tại trình duyệt.

Ví dụ: max-age = 60

**Đặt bánh quy**

Chứa cookie để gửi đến trình duyệt.

Ví dụ: x = 10

**Kiểm soát truy cập-Cho phép-Nguồn gốc**

Được sử dụng để bật CORS (Cross-Origin-Resource-Sharing)

ví dụ: Access-Control-Allow-Origin: http://www.example.com

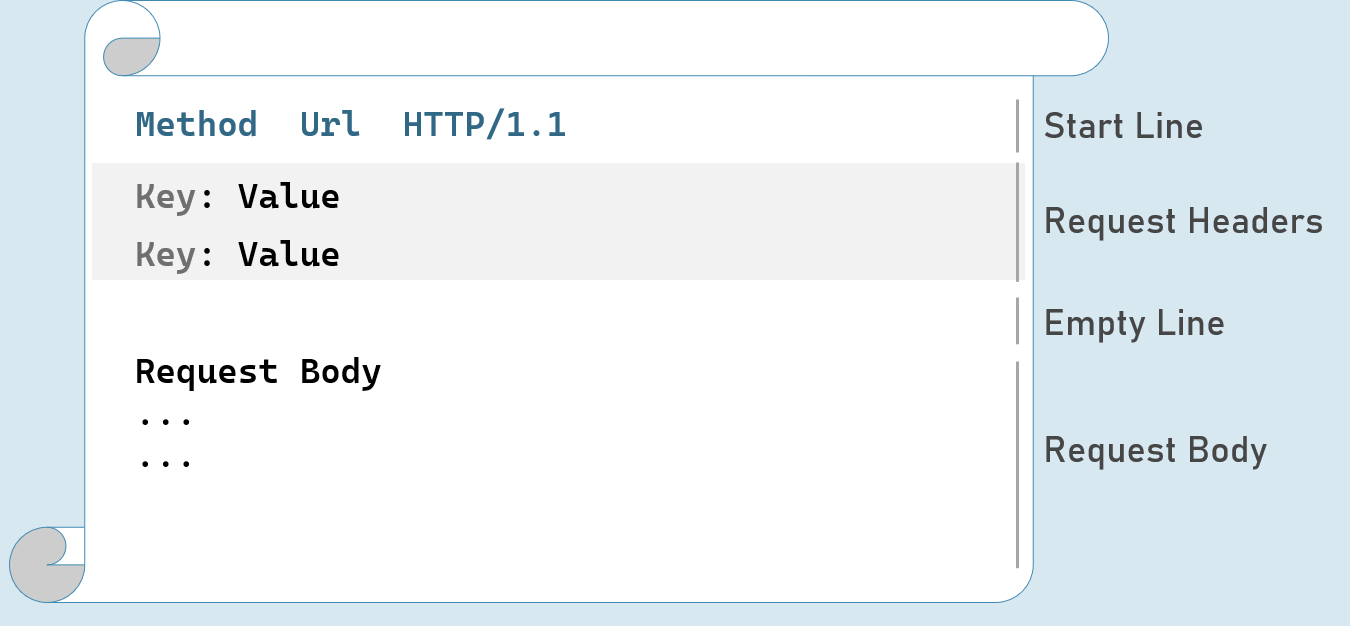
**Vị trí**

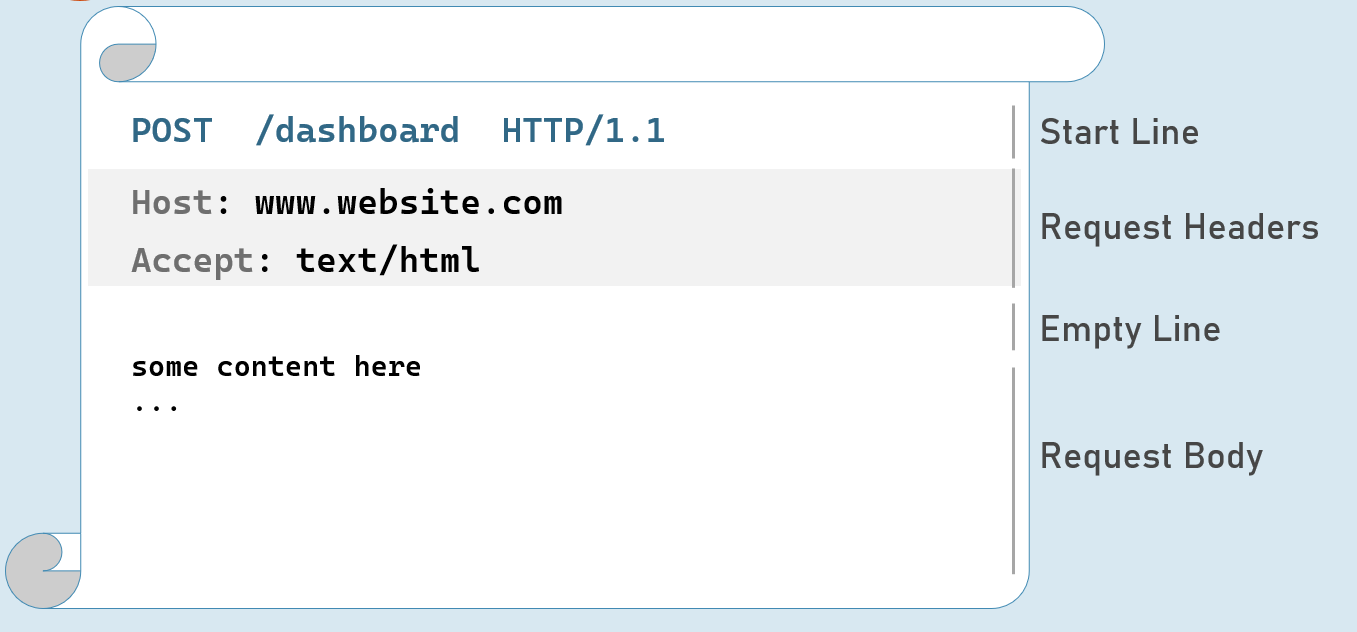
Chứa url để chuyển hướng.

ví dụ: http://www.example-redirect.com

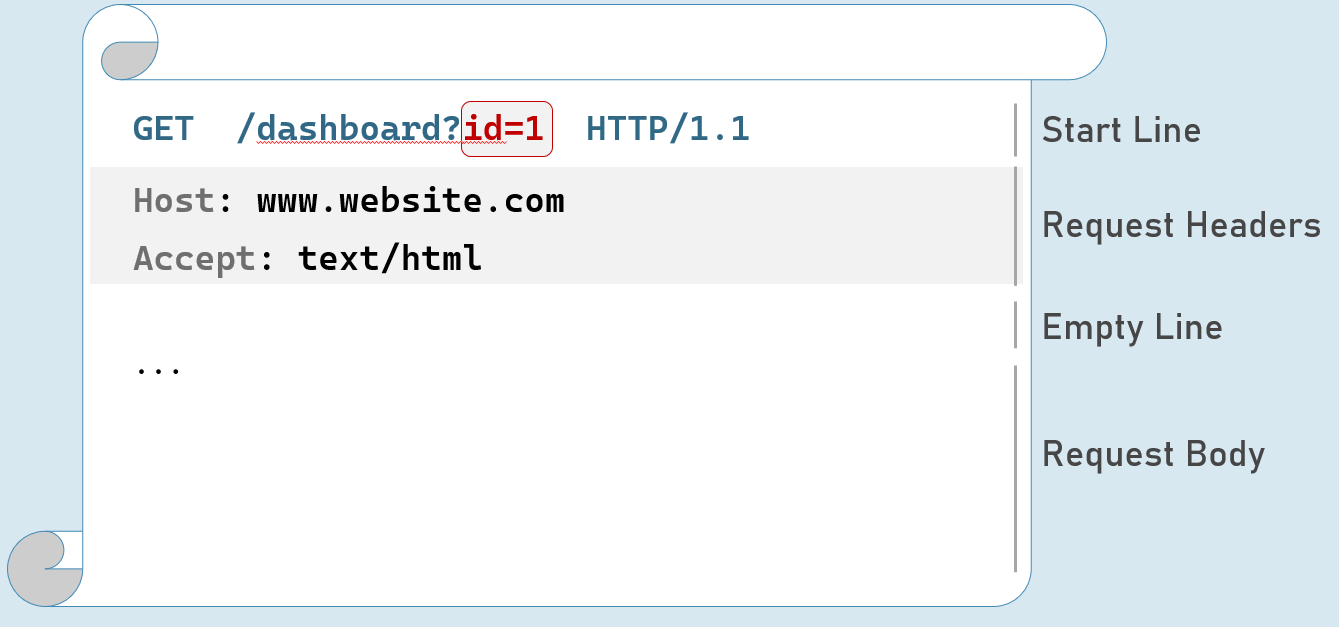
Đọc thêm: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers>

Yêu cầu HTTP





**Yêu cầu HTTP - với chuỗi truy vấn**



Tiêu đề yêu cầu HTTP

**Chấp nhận**

Thể hiện loại nội dung phản hồi MIME được máy khách chấp nhận. Ví dụ: văn bản/html

**Ngôn ngữ chấp nhận**

Đại diện cho ngôn ngữ tự nhiên của nội dung phản hồi được khách hàng chấp nhận. ví dụ: en-US

**Loại nội dung**

Loại MIME của nội dung yêu cầu.

Ví dụ: text/x-www-form-urlencoded, application/json, application/xml, multipart/form-data

**Độ dài nội dung**

Độ dài (byte) của nội dung yêu cầu.

Ví dụ: 100

**Ngày**

Ngày giờ yêu cầu.

Ví dụ: Tue, 15 Nov 1994 08:12:31 GMT

**Chủ nhà**

Tên miền máy chủ.

Ví dụ: www.example.com

**Tác nhân người dùng**

Chi tiết trình duyệt (máy khách).

Ví dụ: Mozilla/5.0 Firefox/12.0

**Bánh quy**

Chứa cookie để gửi đến máy chủ.

Ví dụ: x = 100

Đọc thêm: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers>

Phương thức yêu cầu HTTP

**ĐƯỢC**

Yêu cầu truy xuất thông tin (trang, đối tượng thực thể hoặc tệp tĩnh).

**Trụ**

Gửi một đối tượng thực thể đến máy chủ; Nói chung, nó sẽ được chèn vào cơ sở dữ liệu.

**Để**

Gửi một đối tượng thực thể đến máy chủ; thường cập nhật tất cả các thuộc tính (cập nhật đầy đủ) nó trong cơ sở dữ liệu.

**Vá**

Gửi một đối tượng thực thể đến máy chủ; thường cập nhật một số thuộc tính (cập nhật một phần) nó trong cơ sở dữ liệu.

**Xóa**

Yêu cầu xóa một thực thể trong cơ sở dữ liệu.

HTTP Get [vs] Post

**Được:**

* Được sử dụng để truy xuất dữ liệu từ máy chủ.
* Các tham số sẽ có trong url yêu cầu (chỉ dưới dạng chuỗi truy vấn).
* Chỉ có thể gửi số lượng ký tự giới hạn đến máy chủ. Tối đa: 2048 ký tự
* Được sử dụng chủ yếu như một phương pháp yêu cầu mặc định để truy xuất trang, tệp tĩnh, v.v.
* Có thể được lưu vào bộ nhớ cache bởi trình duyệt / công cụ tìm kiếm.

**Trụ:**

* Được sử dụng để chèn dữ liệu vào máy chủ
* Các tham số sẽ nằm trong nội dung yêu cầu (dưới dạng chuỗi truy vấn, json, xml hoặc form-data).
* Có thể gửi dữ liệu không giới hạn đến máy chủ.
* Chủ yếu được sử dụng để gửi biểu mẫu / gọi XHR
* Không thể được lưu vào bộ nhớ cache bởi các trình duyệt / công cụ tìm kiếm.

# **2. Phần mềm trung gian [MVC vs API]**

Về cốt lõi, phần mềm trung gian trong ASP.NET Core là một loạt các thành phần tạo thành một đường ống mà qua đó mọi yêu cầu và phản hồi HTTP đều trôi qua. Mỗi thành phần phần mềm trung gian có thể:

1. **Kiểm tra** yêu cầu đến.
2. **Sửa đổi** yêu cầu hoặc phản hồi (nếu cần).
3. **Gọi** phần mềm trung gian tiếp theo trong quy trình hoặc rút ngắn quá trình và tự tạo phản hồi.

Quy trình này cho phép bạn mô-đun hóa logic của ứng dụng và thêm các tính năng như xác thực, ghi nhật ký, xử lý lỗi, định tuyến, v.v. theo cách sạch sẽ và có thể bảo trì.

**Chuỗi phần mềm trung gian (Request Pipeline)**

Hãy tưởng tượng quy trình yêu cầu là một loạt các đường ống được kết nối. Mỗi phần mềm trung gian giống như một van trong đường ống này, cho phép bạn kiểm soát luồng thông tin và áp dụng các hoạt động cụ thể ở các giai đoạn khác nhau. Thứ tự bạn đăng ký phần mềm trung gian của mình rất quan trọng, vì chúng được thực hiện tuần tự.

**App. Sử dụng vs. app. Chạy**

Hai phương pháp này là cơ bản để thêm phần mềm trung gian vào quy trình của bạn, nhưng chúng có những điểm khác biệt chính:

* **App. Use(async (context, next) => { ... })**
  + **Phần mềm trung gian không phải đầu cuối:** Loại phần mềm trung gian này thường thực hiện một số hành động và sau đó gọi đại diện tiếp theo để chuyển quyền kiểm soát cho phần mềm trung gian tiếp theo trong quy trình.
  + **Có thể sửa đổi yêu cầu / phản hồi:** Nó có thể thay đổi yêu cầu hoặc phản hồi trước khi chuyển nó theo.
  + **Ví dụ:** Xác thực, ghi nhật ký, tiêu đề tùy chỉnh, v.v.
* **App. Run(async (context) => { ... })**
  + **Phần mềm trung gian đầu cuối:** Phần mềm trung gian này không gọi next; nó kết thúc đường ống và tự tạo phản hồi.
  + **Thường được sử dụng cho phản hồi cuối cùng:** Nó thường được sử dụng để xử lý các yêu cầu không cần xử lý thêm (ví dụ: trả về một tin nhắn đơn giản).
  + **Không thể sửa đổi yêu cầu:** Vì nó là cuối dòng, nó không thể sửa đổi yêu cầu trước khi chuyển nó.

**Mã 1: Hậu quả của nhiều ứng dụng. Chạy cuộc gọi**

1. App. Run(async (HttpContext context) => {
2. chờ ngữ cảnh. Response.WriteAsync("Xin chào");
3. });
5. App. Run(async (HttpContext context) => {
6. chờ ngữ cảnh. Response.WriteAsync("Xin chào một lần nữa");
7. });
9. App. Chạy();

Trong mã này, chỉ ứng dụng đầu tiên. Chạy phần mềm trung gian sẽ được thực thi. Nó chấm dứt quy trình bằng cách viết "Xin chào" cho phản hồi và ứng dụng tiếp theo. Run (sẽ viết "Hello again") không bao giờ có cơ hội chạy.

**Mã 2: Chuỗi phần mềm trung gian với ứng dụng. Sử dụng và ứng dụng. Chạy**

1. phần mềm trung bình 1
2. App. Use(async (ngữ cảnh, tiếp theo) => {
3. chờ ngữ cảnh. Response.WriteAsync("Xin chào ");
4. await next(ngữ cảnh);
5. });
7. phần mềm trung gian 2
8. App. Use(async (ngữ cảnh, tiếp theo) => {
9. chờ ngữ cảnh. Response.WriteAsync("Xin chào một lần nữa ");
10. await next(ngữ cảnh);
11. });
13. phần mềm trung gian 3
14. App. Run(async (HttpContext context) => {
15. chờ ngữ cảnh. Response.WriteAsync("Xin chào một lần nữa");
16. });

Mã này trình bày một cách chính xác để chuỗi phần mềm trung gian.

1. Ứng dụng đầu tiên. Sử dụng viết "Xin chào" cho phản hồi và sau đó gọi tiếp theo để chuyển quyền kiểm soát cho phần mềm trung gian tiếp theo.
2. Ứng dụng thứ hai. Sử dụng viết "Xin chào một lần nữa" và cũng gọi tiếp theo.
3. Ứng dụng cuối cùng. Run (là terminal) viết "Hello again" và kết thúc pipeline. Kết quả sẽ là đầu ra của "Hello Hello again Hello again".

**Những điểm chính cần nhớ**

* **Thứ tự phần mềm trung gian rất quan trọng:** Thứ tự mà bạn đăng ký phần mềm trung gian rất quan trọng, vì chúng được thực hiện theo trình tự.
* **Sử dụng ứng dụng. Sử dụng cho các hành động không phải là thiết bị đầu cuối:** Sử dụng nó cho các tác vụ như xác thực, ghi nhật ký hoặc sửa đổi tiêu đề/nội dung.
* **Sử dụng ứng dụng. Chạy để chấm dứt đường ống:** Sử dụng nó khi bạn muốn tạo phản hồi cuối cùng.
* **Short-Circuiting:** Phần mềm trung gian có thể chọn rút ngắn đường ống (không gọi tiếp theo) và trả về phản hồi sớm nếu cần.

Phần mềm trung gian tùy chỉnh trong ASP.NET Cor

Mặc dù ASP.NET Core cung cấp rất nhiều thành phần phần mềm trung gian tích hợp, nhưng đôi khi bạn cần tạo thành phần của riêng mình để giải quyết các yêu cầu cụ thể dành riêng cho ứng dụng của mình. Phần mềm trung gian tùy chỉnh cho phép bạn:

* **Gói logic:** Gói các hoạt động liên quan (ví dụ: ghi nhật ký, kiểm tra bảo mật, tiêu đề tùy chỉnh) thành một thành phần có thể tái sử dụng.
* **Tùy chỉnh hành vi:** Điều chỉnh quy trình yêu cầu/phản hồi để phù hợp chính xác với nhu cầu ứng dụng của bạn.
* **Cải thiện tổ chức mã:** Giữ cho mã phần mềm trung gian của bạn sạch sẽ và có thể bảo trì.

**Giải phẫu của một lớp phần mềm trung gian tùy chỉnh**

1. **Triển khai IMiddleware:** Giao diện này yêu cầu một phương thức duy nhất: InvokeAsync(HttpContext context, RequestDelegate next). Đây là trọng tâm của logic phần mềm trung gian của bạn.
2. **Phương thức InvokeAsync hoặc Invoke:**
   * context: HttpContext cung cấp quyền truy cập vào các đối tượng yêu cầu và phản hồi.
   * next: RequestDelegate cho phép bạn gọi phần mềm trung gian tiếp theo trong quy trình.

**Giải thích mã**

Hãy mổ xẻ mã bạn đã cung cấp:

1. MyCustomMiddleware.cs
2. không gian tên MiddlewareExample.CustomMiddleware
3. {
4. lớp công cộng MyCustomMiddleware : IMiddleware
5. {
6. công khai async Task InvokeAsync(HttpContext context, RequestDelegate next)
7. {
8. chờ ngữ cảnh. Response.WriteAsync("Phần mềm trung gian tùy chỉnh của tôi - Bắt đầu\n");
9. await next(ngữ cảnh);
10. chờ ngữ cảnh. Response.WriteAsync("Phần mềm trung gian tùy chỉnh của tôi - kết thúc\n");
11. }
12. }
14. Phương thức mở rộng để đăng ký dễ dàng
15. lớp tĩnh công khai CustomMiddlewareExtension
16. {
17. IApplicationBuilder tĩnh công khai UseMyCustomMiddleware (ứng dụng IApplicationBuilder này)
18. {
19. trả lại ứng dụng. UseMiddleware<MyCustomMiddleware>();
20. }
21. }
22. }

* **Lớp MyCustomMiddleware:** Lớp này triển khai IMiddleware. Phương thức InvokeAsync của nó:
  + Ghi "Phần mềm trung gian tùy chỉnh của tôi - Bắt đầu" vào phản hồi.
  + Gọi next(context) để gọi phần mềm trung gian tiếp theo trong quy trình.
  + Viết "Phần mềm trung gian tùy chỉnh của tôi - Kết thúc" vào phản hồi sau khi phần mềm trung gian tiếp theo kết thúc.
* **CustomMiddlewareExtension Class:** Điều này cung cấp một phương thức mở rộng thuận tiện UseMyCustomMiddleware để đăng ký phần mềm trung gian của bạn trong phương thức Startup.Configure.

1. Program.cs (hoặc Startup.cs)
2. sử dụng MiddlewareExample.CustomMiddleware;
4. // ...
6. chủ thầu. Services.AddTransient<MyCustomMiddleware>(); Đăng ký tạm thời
8. App. Use(async (ngữ cảnh HttpContext, RequestDelegate tiếp theo) => {
9. chờ ngữ cảnh. Response.WriteAsync("Từ Midleware 1\n");
10. await next(ngữ cảnh);
11. });
13. App. UseMyCustomMiddleware(); Sử dụng phương thức mở rộng
15. App. Run(async (HttpContext context) => {
16. chờ ngữ cảnh. Response.WriteAsync("Từ phần mềm trung gian 3\n");
17. });

**Cách thức hoạt động**

1. **Đăng ký:** Bạn đăng ký MyCustomMiddleware như một dịch vụ tạm thời để ASP.NET Core có thể tạo các phiên bản của nó khi cần thiết.
2. **Tích hợp đường ống:** Ứng dụng. UseMyCustomMiddleware() bổ sung liền mạch phần mềm trung gian của bạn vào quy trình.
3. **Thứ tự thực thi:** Các thành phần phần mềm trung gian được thực thi theo thứ tự chúng được thêm vào quy trình. Trong trường hợp này, thứ tự sẽ là Middleware 1, MyCustomMiddleware, sau đó là Middleware 3.

**Ra**

Khi chạy ứng dụng, bạn sẽ thấy kết quả sau trong trình duyệt của mình:

1. Từ Midleware 1
2. Phần mềm trung gian tùy chỉnh của tôi - Bắt đầu
3. Từ phần mềm trung gian 3
4. Phần mềm trung gian tùy chỉnh của tôi - Kết thúc

Điều này thể hiện rõ ràng luồng thực thi thông qua chuỗi phần mềm trung gian.

Phần mềm trung gian thông thường tùy chỉnh

Phần mềm trung gian ASP.NET Core có hai hương vị: thông thường và dựa trên nhà máy. Phần mềm trung gian thông thường, như được hiển thị trong ví dụ của bạn, là một cách đơn giản nhưng mạnh mẽ để đóng gói logic tùy chỉnh để xử lý các yêu cầu và phản hồi HTTP.

**Các đặc điểm chính**

* **Dựa trên lớp:** Phần mềm trung gian thông thường được triển khai như một lớp.
* **Constructor Injection:** Nó nhận các phụ thuộc (nếu có) thông qua constructor của nó.
* **Phương thức Invoke:** Đây là trái tim của phần mềm trung gian, chứa logic xử lý từng yêu cầu.
* **RequestDelegate:** Phương thức Invoke nhận tham số RequestDelegate (\_next trong ví dụ của bạn). Đại biểu này đại diện cho phần mềm trung gian tiếp theo trong quy trình.
* **Tính linh hoạt:** Bạn có toàn quyền kiểm soát các đối tượng yêu cầu và phản hồi trong phương thức Invoke.

**Phân tích mã: HelloCustomMiddleware**

1. HelloCustomMiddleware.cs
2. lớp công cộng HelloCustomMiddleware
3. {
4. \_next RequestDelegate chỉ đọc riêng;
6. công khai HelloCustomMiddleware(RequestDelegate tiếp theo)
7. {
8. \_next = tiếp theo;
9. }
11. công khai async Task Invoke(HttpContext httpContext)
12. {
13. if (httpContext.Request.Query.ContainsKey("firstname") &&
14. httpContext.Request.Query.ContainsKey("lastname"))
15. {
16. string fullName = httpContext.Request.Query["firstname"] + " " + httpContext.Request.Query["lastname"];
17. chờ đợi httpContext.Response.WriteAsync(fullName);
18. }
19. await \_next(httpContext);
20. }
21. }
23. Phương thức mở rộng để đăng ký dễ dàng
24. lớp tĩnh công cộng HelloCustomModdleExtensions
25. {
26. công khai tĩnh IApplicationBuilder UseHelloCustomMiddleware (trình tạo IApplicationBuilder này)
27. {
28. người xây dựng trở lại. UseMiddleware<HelloCustomMiddleware>();
29. }
30. }

Hãy phân tích từng phần:

1. **Constructor:** Constructor nhận RequestDelegate, được lưu trữ để sử dụng sau này để gọi phần mềm trung gian tiếp theo trong pipeline.
2. **Phương pháp gọi:**
   * Nó kiểm tra xem chuỗi truy vấn có chứa cả tham số "tên" và "họ" hay không.
   * Nếu vậy, nó kết hợp các giá trị thành một chuỗi fullName và ghi nó vào phản hồi.
   * **Điều quan trọng:** Nó gọi await \_next(httpContext); để tiếp tục chuỗi phần mềm trung gian. Dòng này đảm bảo rằng yêu cầu được chuyển đến các thành phần phần mềm trung gian tiếp theo, ngay cả khi tên đầy đủ được tạo ra.
   * Theo thiết kế, bất kỳ mã nào sau dòng này, chẳng hạn như nhận xét "// sau logic", sẽ không thực thi cho các yêu cầu chứa cả "firstname" và "lastname", như await \_next(httpContext); line ngay lập tức chuyển quyền kiểm soát sang phần mềm trung gian tiếp theo trong quy trình.
3. **UseHelloCustomMiddleware Extension:** Phương thức mở rộng này đơn giản hóa quá trình đăng ký bằng cách ẩn các chi tiết về việc khởi tạo và sử dụng lớp phần mềm trung gian tùy chỉnh của bạn.

**Program.cs (hoặc Startup.cs): Sử dụng phần mềm trung gian**

1. // ... phần mềm trung gian khác ...
2. App. UseMyCustomMiddleware();
3. App. UseHelloCustomMiddleware();
4. // ...

**Cách thức hoạt động**

1. Khi có yêu cầu, ASP.NET Core sẽ đi qua quy trình phần mềm trung gian.
2. Nó đến HelloCustomMiddleware, kiểm tra các tham số truy vấn cụ thể.
3. Nếu có các tham số, phần mềm trung gian sẽ tạo ra một lời chào được cá nhân hóa.
4. Bất kể nó có tạo ra lời chào hay không, phần mềm trung gian gọi next(context) để chuyển yêu cầu đến thành phần phần mềm trung gian tiếp theo trong quy trình.

**Những điểm chính**

* **Đơn giản:** Phần mềm trung gian thông thường rất dễ viết và dễ hiểu.
* **Kiểm soát:** Bạn có quyền kiểm soát chi tiết về cách xử lý yêu cầu và cách tạo phản hồi.
* **Phương pháp mở rộng:** Sử dụng các phương thức mở rộng để làm cho việc đăng ký phần mềm trung gian sạch sẽ và dễ đọc.

Thứ tự lý tưởng của đường ống phần mềm trung gian

1. **Xử lý ngoại lệ / lỗi:**
   * **Mục đích:** Bắt và xử lý các ngoại lệ xảy ra ở bất kỳ đâu trong quy trình.
   * **Ví dụ:** UseExceptionHandler, UseDeveloperExceptionPage (dành cho môi trường phát triển).
2. **Chuyển hướng HTTPS:**
   * **Mục đích:** Chuyển hướng các yêu cầu HTTP đến HTTPS để bảo mật.
   * **Ví dụ:** UseHttpsRedirection.
3. **Tệp tĩnh:**
   * **Mục đích:** Phục vụ các tệp tĩnh như hình ảnh, CSS và JavaScript trực tiếp đến máy khách.
   * **Ví dụ:** UseStaticFiles.
4. **Routing:**
   * **Mục đích:** Khớp các yêu cầu đến với các điểm cuối cụ thể dựa trên URL của chúng.
   * **Ví dụ:** UseRouting, UseEndpoints.
5. **CORS (Chia sẻ tài nguyên nguồn gốc chéo):**
   * **Mục đích:** Cho phép các yêu cầu liên nguồn an toàn từ các miền khác nhau.
   * **Ví dụ:** UseCors.
6. **Xác thực:**
   * **Mục đích:** Xác minh danh tính người dùng và thiết lập nguyên tắc người dùng.
   * **Ví dụ:** UseAuthentication.
7. **Phép:**
   * **Mục đích:** Xác định xem người dùng có được phép truy cập vào một tài nguyên cụ thể hoặc thực hiện một hành động nhất định hay không.
   * **Ví dụ:** UseAuthorization.
8. **Phần mềm trung gian tùy chỉnh:**
   * **Mục đích: Các**  thành phần phần mềm trung gian dành riêng cho ứng dụng của bạn để xử lý các tác vụ như ghi nhật ký, cờ tính năng, v.v.

**Lý do đằng sau lệnh**

* **Xử lý ngoại lệ sớm:** Việc phát hiện sớm các ngoại lệ sẽ ngăn chúng lan truyền và gây ra các vấn đề tiếp theo.
* **Bảo mật là trên hết:** Chuyển hướng, xác thực và ủy quyền HTTPS là điều cần thiết để bảo mật ứng dụng của bạn.
* **Tối ưu hóa hiệu suất:** Các tệp tĩnh, bộ nhớ đệm phản hồi và nén được đặt sớm để tối ưu hóa quá trình tạo phản hồi.
* **Định tuyến như một nền tảng:** Định tuyến xác định cách các yêu cầu được xử lý bởi logic cốt lõi của ứng dụng của bạn.
* **CORS cho tính linh hoạt:** CORS cho phép ứng dụng của bạn được sử dụng bởi nhiều loại khách hàng hơn.
* **Phần mềm trung gian tùy chỉnh: Phần**  mềm trung gian tùy chỉnh của bạn có thể được đặt một cách chiến lược trong quy trình để áp dụng logic ở giai đoạn thích hợp.

**Tính linh hoạt và ngoại lệ**

Mặc dù đây là thứ tự được khuyến nghị chung, nhưng có thể có các trường hợp ngoại lệ dựa trên nhu cầu cụ thể của ứng dụng của bạn. Chẳng hạn:

* **Kiểm tra tình trạng:** Bạn có thể muốn đặt phần mềm trung gian kiểm tra tình trạng rất sớm trong quy trình để nhanh chóng xác định trạng thái của ứng dụng mà không cần thực thi các thành phần phần mềm trung gian khác.
* **Phần mềm trung gian chuyên dụng:** Một số thành phần phần mềm trung gian có thể có các yêu cầu đặt hàng cụ thể được ghi lại bởi các nhà cung cấp của họ.

**Ví dụ (Program.cs hoặc Startup.cs):**

1. var builder = WebApplication.CreateBuilder(args);
2. var app = trình tạo. Xây dựng();
4. nếu (ứng dụng. Môi trường.IsDevelopment())
5. {
6. App. UseDeveloperExceptionPage();
7. }
9. App. Sử dụng HttpsRedirection();
10. App. UseStaticFiles ();
11. App. UseRouting();
12. App. UseAuthentication();
13. App. UseAuthorization();
15. // ... phần mềm trung gian tùy chỉnh của bạn ...
17. App. UseEndpoints(điểm cuối =>
18. {
19. điểm cuối. Bộ điều khiển bản đồ(); Hoặc MapRazorPages(), MapGet(), v.v.
20. });

Bằng cách tuân thủ thứ tự được đề xuất này, bạn sẽ tạo ra một ứng dụng ASP.NET Core có cấu trúc tốt và hiệu quả, dễ bảo trì, gỡ lỗi và bảo mật hơn.

Sử dụng khi nào ()

UseWhen() là một phương thức mở rộng mạnh mẽ trong giao diện IApplicationBuilder của ASP.NET Core. Nó cho phép bạn thêm phần mềm trung gian có điều kiện vào quy trình yêu cầu của mình dựa trên một vị ngữ (một điều kiện). Điều này có nghĩa là bạn có thể tạo các đường ống động trong đó các thành phần phần mềm trung gian cụ thể chỉ được thực thi khi đáp ứng các điều kiện nhất định.

**Cú pháp**

1. App. Sử dụng khi nào (
2. context => /\* Điều kiện của bạn ở đây \*/,
3. app => /\* Cấu hình phần mềm trung gian cho nhánh \*/
4. );

* **context:** Đối tượng HttpContext đại diện cho yêu cầu hiện tại.
* **Vị ngữ (Điều kiện):** Một hàm nhận HttpContext và trả về true nếu nhánh phần mềm trung gian nên được thực thi, false nếu không.
* **Cấu hình phần mềm trung gian:** Một hành động định cấu hình các thành phần phần mềm trung gian sẽ được thực thi nếu điều kiện là đúng. Đây là nơi bạn sử dụng ứng dụng. Use(), ứng dụng. Run() hoặc các phương thức đăng ký phần mềm trung gian khác.

**Cách hoạt động của UseWhen()**

1. **Đánh giá vị ngữ:** Khi một yêu cầu đến, phương thức UseWhen() trước tiên đánh giá hàm vị ngữ so với HttpContext.
2. **Phân nhánh (nếu đúng):** Nếu vị ngữ trả về true, nhánh phần mềm trung gian được chỉ định trong hành động cấu hình sẽ được thực thi. Yêu cầu chảy qua nhánh này, có khả năng trải qua các sửa đổi hoặc tạo phản hồi.
3. **Tham gia lại đường ống chính:** Sau khi nhánh được thực thi (hoặc bỏ qua nếu vị ngữ là sai), luồng yêu cầu sẽ tham gia lại quy trình chính, tiếp tục với các thành phần phần mềm trung gian tiếp theo được đăng ký sau lệnh gọi UseWhen().

**Ví dụ mã: Giải thích**

1. App. Sử dụng khi nào (
2. context => ngữ cảnh. Request.Query.ContainsKey("tên người dùng"),
3. ứng dụng => {
4. App. Use(async (ngữ cảnh, tiếp theo) =>
5. {
6. chờ ngữ cảnh. Response.WriteAsync("Xin chào từ nhánh phần mềm trung gian");
7. chờ đợi tiếp theo();
8. });
9. });
11. App. Run(ngữ cảnh không đồng bộ =>
12. {
13. chờ ngữ cảnh. Response.WriteAsync("Xin chào từ phần mềm trung gian tại chuỗi chính");
14. });

* **Điều kiện:** Ngữ cảnh vị ngữ. Request.Query.ContainsKey("username") kiểm tra xem chuỗi truy vấn có chứa tham số có tên "username" hay không.
* **Phần mềm trung gian nhánh:** Nếu có tham số "tên người dùng", phần mềm trung gian nhánh sẽ được thực thi. Nó viết "Hello from Middleware branch" vào phản hồi và sau đó gọi next để cho phép phần còn lại của pipeline tiếp tục.
* **Đường ống chính:** Ứng dụng cuối cùng. Chạy phần mềm trung gian là một phần của quy trình chính. Nó viết "Hello from middleware at main chain" vào phản hồi.

**Ra**

* Nếu yêu cầu chứa tham số truy vấn "username" (ví dụ: /path?username=John), kết quả sẽ là:
  1. Xin chào từ nhánh Middleware
  2. Xin chào từ phần mềm trung gian tại chuỗi chính
* Nếu yêu cầu không chứa tham số "tên người dùng" (ví dụ: /path), kết quả sẽ là:
  1. Xin chào từ phần mềm trung gian tại chuỗi chính

**Khi nào nên sử dụng UseWhen()**

* **Tính năng có điều kiện:** Bật hoặc tắt một số tính năng nhất định dựa trên yêu cầu (ví dụ: chỉ ghi nhật ký cho một số người dùng nhất định, áp dụng quy tắc bộ nhớ đệm dựa trên các tham số truy vấn).
* **Đường ống động:** Tạo các đường ống thích ứng với các yêu cầu khác nhau (ví dụ: phần mềm trung gian xác thực khác nhau cho các tuyến đường cụ thể).
* **Thử nghiệm A/B:** Định tuyến một tập hợp con người dùng thông qua các nhánh phần mềm trung gian thay thế để thử nghiệm.
* **Gỡ lỗi và chẩn đoán:** Chỉ áp dụng phần mềm trung gian chẩn đoán trong môi trường phát triển.

Những điểm chính cần nhớ:

**Hiểu biết khái niệm:**

1. **Pipeline:** Phần mềm trung gian tạo thành một đường ống cho các yêu cầu và phản hồi HTTP. Mỗi thành phần có thể kiểm tra, sửa đổi hoặc chấm dứt luồng.
2. **Vấn đề đặt hàng:** Phần mềm trung gian được thực hiện theo thứ tự nó được đăng ký. Hãy suy nghĩ cẩn thận về trình tự.
3. **Các loại phần mềm trung gian:**
   * **Tích hợp:** ASP.NET Core cung cấp phần mềm trung gian để xác thực, định tuyến, tệp tĩnh, v.v.
   * **Tùy chỉnh:** Bạn có thể tạo logic của riêng mình để thêm logic cụ thể vào ứng dụng của mình.

**App. Sử dụng vs. app. Chạy:**

1. **App. Sử dụng:** Đối với phần mềm trung gian không phải thiết bị đầu cuối. Nó gọi next để chuyển quyền kiểm soát cho thành phần tiếp theo.
2. **App. Run:** Đối với phần mềm trung gian đầu cuối. Nó kết thúc đường ống và tạo phản hồi.

**Phần mềm trung gian tùy chỉnh:**

1. **Hai cách:**
   * **Thông thường:** Dựa trên lớp, sử dụng phương thức Invoke và chèn hàm khởi tạo.
   * **Factory-Based:** Sử dụng một delegate để tạo phiên bản phần mềm trung gian.
2. **Lợi ích:** Đóng gói logic, cải thiện tổ chức mã và cho phép bạn điều chỉnh hành vi của ứng dụng.

**Thứ tự đề xuất:** (Không nghiêm ngặt, nhưng là một hướng dẫn tốt)

1. Xử lý ngoại lệ
2. Chuyển hướng HTTPS
3. Tập tin tĩnh
4. Routing
5. CORS
6. Xác thực
7. Phép
8. Phần mềm trung gian tùy chỉnh
9. MVC/Razor Pages/API tối thiểu

**Điểm thưởng:**

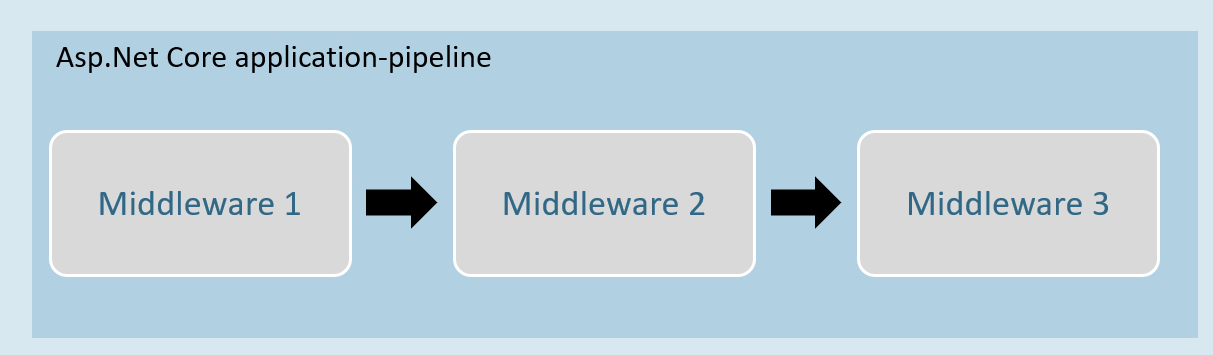
* **Ngắn mạch:** Phần mềm trung gian có thể chọn không gọi tiếp theo và trả về phản hồi sớm.
* **UseWhen:** Thêm các nhánh phần mềm trung gian có điều kiện dựa trên tiêu chí yêu cầu.
* **Tính linh hoạt về thứ tự phần mềm trung gian:** Hiểu lý do đằng sau thứ tự được đề xuất, nhưng cũng biết khi nào nên đi chệch khỏi nó dựa trên các yêu cầu cụ thể của ứng dụng của bạn.

Giới thiệu về phần mềm trung gian

Phần mềm trung gian là một thành phần được lắp ráp vào pipeline ứng dụng để xử lý các yêu cầu và phản hồi.

Phần mềm trung gian được xâu chuỗi lần lượt và thực thi theo cùng một trình tự như cách chúng được thêm vào.





Phần mềm trung gian có thể là một đại biểu yêu cầu (phương thức ẩn danh hoặc biểu thức lambda) [hoặc] một lớp.

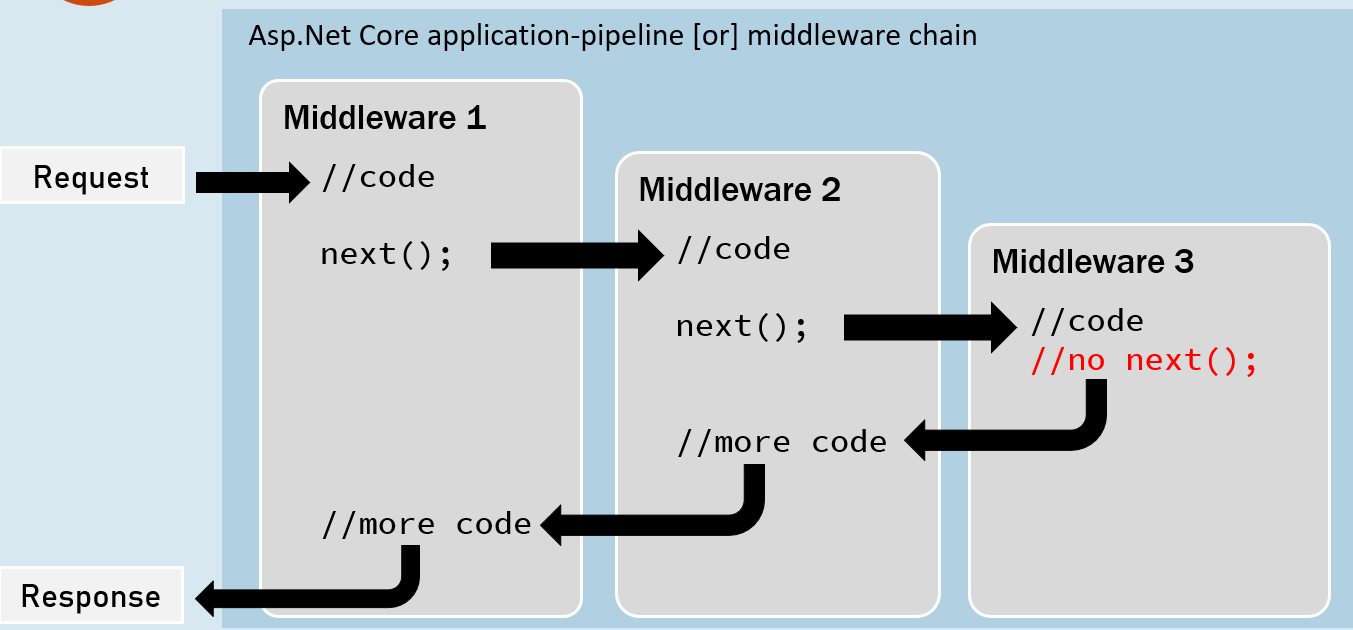
Phần mềm trung gian - Chạy

**App. Chạy( )**

1. App. Run(async (ngữ cảnh HttpContext) =>
2. {
3. mã
4. });

Phương thức mở rộng được gọi là "Run" được sử dụng để thực hiện phần mềm trung gian kết thúc / ngắn mạch không chuyển tiếp yêu cầu đến phần mềm trung gian tiếp theo.

Chuỗi phần mềm trung gian



**App. Sử dụng ( )**

1. App. Use(async (ngữ cảnh HttpContext, RequestDelegate tiếp theo) =>
2. {
3. Trước logic
4. await next(ngữ cảnh);
5. Sau logic
6. });

Phương thức mở rộng được gọi là "Use" được sử dụng để thực hiện một phần mềm trung gian không kết thúc / ngắn mạch có thể / có thể không chuyển tiếp yêu cầu đến phần mềm trung gian tiếp theo.

Lớp phần mềm trung gian

Lớp phần mềm trung gian được sử dụng để tách logic phần mềm trung gian từ biểu thức lambda thành một lớp riêng biệt/có thể tái sử dụng.

1. lớp MiddlewareClassName : IMiddleware
2. {
3. công khai async Task InvokeAsync(HttpContext context, RequestDelegate next)
4. {
5. Trước logic
6. await next(ngữ cảnh);
7. Sau logic
8. }
9. }

App. UseMiddleware<MiddlewareClassName>();

Phần mở rộng phần mềm trung gian

1. lớp MiddlewareClassName : IMiddleware
2. {
3. công khai không đồng bộ Task InvokeAsync(HttpContext context, RequestDelegate next)
4. {
5. Trước logic
6. await next(ngữ cảnh);
7. Sau logic
8. }
9. });

Phương thức mở rộng phần mềm trung gian được sử dụng để gọi phần mềm trung gian bằng một cuộc gọi phương thức duy nhất.

1. lớp tĩnh ClassName
2. {
3. IApplicationBuilder ExtensionMethodName tĩnh công cộng (ứng dụng IApplicationBuilder này)
4. {
5. trả lại ứng dụng. UseMiddleware<MiddlewareClassName>();
6. }
7. }

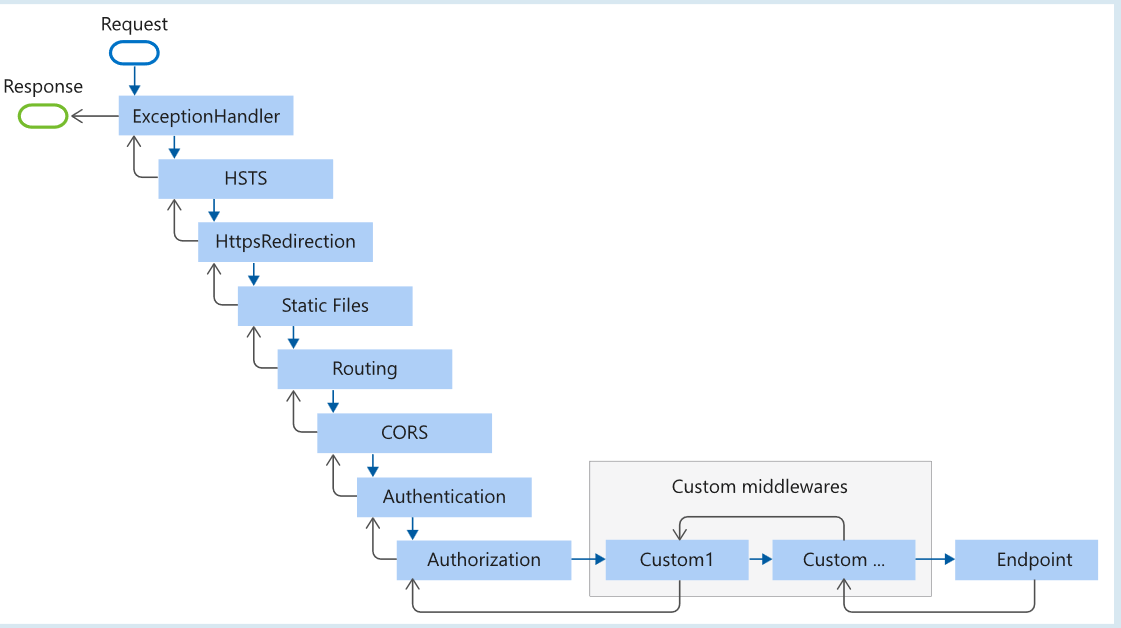
App. ExtensionMethodName();

Phần mềm trung gian thông thường

1. lớp MiddlewareClassName
2. {
3. \_next RequestDelegate chỉ đọc riêng;
5. public MiddlewareClassName(RequestDelegate tiếp theo)
6. {
7. \_next = tiếp theo;
8. }
10. công khai async Task InvokeAsync(HttpContext context)
11. {
12. Trước logic
13. chờ đợi \_next(bối cảnh);
14. Sau logic
15. }
16. });
17. lớp tĩnh ClassName
18. {
19. IApplicationBuilder ExtensionMethodName tĩnh công cộng (ứng dụng IApplicationBuilder này)
20. {
21. trả lại ứng dụng. UseMiddleware<MiddlewareClassName>();
22. }
23. }

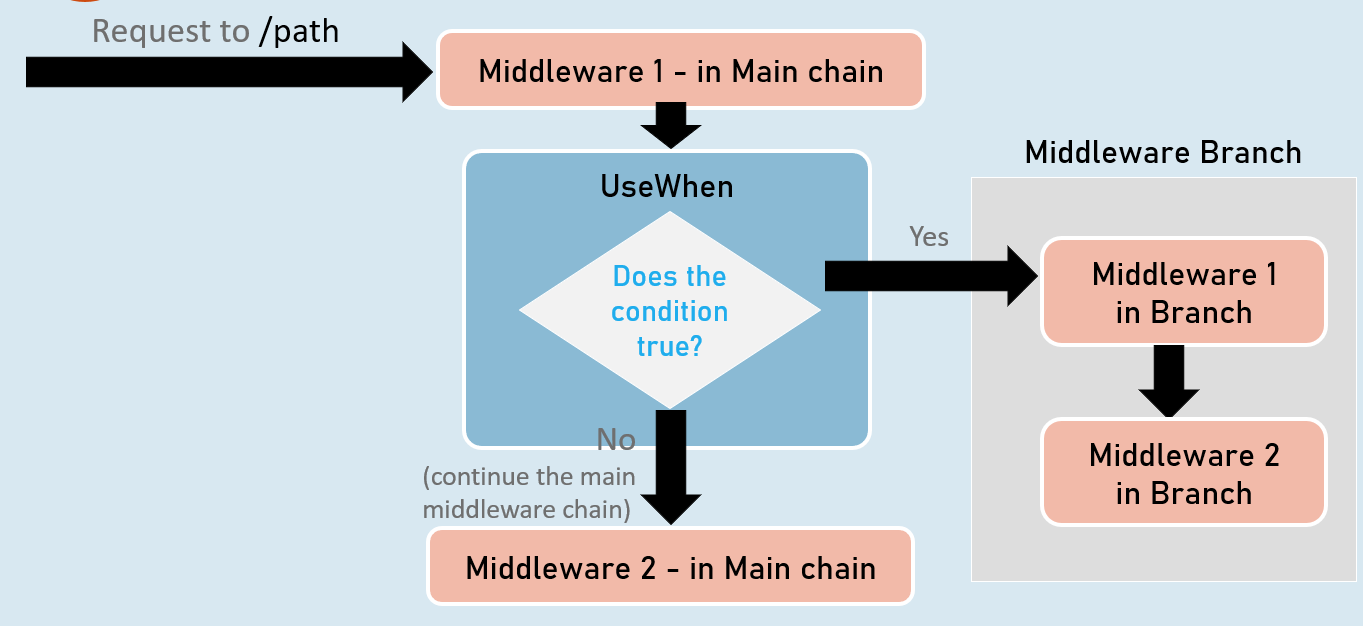
App. ExtensionMethodName();

Thứ tự phù hợp của phần mềm trung gian



1. App. UseExceptionHandler("/lỗi");
2. App. Sử dụngHsts();
3. App. Sử dụng HttpsRedirection();
4. App. UseStaticFiles ();
5. App. UseRouting();
6. App. Sử dụng Cors();
7. App. UseAuthentication();
8. App. UseAuthorization();
9. App. UseSession();
10. App. Bộ điều khiển bản đồ();
11. Thêm phần mềm trung gian tùy chỉnh của bạn
12. App. Chạy();

Phần mềm trung gian - UseWhen



**App. Sử dụng khi nào ( )**

1. App. Sử dụng khi nào (
2. context => { trả lại boolean; },
3. ứng dụng =>
4. {
5. Thêm phần mềm trung gian của bạn
6. }
7. );

Phương thức mở rộng được gọi là "UseWhen" được sử dụng để thực hiện một nhánh của phần mềm trung gian chỉ khi điều kiện được chỉ định là đúng.

# **3. Định tuyến [MVC vs API]**

Routing

Về cốt lõi, định tuyến là cơ chế mà ASP.NET Core sử dụng để khớp các yêu cầu HTTP đến với các điểm cuối cụ thể (ví dụ: hành động của bộ điều khiển, Razor Pages hoặc trình xử lý API tối thiểu) trong ứng dụng của bạn. Điều này cho phép bạn xác định các URL rõ ràng và có ý nghĩa cho biết rõ các tài nguyên hoặc hành động đang được yêu cầu.

**Cách định tuyến hoạt động trong ASP.NET Core**

1. **Đăng ký điểm cuối:** Bạn xác định điểm cuối (tuyến đường) trong ứng dụng của mình, chỉ định:
   * Mẫu URL (ví dụ: /products, /api/orders/{id}).
   * (Các) phương thức HTTP mà endpoint xử lý (GET, POST, v.v.).
   * Mã để thực thi khi điểm cuối được khớp (RequestDelegate).
2. **Yêu cầu khớp (phần mềm trung gian):**
   * Thành phần phần mềm trung gian UseRouting được thêm vào quy trình.
   * Khi có yêu cầu, UseRouting sẽ phân tích URL đến và phương thức HTTP.
   * Nó so sánh URL với các điểm cuối đã đăng ký của bạn để tìm kết quả phù hợp nhất.
3. **Thực thi điểm cuối (Phần mềm trung gian):**
   * Thành phần phần mềm trung gian UseEndpoints được thêm vào quy trình, theo UseRouting.
   * Nếu UseRouting tìm thấy điểm cuối phù hợp, UseEndpoints sẽ thực thi mã (RequestDelegate) được liên kết với điểm cuối đó.

UseRouting so với UseEndpoints

* **UseRouting:**
  + Nó chịu trách nhiệm đối **sánh tuyến đường** - tìm điểm cuối phù hợp cho một yêu cầu nhất định.
  + Nó thêm dữ liệu tuyến vào HttpContext, mà phần mềm trung gian tiếp theo có thể sử dụng để đưa ra quyết định.
  + Nó **phải** đến trước UseEndpoints.
* **UseEndpoints:**
  + Nó chịu trách nhiệm thực **thi điểm cuối** - gọi mã (đại diện) được liên kết với điểm cuối phù hợp.
  + Nó cũng cho phép bạn định cấu hình các điểm cuối (ví dụ: xác định chính sách, bộ lọc) bằng cách sử dụng các biểu thức lambda.

Phương pháp Map\*: Tạo điểm cuối

ASP.NET Core cung cấp một họ các phương thức mở rộng Map\* trên giao diện IEndpointRouteBuilder giúp đơn giản hóa việc tạo điểm cuối:

* MapGet: Tạo một điểm cuối chỉ xử lý các yêu cầu GET.
* MapPost: Tạo một điểm cuối chỉ xử lý các yêu cầu POST.
* MapPut, MapDelete: Tạo điểm cuối cho các yêu cầu PUT và DELETE, tương ứng.
* MapMethods: Tạo một điểm cuối xử lý nhiều phương thức HTTP.
* MapControllerRoute, MapAreaControllerRoute: Được sử dụng để định cấu hình bộ điều khiển MVC / Razor Pages.
* MapFallbackToFile: Được sử dụng để chỉ định tệp mặc định để phục vụ khi không có điểm cuối nào khác khớp.

**Code: Giải thích chi tiết**

1. Bật định tuyến
2. App. UseRouting();
4. Tạo điểm cuối
5. App. UseEndpoints(điểm cuối =>
6. {
7. Thêm điểm cuối của bạn tại đây
8. điểm cuối. MapGet("map1", async (context) => {
9. chờ ngữ cảnh. Response.WriteAsync("Trong bản đồ 1");
10. });
12. điểm cuối. MapPost("map2", async (context) => {
13. chờ ngữ cảnh. Response.WriteAsync("Trong bản đồ 2");
14. });
15. });
17. App. Run(async context => {
18. chờ ngữ cảnh. Response.WriteAsync($"Yêu cầu nhận được tại {context. Request.Path}");
19. });
20. **App. UseRouting();:** Dòng này kích hoạt phần mềm trung gian định tuyến. Nó thiết lập bộ máy để phân tích các yêu cầu đến và so khớp chúng với các điểm cuối đã xác định của bạn.
21. **App. UseEndpoints(endpoints => { ... });:** Biểu thức lambda này định cấu hình các điểm cuối của ứng dụng của bạn:
    * điểm cuối. MapGet("map1", ...);: Đăng ký điểm cuối GET phản hồi đường dẫn "/map1" với văn bản "Trong bản đồ 1".
    * điểm cuối. MapPost("map2", ...);: Đăng ký điểm cuối POST cho đường dẫn "/map2", phản hồi bằng "In Map 2".
22. **App. Run(async context => { ... });:** Đây là phần mềm trung gian đầu cuối dự phòng. Nếu không có điểm cuối nào khác khớp với yêu cầu (ví dụ: nếu bạn truy cập "/map3"), nó sẽ thực thi mã này, ghi đường dẫn được yêu cầu đến phản hồi.

GetEndpoint()

Trong ASP.NET Core, phương thức GetEndpoint() là một công cụ mạnh mẽ để truy xuất thông tin về điểm cuối cụ thể đã được chọn để xử lý yêu cầu HTTP đến. Phương thức này là một phương thức mở rộng có sẵn trên đối tượng HttpContext.

* **Mục đích:** Nó cho phép bạn truy cập thông tin chi tiết về điểm cuối phù hợp, chẳng hạn như tên hiển thị, mẫu tuyến đường, siêu dữ liệu, v.v.
* **Khi nào nên sử dụng:** Bạn thường sử dụng GetEndpoint() trong các thành phần phần mềm trung gian để đưa ra quyết định dựa trên điểm cuối đã chọn hoặc để trích xuất thông tin có liên quan đến logic tùy chỉnh của bạn.
* **Vị trí phần mềm trung gian:** Phương thức GetEndpoint() sẽ trả về một đối tượng Endpoint hợp lệ **chỉ sau khi** phần mềm trung gian UseRouting đã thực thi và khớp thành công yêu cầu với một điểm cuối.

Mã:

1. var builder = WebApplication.CreateBuilder(args);
2. var app = trình tạo. Xây dựng();
4. Phần mềm trung gian 1: Trước khi định tuyến
5. App. Use(async (ngữ cảnh, tiếp theo) =>
6. {
7. Microsoft.AspNetCore.Http.Endpoint? endPoint = ngữ cảnh. GetEndpoint();
8. nếu (endPoint != null)
9. {
10. chờ ngữ cảnh. Response.WriteAsync($"Điểm cuối: {endPoint.DisplayName}\n");
11. }
12. await next(ngữ cảnh);
13. });
15. Bật phần mềm trung gian định tuyến
16. App. UseRouting();
18. Phần mềm trung gian 2: Sau khi định tuyến
19. App. Use(async (ngữ cảnh, tiếp theo) =>
20. {
21. Microsoft.AspNetCore.Http.Endpoint? endPoint = ngữ cảnh. GetEndpoint();
22. nếu (endPoint != null)
23. {
24. chờ ngữ cảnh. Response.WriteAsync($"Điểm cuối: {endPoint.DisplayName}\n");
25. }
26. await next(ngữ cảnh);
27. });
29. Tạo điểm cuối
30. App. UseEndpoints(điểm cuối =>
31. {
32. điểm cuối. MapGet("map1", async (ngữ cảnh) =>
33. {
34. chờ ngữ cảnh. Response.WriteAsync("Trong bản đồ 1");
35. });
37. điểm cuối. MapPost("map2", async (ngữ cảnh) =>
38. {
39. chờ ngữ cảnh. Response.WriteAsync("Trong bản đồ 2");
40. });
41. });
43. Phần mềm trung gian dự phòng
44. App. Run(ngữ cảnh không đồng bộ =>
45. {
46. chờ ngữ cảnh. Response.WriteAsync($"Yêu cầu nhận được tại {context. Request.Path}");
47. });
49. App. Chạy();

Hãy phân tích mã từng bước:

1. **Phần mềm trung gian 1 (Trước khi định tuyến):**
   * Ở đây, GetEndpoint() sẽ trả về null vì định tuyến chưa xảy ra. Yêu cầu chưa được khớp với bất kỳ điểm cuối cụ thể nào.
2. **App. UseRouting();**
   * Điều này cho phép phần mềm trung gian định tuyến, chịu trách nhiệm khớp yêu cầu với một điểm cuối.
3. **Phần mềm trung gian 2 (Sau khi định tuyến):**
   * Bây giờ, GetEndpoint() sẽ trả về đối tượng điểm cuối phù hợp (nếu tìm thấy kết quả khớp). Bạn có thể truy cập DisplayName (hoặc các thuộc tính khác) để lấy thông tin về điểm cuối đã chọn.

* Đối với yêu cầu GET đến "/map1", tên hiển thị sẽ là "map1".
* Đối với yêu cầu POST đến "/map2", tên hiển thị sẽ là "map2".
* Đối với bất kỳ đường dẫn nào khác, tên hiển thị sẽ là null (vì phần mềm trung gian dự phòng xử lý những trường hợp đó).

1. **Tạo điểm cuối:**
   * Ứng dụng. UseEndpoints xác định điểm cuối (tuyến đường) của bạn.
2. **Phần mềm trung gian dự phòng:**
   * Phần mềm trung gian này xử lý các yêu cầu không khớp với bất kỳ điểm cuối đã xác định nào. Nó chỉ đơn giản là viết đường dẫn được yêu cầu cho phản hồi.

Thông số tuyến đường

Thông số tuyến là trình giữ chỗ trong các mẫu URL của bạn để thu thập các giá trị từ các yêu cầu đến. Sau đó, các giá trị này có thể được sử dụng trong trình xử lý điểm cuối của bạn để tùy chỉnh phản hồi hoặc thực hiện các hành động cụ thể.

**Các loại thông số tuyến đường**

1. **Các thông số bắt buộc:**
   * **Cú pháp:** Được đặt trong dấu ngoặc nhọn {}.
   * **Hành vi:** Phải được cung cấp trong URL để tuyến trùng khớp. Nếu không có, yêu cầu sẽ không khớp với điểm cuối này.
   * **Ví dụ:** /products/{id} (Tham số id là bắt buộc).
2. **Các thông số tùy chọn:**
   * **Cú pháp:** Được đặt trong dấu ngoặc nhọn {} và theo sau là dấu chấm hỏi ?.
   * **Hành vi:** Có thể bỏ qua khỏi URL. Nếu không có, giá trị của tham số sẽ là null.
   * **Ví dụ:** /products/details/{id?} (Tham số id là tùy chọn).
3. **Các thông số với giá trị mặc định:**
   * **Cú pháp:** Được đặt trong dấu ngoặc nhọn {}, theo sau là dấu bằng =, sau đó là giá trị mặc định.
   * **Hành vi:** Nếu không được cung cấp trong URL, thông số sẽ lấy giá trị mặc định được chỉ định.
   * **Ví dụ:** /employee/profile/{EmployeeName=harsha} (Tham số EmployeeName mặc định là "harsha").

Mã:

1. // ... (UseRouting và phần mềm trung gian khác)...
3. App. UseEndpoints(điểm cuối =>
4. {
5. Các thông số bắt buộc
6. điểm cuối. Map("tệp/{tên tệp}.{ extension}", async context =>
7. {
8. xâu? fileName = Convert.ToString(ngữ cảnh. Request.RouteValues["tên tệp"]);
9. xâu? extension = Convert.ToString(ngữ cảnh. Request.RouteValues["phần mở rộng"]);
11. chờ ngữ cảnh. Response.WriteAsync($"Trong tệp - {fileName} - {phần mở rộng}");
12. });
14. Thông số mặc định
15. điểm cuối. Map("nhân viên/hồ sơ/{EmployeeName=harsha}", ngữ cảnh không đồng bộ =>
16. {
17. xâu? employeeName = Convert.ToString(ngữ cảnh. Request.RouteValues["employeename"]);
18. chờ ngữ cảnh. Response.WriteAsync($"Trong hồ sơ nhân viên - {employeeName}");
19. });
21. Thông số tùy chọn
22. điểm cuối. Map("products/details/{id?}", async context => {
23. if (ngữ cảnh. Request.RouteValues.ContainsKey("id"))
24. {
25. int id = Convert.ToInt32(ngữ cảnh. Request.RouteValues["id"]);
26. chờ ngữ cảnh. Response.WriteAsync($"Chi tiết sản phẩm - {id}");
27. }
28. khác
29. {
30. chờ ngữ cảnh. Response.WriteAsync($"Chi tiết sản phẩm - id không được cung cấp");
31. }
32. });
33. });
35. // ... (Phần mềm trung gian dự phòng)...
36. **Ví dụ về các thông số bắt buộc:**
    * Các tệp tuyến đường/{filename}. {extension} mong đợi cả tên tệp và phần mở rộng đều có trong URL (ví dụ: /files/sample.txt).
    * Trình xử lý điểm cuối trích xuất các giá trị này từ ngữ cảnh. Request.RouteValues và sử dụng chúng trong phản hồi.
37. **Ví dụ về tham số mặc định:**
    * Định tuyến employee/profile/{EmployeeName=harsha} có giá trị mặc định cho EmployeeName.
    * Nếu bạn truy cập /employee/profile, phản hồi sẽ là "Trong hồ sơ nhân viên - harsha".
    * Nếu bạn truy cập /employee/profile/john, phản hồi sẽ là "Trong hồ sơ nhân viên - john".
38. **Ví dụ tham số tùy chọn:**
    * Route products/details/{id?} cho phép bỏ qua tham số id.
    * Nếu bạn truy cập /products/details/123, nó sẽ hiển thị chi tiết sản phẩm cho ID 123.
    * Nếu bạn truy cập vào /products/details, điều đó sẽ cho biết rằng ID không được cung cấp.

Hạn chế tuyến đường

Các ràng buộc tuyến đường là một công cụ thiết yếu trong định tuyến ASP.NET Core cho phép bạn thêm xác thực bổ sung vào các thông số tuyến đường của mình. Chúng xác định các quy tắc hạn chế các giá trị mà tham số có thể chấp nhận, giúp bạn lọc ra các yêu cầu không hợp lệ trước khi chúng đến trình xử lý điểm cuối của bạn.

**Tại sao nên sử dụng ràng buộc tuyến đường?**

* **Xác thực nâng cao:** Đảm bảo rằng chỉ các yêu cầu có giá trị tham số hợp lệ mới được xử lý.
* **Cải thiện bảo mật:** Ngăn chặn đầu vào độc hại bằng cách từ chối các yêu cầu có giá trị có khả năng gây hại.
* **Mã sạch hơn:** Tránh làm lộn xộn trình xử lý điểm cuối của bạn bằng logic xác thực.
* **Định tuyến rõ ràng:** Làm cho các tuyến đường của bạn tự ghi lại nhiều hơn và dễ hiểu hơn.

Các ràng buộc tuyến đường phổ biến

ASP.NET Core cung cấp nhiều ràng buộc tuyến đường tích hợp:

* **int:** Yêu cầu giá trị tham số phải là số nguyên.
* **bool:** Yêu cầu giá trị tham số phải là boolean (true hoặc false).
* **datetime:** Yêu cầu giá trị tham số phải là chuỗi ngày và giờ hợp lệ.
* **decimal, double, float, long:** Yêu cầu giá trị tham số phải thuộc loại số được chỉ định.
* **guid:** Yêu cầu giá trị tham số phải là GUID hợp lệ (Mã định danh duy nhất toàn cầu).
* **alpha:** Yêu cầu giá trị tham số chỉ bao gồm các ký tự bảng chữ cái (a-z, A-Z).
* **regex:** Yêu cầu giá trị tham số khớp với mẫu biểu thức chính quy.
* **length:** Yêu cầu giá trị tham số phải có độ dài cụ thể hoặc trong một phạm vi cụ thể.
* **min, max, range:** Yêu cầu giá trị tham số lớn hơn hoặc bằng giá trị nhỏ nhất (min), nhỏ hơn hoặc bằng giá trị lớn nhất (max) hoặc trong một phạm vi (range) cụ thể.

Mã

1. // ... (UseRouting và phần mềm trung gian khác)...
3. App. UseEndpoints(điểm cuối =>
4. {
5. // ... (điểm cuối khác)...
7. Ràng buộc theo bảng chữ cái và độ dài
8. điểm cuối. Map("nhân viên/hồ sơ/{EmployeeName:length(4,7):alpha=harsha}", ngữ cảnh không đồng bộ =>
9. {
10. // ...
11. });

14. Số nguyên, Phạm vi và Ràng buộc Tùy chọn
15. điểm cuối. Map("products/details/{id:int:range(1,1000)?}", async context => {
16. // ...
17. });
19. Ràng buộc DateTime
20. điểm cuối. Map("daily-digest-report/{reportdate:datetime}", async context =>
21. {
22. // ...
23. });
25. Ràng buộc GUID
26. điểm cuối. Map("cities/{cityid:guid}", async context =>
27. {
28. // ...
29. });
31. Int, Min, Regex Ràng buộc
32. điểm cuối. Map("sales-report/{year:int:min(1900)}/{month:regex(^(apr|jul|oct|jan)$)}", async context =>
33. {
34. // ...
35. });
36. });
38. // ... (Phần mềm trung gian dự phòng)...
39. **Ràng buộc theo bảng chữ cái và độ dài:** /employee/profile/{EmployeeName:length(4,7):alpha=harsha}: Đảm bảo EmployeeName dài 4-7 ký tự và chỉ bao gồm các ký tự bảng chữ cái. Nếu không được cung cấp, nó mặc định là "harsha".
40. **Số nguyên, phạm vi và ràng buộc tùy chọn:** /products/details/{id:int:range(1,1000)?}: Yêu cầu id phải là số nguyên từ 1 đến 1000. Dấu chấm hỏi làm cho nó trở thành tùy chọn.
41. **Hạn chế DateTime:** /daily-digest-report/{reportdate:datetime}: Yêu cầu reportdate phải là một chuỗi ngày-giờ hợp lệ.
42. **Ràng buộc GUID:** /cities/{cityid:guid}: Yêu cầu cityid là GUID hợp lệ.
43. **Ràng buộc số nguyên, tối thiểu và biểu thức chính quy:** /sales-report/{year:int:min(1900)}/{month:regex(^(apr|jul|oct|jan)$)}: Yêu cầu year phải là một số nguyên lớn hơn hoặc bằng 1900 và month phải là một trong các giá trị được chỉ định (apr, jul, oct, jan).

Các lớp ràng buộc tuyến đường tùy chỉnh

Mặc dù ASP.NET Core cung cấp nhiều ràng buộc tuyến đường tích hợp, nhưng đôi khi ứng dụng của bạn yêu cầu các quy tắc xác thực chuyên biệt hơn. Các lớp ràng buộc tuyến tùy chỉnh cho phép bạn xác định tiêu chí của riêng mình để xác định xem giá trị tham số có hợp lệ hay không.

**Yêu cầu chính**

1. **Triển khai IRouteConstraint:** Tạo một lớp triển khai giao diện IRouteConstraint.
2. **Phương thức đối khớp:** Triển khai phương thức Match, phương thức này sẽ chứa logic xác thực tùy chỉnh của bạn. Phương pháp này nhận được một số thông số:
   * httpContext: HttpContext hiện tại.
   * route: Đối tượng IRouter được liên kết với tuyến đường.
   * routeKey: Tên của tham số tuyến đường đang được xác thực.
   * values: Từ điển chứa các giá trị tuyến đường.
   * routeDirection: Cho biết liệu tuyến đường đang được khớp cho yêu cầu đến hay để tạo URL.
3. **Trả về true hoặc false:** Phương thức Match phải trả về true nếu giá trị tham số hợp lệ theo ràng buộc của bạn và false nếu không.

Mã

1. MonthsCustomConstraint.cs
2. lớp công khai MonthsCustomConstraint : IRouteConstraint
3. {
4. public bool Match(HttpContext? httpContext, IRouter? route, string routeKey, RouteValueDictionary values, RouteDirection routeDirection)
5. {
6. Kiểm tra xem giá trị tham số có tồn tại không
7. if (!values. ContainsKey(khóa tuyến đường))
8. {
9. trả về false; Không phải là một trận đấu
10. }
12. Regex regex = new Regex("^(apr|jul|oct|jan)$");
13. xâu? monthValue = Convert.ToString(giá trị[routeKey]);
15. if (regex. IsMatch(giá trị tháng))
16. {
17. trả về true; Đó là một trận đấu
18. }
19. trả về false; Không phải là một trận đấu
20. }
21. }

Hãy chia nhỏ điều này:

1. **Triển khai IRouteConstraint:** Lớp MonthsCustomConstraint thực hiện rõ ràng giao diện này, báo hiệu rằng đó là một ràng buộc tuyến tùy chỉnh.
2. **Phương pháp phù hợp:**
   * Trước tiên, nó kiểm tra xem từ điển giá trị có chứa tham số route đang được xác thực (routeKey) hay không. Nếu không, đó là một sự không khớp ngay lập tức và false được trả về.
   * Biểu thức chính quy (^(apr|jul|oct|jan)$) được sử dụng để xác định các giá trị tháng hợp lệ.
   * Giá trị được liên kết với routeKey được truy xuất từ từ điển giá trị và được chuyển đổi thành chuỗi.
   * Phương thức Regex.IsMatch kiểm tra xem giá trị được truy xuất có khớp với mẫu tháng được phép hay không.
   * Trả về true nếu giá trị khớp và false nếu không.

**Sử dụng Ràng buộc Tùy chỉnh**

1. // ... (trong cấu hình điểm cuối của bạn)...
2. điểm cuối. Map("sales-report/{year:int:min(1900)}/{month:months}", async context =>
3. {
4. // ... logic trình xử lý điểm cuối của bạn...
5. });

* Lưu ý ràng buộc :months sau tham số month. Điều này cho biết rằng giá trị cho tháng phải được xác thực dựa trên lớp MonthsCustomConstraint.

Lựa chọn điểm cuối

Khi một yêu cầu đến ứng dụng ASP.NET Core của bạn, phần mềm trung gian định tuyến (UseRouting) sẽ phân tích phương thức URL và HTTP. Sau đó, nó so sánh thông tin này với bộ sưu tập các điểm cuối mà bạn đã xác định bằng cách sử dụng các phương thức như MapGet, MapPost, v.v. Mục tiêu là tìm điểm cuối phù hợp nhất để xử lý yêu cầu.

Tuy nhiên, điều gì sẽ xảy ra khi nhiều điểm cuối có vẻ giống như các điểm kết hợp tiềm năng? ASP.NET Core sử dụng một thuật toán được xác định rõ ràng để xác định điểm cuối chiến thắng.

**Thuật toán lựa chọn điểm cuối**

1. **Ưu tiên:**
   * **Đối sánh rõ ràng:** Các điểm cuối được xác định với các mẫu cụ thể hơn (ví dụ: /products/{id}) được ưu tiên hơn so với các điểm cuối có mẫu rộng hơn (ví dụ: /products).
   * **Thứ tự đăng ký:** Nếu nhiều điểm cuối có các mẫu cụ thể như nhau có thể khớp nhau, điểm cuối được đăng ký *đầu tiên* sẽ thắng.
2. **Phương thức HTTP:**
   * **Đối sánh chính xác:** Nếu phương thức yêu cầu (GET, POST, v.v.) khớp chính xác với phương thức được chỉ định cho một điểm cuối, điểm cuối đó được ưu tiên.
3. **Hạn chế tuyến đường:**
   * **Các ràng buộc cụ thể hơn:** Các điểm cuối có ràng buộc tuyến đường hạn chế hơn (ví dụ: id:int:range(1,100) so với id:int) được ưa chuộng.
4. **Catch-All (Dự phòng):**
   * Nếu không có điểm cuối nào khác khớp và bạn có điểm cuối bắt tất cả (được xác định bằng MapFallback), điểm cuối đó sẽ được chọn.

**Thứ tự ưu tiên: Tóm tắt trực quan**

1. Đối sánh rõ ràng với phương thức HTTP chính xác và các ràng buộc tuyến đường cụ thể hơn
2. Đối sánh rõ ràng với phương thức HTTP chính xác và các ràng buộc tuyến đường ít cụ thể hơn
3. Đối sánh rõ ràng với bất kỳ phương thức HTTP nào và các ràng buộc tuyến đường cụ thể hơn
4. Đối sánh rõ ràng với bất kỳ phương thức HTTP nào và các ràng buộc tuyến đường ít cụ thể hơn
5. Trình tự đăng ký (nếu tính cụ thể bằng nhau)
6. Điểm cuối Catch-All (nếu không tìm thấy kết quả trùng khớp nào khác)

**Ý nghĩa và mẹo thực tế**

* **Lưu ý đơn đặt hàng của bạn:** Hãy lưu ý đến thứ tự mà bạn đăng ký các điểm cuối của mình, đặc biệt nếu chúng có các mẫu tương tự.
* **Tính cụ thể chiến thắng:** Xác định các tuyến đường của bạn càng cụ thể càng tốt để tránh sự mơ hồ.
* **Ràng buộc tuyến đường:** Sử dụng ràng buộc tuyến đường để thu hẹp các giá trị hợp lệ cho các tham số.
* **Catch-All with Caution:** Catch-all endpoints có thể hữu ích, nhưng hãy sử dụng chúng một cách tiết kiệm để tránh các kết quả không mong muốn.
* **Siêu dữ liệu điểm cuối:** Khám phá siêu dữ liệu của đối tượng Điểm cuối để biết thông tin chi tiết về lý do tại sao một điểm cuối cụ thể được chọn.

Mã

1. App. UseEndpoints(điểm cuối =>
2. {
3. điểm cuối. MapGet("/products/{id:int}", GetProductById); Cụ thể nhất
4. điểm cuối. MapGet("/products", GetAllProducts); Ít cụ thể hơn
5. điểm cuối. MapGet("/{path?}", CatchAllHandler); Bắt tất cả
6. });

Trong ví dụ này:

* /products/123 sẽ khớp với điểm cuối đầu tiên (GetProductById).
* /products sẽ khớp với điểm cuối thứ hai (GetAllProducts).
* /anything-else sẽ khớp với điểm cuối catch-all (CatchAllHandler).

**Giải quyết sự mơ hồ**

Nếu hệ thống định tuyến không thể xác định dứt khoát kết quả phù hợp nhất, bạn sẽ gặp phải AmbiguousMatchException. Ngoại lệ này báo hiệu rằng bạn cần tinh chỉnh định nghĩa tuyến đường hoặc thứ tự đăng ký để loại bỏ xung đột.

Các tệp tĩnh trong ASP.NET Core

Tệp tĩnh là nội dung tạo nên cách trình bày trực quan và chức năng của ứng dụng web của bạn:

* **Tệp HTML:** Cấu trúc của các trang web của bạn.
* **CSS Stylesheets:** Kiểu dáng và giao diện nội dung của bạn.
* **Tệp JavaScript:** Các yếu tố tương tác và logic của ứng dụng của bạn.
* **Hình ảnh:** Các yếu tố hình ảnh giúp nâng cao trải nghiệm người dùng.

ASP.NET Core cung cấp thành phần phần mềm trung gian UseStaticFiles() để phục vụ hiệu quả các tệp tĩnh này trực tiếp đến trình duyệt mà không yêu cầu bất kỳ xử lý phía máy chủ nào.

**WebRoot: Vị trí mặc định**

Thuộc tính WebRoot trong ASP.NET Core chỉ định thư mục mặc định mà từ đó các tệp tĩnh được phân phối. Theo mặc định, thư mục này được đặt tên là "wwwroot" và nằm ở thư mục gốc của dự án của bạn. Tuy nhiên, bạn có thể tùy chỉnh vị trí này nếu cần.

**Phần mềm trung gian UseStaticFiles(): Kích hoạt phân phối tệp tĩnh**

* **Cách sử dụng cơ bản:** Ứng dụng gọi điện. UseStaticFiles (); không có đối số sẽ phục vụ các tệp tĩnh từ thư mục WebRoot mặc định.
* **Tùy chỉnh:** Bạn có thể tùy chỉnh hành vi của UseStaticFiles() bằng cách truyền một đối tượng StaticFileOptions:
  + FileProvider: Chỉ định một nhà cung cấp tệp khác (ví dụ: PhysicalFileProvider) để phục vụ tệp từ một vị trí tùy chỉnh.
  + RequestPath: Định cấu hình đường dẫn URL cơ sở cho các tệp tĩnh của bạn (ví dụ: /static).
  + ContentTypeProvider: Tùy chỉnh cách xác định các loại nội dung cho các phần mở rộng tệp khác nhau.
  + OnPrepareResponse: Thực hiện các hành động bổ sung đối với phản hồi trước khi phản hồi được gửi đến máy khách.

Mã

1. sử dụng Microsoft.Extensions.FileProviders;
3. // ...
5. var builder = WebApplication.CreateBuilder(WebApplicationOptions() mới
6. {
7. WebRootPath = "myroot"
8. });
9. var app = trình tạo. Xây dựng();
11. Phục vụ từ WebRoot được chỉ định ("myroot" trong trường hợp này)
12. App. UseStaticFiles ();
14. Phân phát từ thư mục tùy chỉnh ("mywebroot") nằm trong ContentRootPath của dự án
15. App. UseStaticFiles(StaticFileOptions mới()
16. {
17. FileProvider = PhysicalFileProvider mới (
18. Path.Combine(người xây dựng. Environment.ContentRootPath, "mywebroot")
19. )
20. });
21. // ... (phần còn lại của phần mềm trung gian và điểm cuối của bạn)...

**Lời giải thích**

1. **WebRoot tùy chỉnh:** Thuộc tính WebRootPath trong WebApplicationOptions được đặt thành "myroot", làm cho "myroot" trở thành vị trí mặc định cho các tệp tĩnh do ứng dụng đầu tiên phục vụ. UseStaticFiles().
2. **Tệp tĩnh mặc định:** Ứng dụng ban đầu. UseStaticFiles (); call phục vụ các tệp trực tiếp từ thư mục "myroot". Ví dụ, một yêu cầu đến /styles.css sẽ tìm kiếm một tệp có tên styles.css trong "myroot".
3. **Vị trí tệp tĩnh tùy chỉnh:** Ứng dụng thứ hai. UseStaticFiles định cấu hình PhysicalFileProvider để phục vụ tệp từ một vị trí tùy chỉnh: "mywebroot". Thư mục này nằm trong ContentRootPath của ứng dụng (thư mục gốc của dự án).

**Những cân nhắc quan trọng**

* **Bảo mật:** Luôn thận trọng với các tệp bạn hiển thị dưới dạng nội dung tĩnh. Tránh đặt thông tin nhạy cảm vào WebRoot hoặc thư mục tùy chỉnh của bạn.
* **Hiệu suất:** Cân nhắc sử dụng các kỹ thuật nén và bộ nhớ đệm để tối ưu hóa việc phân phối các tệp tĩnh.
* **Chính sách bảo mật nội dung (CSP):** Triển khai CSP để giảm thiểu các cuộc tấn công tập lệnh chéo trang web (XSS) có thể khai thác các tệp tĩnh của bạn.

Bằng cách quản lý hiệu quả các tệp tĩnh của bạn và sử dụng phần mềm trung gian UseStaticFiles(), bạn có thể nâng cao hiệu suất và trải nghiệm người dùng của ứng dụng ASP.NET Core của mình.

Các điểm chính cần nhớ:

**Routing**

* **Mục đích:** Khớp các yêu cầu HTTP đến với các điểm cuối cụ thể (bộ điều khiển, Trang Razor, API tối thiểu) trong ứng dụng của bạn.
* **Phần mềm trung gian:** UseRouting và UseEndpoints là các thành phần phần mềm trung gian cần thiết để định tuyến.
  + UseRouting: Phân tích URL yêu cầu và khớp nó với điểm cuối.
  + UseEndpoints: Thực thi mã của điểm cuối phù hợp.
* **Phương thức Map\*:** Được sử dụng để xác định điểm cuối cho các phương thức HTTP khác nhau (ví dụ: MapGet, MapPost, MapControllerRoute).

**Thứ tự lựa chọn điểm cuối**

* **Tính cụ thể:** Các tuyến cụ thể hơn (với nhiều tham số hoặc ràng buộc hơn) được ưu tiên hơn các tuyến ít cụ thể hơn.
* **Thứ tự đăng ký:** Nếu nhiều tuyến đường cụ thể như nhau, tuyến đường đăng ký đầu tiên sẽ thắng.
* **Phương thức HTTP:** Ưu tiên các tuyến đường có phương thức khớp chính xác.
* **Hạn chế tuyến đường:** Các tuyến đường có ràng buộc hạn chế hơn được ưu tiên.
* **Catch-All:** Điểm cuối dự phòng xử lý các yêu cầu không phù hợp.

**Thông số tuyến đường**

* **Loại:** Bắt buộc ({id}), tùy chọn ({id?}), giá trị mặc định ({id=123}).
* **Truy cập:** Các giá trị tham số được truy cập thông qua ngữ cảnh. Request.RouteValues.

**Hạn chế tuyến đường**

* **Mục đích:** Hạn chế các giá trị cho phép cho các tham số tuyến đường.
* **Tích hợp:** int, bool, datetime, guid, regex, length, min, max, range, v.v.
* **Tùy chỉnh:** Tạo các lớp triển khai IRouteConstraint để xác định logic xác thực của riêng bạn.

**GetEndpoint()**

* **Mục đích:** Truy xuất thông tin về điểm cuối phù hợp.
* **Cách sử dụng:** Ngữ cảnh cuộc gọi. GetEndpoint() trong phần mềm trung gian **sau** UseRouting.
* **Thông tin:** Truy cập các thuộc tính điểm cuối như DisplayName, mẫu tuyến đường và siêu dữ liệu.

**Tập tin tĩnh**

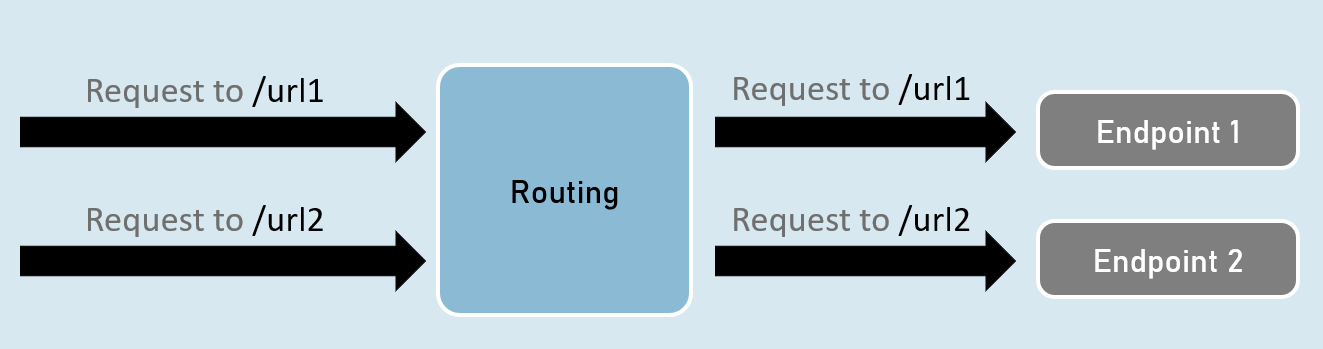
* **WebRoot:** Thư mục mặc định mà từ đó các tệp tĩnh được phục vụ (thường là "wwwroot").
* **UseStaticFiles():** Phần mềm trung gian để phục vụ các tệp tĩnh (HTML, CSS, JavaScript, hình ảnh).
* **Tùy chỉnh:** Sử dụng StaticFileOptions để thay đổi nhà cung cấp tệp, đường dẫn yêu cầu hoặc các cài đặt khác.

**Mẹo phỏng vấn chính**

* **Giải thích dòng chảy:** Nêu rõ cách một yêu cầu chảy qua phần mềm trung gian định tuyến và cách các điểm cuối được chọn.
* **Ví dụ về mã:** Hãy chuẩn bị để viết các đoạn mã minh họa đăng ký điểm cuối, cách sử dụng tham số và ứng dụng ràng buộc.
* **Khắc phục sự cố:** Giải thích cách bạn chẩn đoán và khắc phục các sự cố định tuyến phổ biến (ví dụ: lỗi 404, kết quả khớp không rõ ràng).
* **Các phương pháp hay nhất:** Thảo luận về cách thiết kế các tuyến đường sạch, có thể bảo trì và an toàn.

Giới thiệu về định tuyến

Định tuyến là một quá trình khớp các yêu cầu HTTP đến bằng cách kiểm tra phương thức HTTP và url; và sau đó gọi các điểm cuối tương ứng.



Định tuyến - UseRouting và UseEndPoints

**Sử dụng Định tuyến ( )**

App. UseRouting();

Cho phép định tuyến và chọn điểm cuối thích hợp dựa trên đường dẫn url và phương thức HTTP.

**UseEndPoints( )**

1. App. UseEndPoints(điểm cuối =>
2. {
3. điểm cuối. Bản đồ(...);
4. điểm cuối. Bản đồGet(...);
5. điểm cuối. MapPost(...);
6. );

Thực thi điểm cuối thích hợp dựa trên điểm cuối được chọn bởi phương thức UseRouting() ở trên.

Bản đồ, MapGet, MapPost

**điểm cuối. Bản đồ( )**

1. điểm cuối. Map("path", async (HttpContext context) =>
2. {
3. mã
4. });

Thực thi điểm cuối khi đường dẫn url của yêu cầu HTTP bắt đầu bằng đường dẫn được chỉ định.

**điểm cuối. MapGet( )**

1. điểm cuối. MapGet("path", async (HttpContext context) =>
2. {
3. mã
4. });

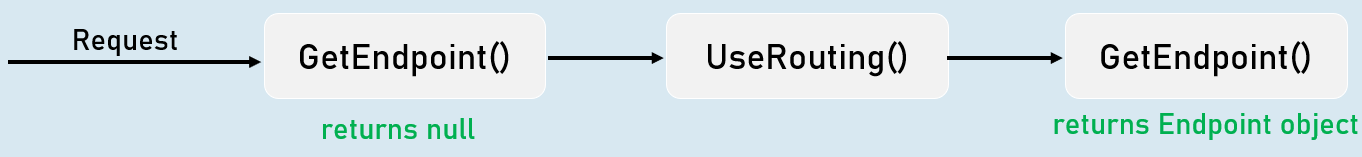
Thực thi điểm cuối khi đường dẫn url của yêu cầu HTTP GET bắt đầu bằng đường dẫn được chỉ định.

**điểm cuối. Bản đồ( )**

1. điểm cuối. MapPost("đường dẫn", async (ngữ cảnh HttpContext) =>
2. {
3. mã
4. });

Thực thi điểm cuối khi đường dẫn url của yêu cầu HTTP POST bắt đầu bằng đường dẫn được chỉ định.

GetEndpoint( )



ngữ cảnh. GetEndpoint();

Trả về một thực thể của loại Microsoft.AspNetCore.Http.Endpoint, đại diện cho một điểm cuối.

Thực thể đó chứa hai thuộc tính quan trọng: DisplayName, RequestDelegate.

Thông số tuyến đường

**"{tham số}"**

Tham số route có thể khớp với bất kỳ giá trị nào.



**Thông số tuyến đường mặc định**

"{tham số = default\_value}"

Tham số route có giá trị mặc định khớp với bất kỳ giá trị nào.

Nó cũng khớp với giá trị trống. Trong trường hợp này, giá trị mặc định sẽ được coi là tham số.

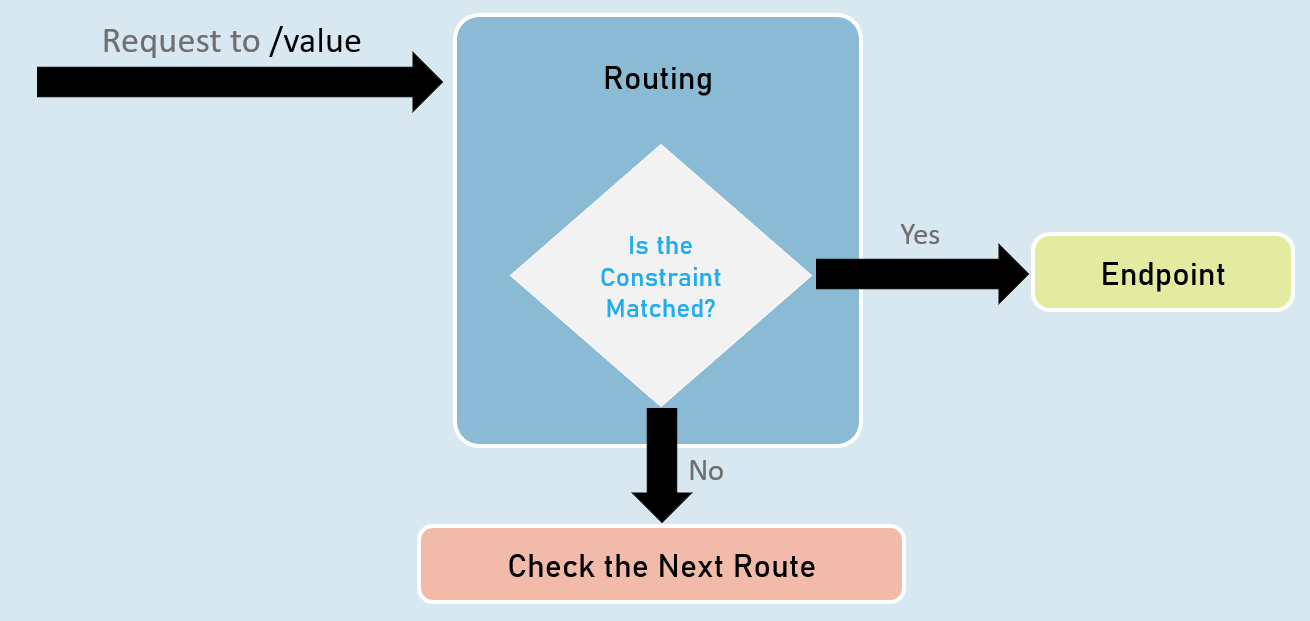
**Thông số tuyến đường tùy chọn**

"{tham số?}"

"?" cho biết một tham số tùy chọn.

Điều đó có nghĩa là, nó khớp với một giá trị hoặc giá trị trống.

Hạn chế tuyến đường



**Tham số tuyến đường có ràng buộc:**

"{tham số:ràng buộc}"

Tham số route có ràng buộc có thể khớp với giá trị thỏa mãn ràng buộc đã cho.

**Nhiều ràng buộc**

"{tham số: ràng buộc 1: ràng buộc 2}"

Một tham số route có thể có nhiều hơn một ràng buộc, được phân tách bằng dấu hai chấm ( : ).

**Int**

Khớp với bất kỳ số nguyên nào.

Ví dụ: {id:int} khớp với 123456789, -123456789

**Bool**

Kết hợp với true hoặc false. Không phân biệt chữ hoa chữ thường.

Ví dụ: {active:bool} khớp với true, false, TRUE, FALSE

**ngày/giờ**

Khớp với giá trị DateTime hợp lệ với định dạng "yyyy-MM-dd hh:mm:ss tt" và "MM/dd/yyyy hh:mm:ss tt".

Ví dụ: {id:datetime} khớp với 2030-01-01%2011:59%20pm

Lưu ý: '%20' bằng dấu cách.

**thập phân**

Khớp với giá trị thập phân hợp lệ.

Ví dụ: {price:decimal} khớp với 49,99, -1, 0,01

**dài**

Khớp với giá trị dài hợp lệ.

Ví dụ: {id:long} khớp với 123456789, -123456789

**Guid**

Khớp với giá trị Guid hợp lệ (Mã định danh duy nhất toàn cầu - Một số thập lục phân là duy nhất phổ quát).

Ví dụ: {id:guid} khớp với 123E4567-E89B-12D3-A456-426652340000

**minlength(giá trị)**

Khớp với một chuỗi có ít nhất số ký tự được chỉ định.

Ví dụ: {username:minlength(4)} khớp với John, Allen, William

**maxlength(giá trị)**

Khớp với một chuỗi có ít hơn hoặc bằng số ký tự được chỉ định.

Ví dụ: {username:maxlength(7)} khớp với John, Allen, William

**Chiều dài (Tối thiểu, Tối đa)**

Kết hợp với một chuỗi có số ký tự từ độ dài tối thiểu và tối đa nhất định (bao gồm cả hai số).

Ví dụ: {username:length(4, 7)} khớp với John, Allen, William

**chiều dài (giá trị)**

Khớp với một chuỗi có số ký tự được chỉ định chính xác.

Ví dụ: {tin:length(9)} khớp với 987654321

**min(giá trị)**

Khớp với giá trị số nguyên lớn hơn hoặc bằng giá trị được chỉ định.

Ví dụ: {age:min(18)} khớp với 18, 19, 100

**max(giá trị)**

Khớp với giá trị số nguyên nhỏ hơn hoặc bằng giá trị được chỉ định.

Ví dụ: {age:max(100)} khớp với -1, 1, 18, 100

**Phạm vi (Tối thiểu, Tối đa)**

Khớp với giá trị số nguyên giữa giá trị tối thiểu và giá trị lớn nhất được chỉ định (bao gồm cả hai số).

Ví dụ: {age:range(18,100)} khớp với 18, 19, 99, 100

**Alpha**

Kết hợp với một chuỗi chỉ chứa bảng chữ cái (A-Z) và (a-z).

Ví dụ: {username:alpha} khớp với rick, william

**regex(biểu thức)**

Khớp với một chuỗi khớp với biểu thức chính quy được chỉ định.

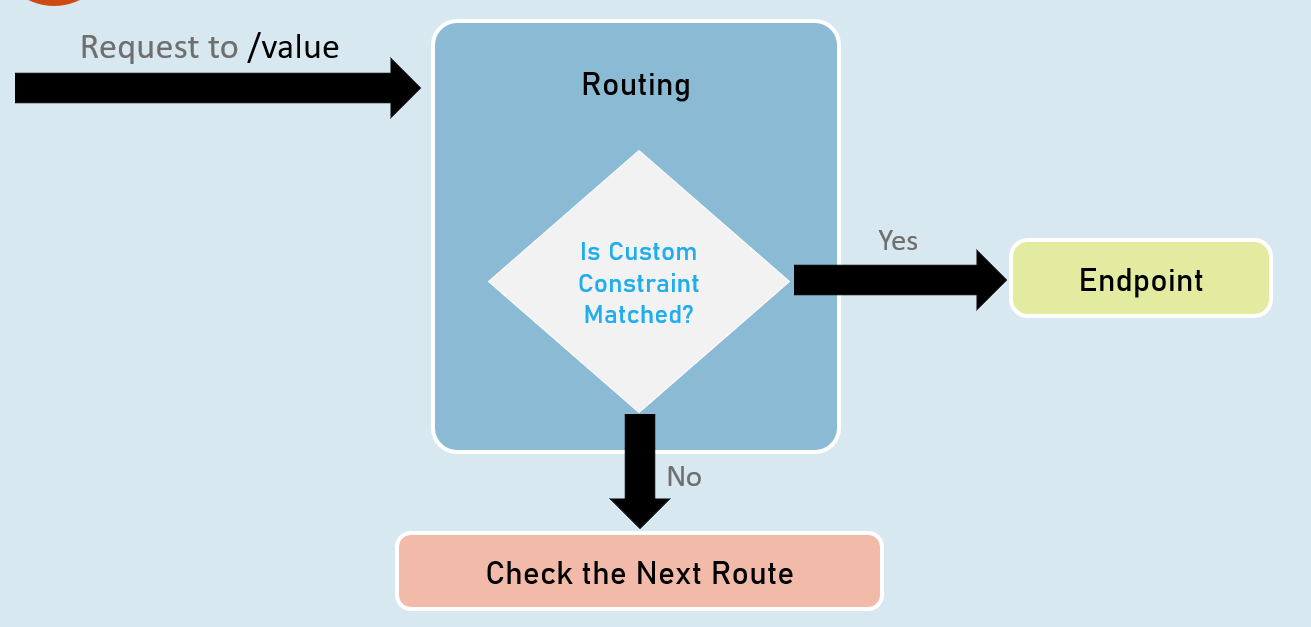
Ví dụ: {age:regex(^[0-9]{2}$)} khớp với bất kỳ số có hai chữ số nào, chẳng hạn như 10, 11, 98, 99

Ví dụ: 2: {age:regex(^\d{3}-\d{3}$)} khớp với bất kỳ số có ba chữ số nào, sau đó là dấu gạch nối và sau đó là số ba chữ số, chẳng hạn như 123-456

Các lớp ràng buộc tuyến đường tùy chỉnh

Lớp ràng buộc tuyến đường tùy chỉnh

1. lớp công khai Tên lớp: IRouteConstraint
2. {
3. public bool Match(HttpContext? HttpContext, IRouter? route, string routeKey, giá trị RouteValueDictionary, RouteDirection routeDirection)
4. {
5. Trả về true hoặc false
6. }
7. }
8. chủ thầu. Services.AddRouting(tùy chọn =>
9. {
10. Tùy chọn. ConstraintMap.Add("tên", loại(Tên lớp));
11. }); Thêm ràng buộc tùy chỉnh vào định tuyến



Thứ tự lựa chọn điểm cuối

Top là ưu tiên cao nhất (sẽ được đánh giá trước)

**1:** Mẫu URL có nhiều phân đoạn hơn.

Ví dụ: "a/b/c/d" cao hơn "a/b/c".

**2:** Mẫu URL có văn bản theo nghĩa đen được ưu tiên hơn phân đoạn tham số.

Ví dụ: "a/b" cao hơn "a/{tham số}".

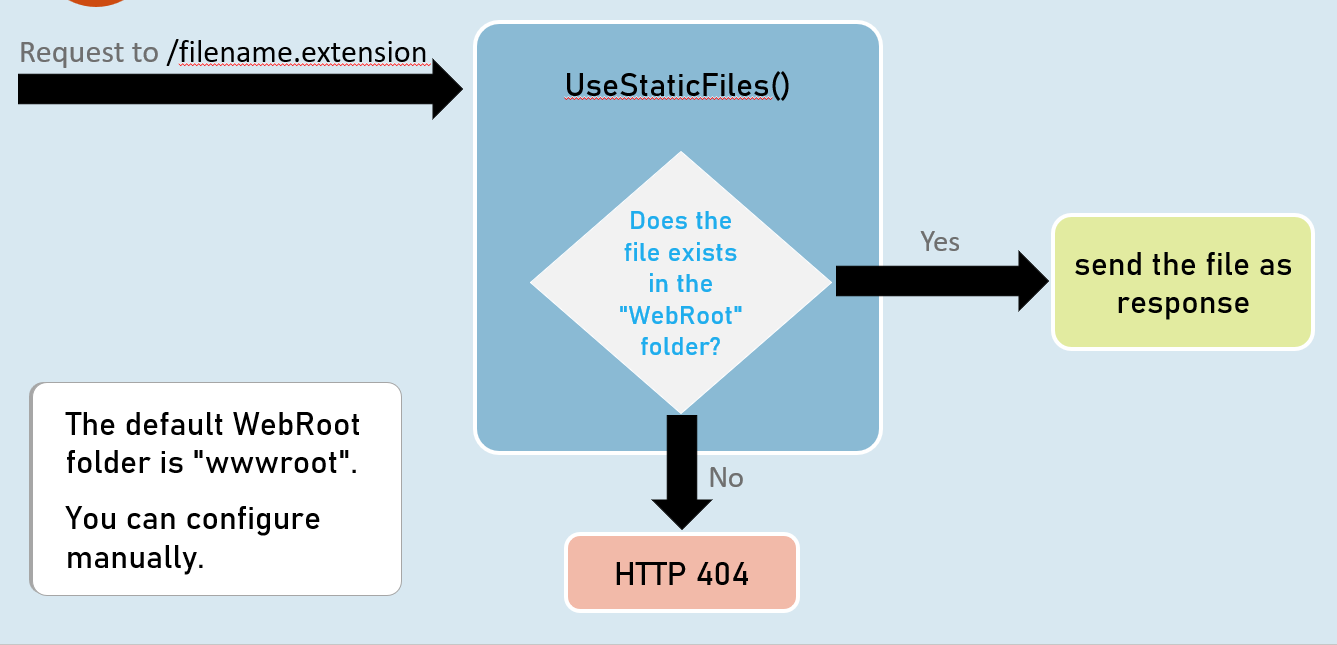
**3:** Mẫu URL có phân đoạn tham số có ràng buộc sẽ được ưu tiên hơn phân đoạn tham số không có ràng buộc.

Ví dụ: "a/b:int" cao hơn "a/b".

**4:** Tham số bắt tất cả (\*\*).

Ví dụ: "a/{b}" cao hơn "a/\*\*".

WebRoot



# **4. Bộ điều khiển và IActionResult [MVC]**

Bộ điều khiển và phương pháp hành động

Trong mẫu kiến trúc Model-View-Controller (MVC), controller đóng vai trò là người điều phối ứng dụng web của bạn. Họ xử lý các yêu cầu HTTP đến, tương tác với mô hình (lớp dữ liệu của bạn) và chọn chế độ xem thích hợp để hiển thị phản hồi trở lại cho người dùng.

* **Bộ điều khiển:** Các lớp nhóm các phương thức hành động liên quan và thường nằm trong thư mục Bộ điều khiển trong dự án của bạn.
* **Phương thức hành động:** Các phương thức công khai trong bộ điều khiển xử lý các yêu cầu cụ thể (ví dụ: hiển thị trang, xử lý dữ liệu biểu mẫu).

**Mục đích**

* **Sắp xếp logic:** Bộ điều khiển cung cấp một nhóm logic cho các hành động hoạt động trên cùng một loại dữ liệu hoặc chức năng.
* **Xử lý yêu cầu:** Họ chịu trách nhiệm xử lý các yêu cầu, truy xuất dữ liệu cần thiết và chuẩn bị phản hồi.
* **Chọn chế độ xem:** Bộ điều khiển thường chọn chế độ xem thích hợp để hiển thị, chuyển dữ liệu (mô hình) đến chế độ xem để trình bày.

**Cú pháp và quy ước**

* **Đặt tên lớp:** Tên lớp của Bộ điều khiển phải kết thúc bằng "Bộ điều khiển" (ví dụ: HomeController, ProductsController).
* **Kế thừa:** Bộ điều khiển kế thừa từ lớp cơ sở Controller (hoặc ControllerBase cho bộ điều khiển API).
* **Đặt tên phương thức hành động:** Phương thức hành động có thể có bất kỳ tên phương thức C# hợp lệ nào.
* **Các loại trả về:** Phương thức hành động có thể trả về nhiều loại khác nhau, bao gồm:
  + IActionResult: Một giao diện chung cho phép bạn trả về các loại kết quả khác nhau (chế độ xem, nội dung, chuyển hướng, v.v.).
  + string, int, v.v.: Đối với bộ điều khiển API, bạn có thể trả về dữ liệu thô.

Định tuyến thuộc tính

Định tuyến thuộc tính cho phép bạn xác định các tuyến trực tiếp trên các lớp điều khiển và phương thức hành động của bạn bằng cách sử dụng các thuộc tính:

* **Thuộc tính [Route]:** Chỉ định mẫu tuyến đường cơ sở cho bộ điều khiển hoặc hành động.
* [HttpGet], [HttpPost], v.v.: Cho biết (các) phương thức HTTP mà hành động nên xử lý.

**Trách nhiệm của người kiểm soát**

* **Xử lý yêu cầu:** Xử lý các yêu cầu đến và trích xuất dữ liệu liên quan (từ tham số tuyến đường, chuỗi truy vấn hoặc nội dung yêu cầu).
* **Tương tác mô hình:** Truy xuất dữ liệu từ mô hình của bạn (cơ sở dữ liệu, dịch vụ) hoặc cập nhật mô hình dựa trên yêu cầu.
* **Lựa chọn chế độ xem:** Xác định chế độ xem nào sẽ được hiển thị và cung cấp dữ liệu mô hình cần thiết cho chế độ xem.
* **Xử lý lỗi:** Xử lý lỗi một cách duyên dáng và trả về phản hồi thích hợp.

Mã

1. HomeController.cs
2. không gian tên ControllersExample.Controllers
3. {
4. [Controller] // Đánh dấu lớp làm controller
5. HomeController lớp công cộng
6. {
7. [Route("home")] // Tuyến đường cho hành động này
8. [Tuyến ("/")]
9. chuỗi công khai Index()
10. {
11. trả về "Hello from Index";
12. }
14. [Tuyến đường ("giới thiệu")]
15. chuỗi công khai Giới thiệu()
16. {
17. trả về "Xin chào từ Giới thiệu";
18. }
20. [Route("contact-us/{mobile:regex(^\\d{10}$)}")] // Tuyến đường có ràng buộc
21. chuỗi công khai Contact()
22. {
23. trả về "Xin chào từ Liên hệ";
24. }
25. }
26. }
28. Program.cs (hoặc Startup.cs)
29. var builder = WebApplication.CreateBuilder(args);
30. chủ thầu. Services.AddControllers(); Bật bộ điều khiển MVC
32. var app = trình tạo. Xây dựng();
33. App. UseRouting();
34. App. Bộ điều khiển bản đồ(); Kết nối bộ điều khiển với hệ thống định tuyến
35. App. Chạy();

* **HomeController:** Đây là lớp controller của bạn.
* **Chỉ mục, Giới thiệu, Liên hệ:** Đây là những phương thức hành động trong bộ điều khiển, mỗi phương thức có một lộ trình tương ứng.
* **Thuộc tính [Route]:** Xác định các tuyến đường cho từng phương thức hành động.
* **[Controller] Attribute:** Đánh dấu lớp là một controller, làm cho framework có thể phát hiện được.
* **chủ thầu. Services.AddControllers();:** Đăng ký các dịch vụ MVC và làm cho các bộ điều khiển có sẵn để chèn phụ thuộc.
* **App. MapControllers();:** Kết nối hệ thống định tuyến với bộ điều khiển của bạn, cho phép chúng xử lý các yêu cầu.

Kết quả nội dung

Trong lĩnh vực ASP.NET Core MVC, các phương thức hành động thường trả về các loại kết quả khác nhau: chế độ xem (HTML), dữ liệu JSON hoặc luồng tệp. Lớp ContentResult phục vụ cho một nhu cầu cụ thể: trả lại nội dung thô trực tiếp cho máy khách, mà không cần chi phí hiển thị chế độ xem đầy đủ. Nội dung này có thể là văn bản thuần túy, XML, JSON, CSV hoặc bất kỳ định dạng nào khác mà bạn chỉ định.

**Tại sao nên sử dụng ContentResult?**

* **Tính linh hoạt:** Bạn có toàn quyền kiểm soát nội dung bạn gửi và tiêu đề Content-Type, cho phép bạn điều chỉnh phản hồi theo các yêu cầu cụ thể của khách hàng.
* **Nhẹ:** ContentResult hiệu quả vì nó không liên quan đến việc hiển thị chế độ xem phức tạp.
* **Tính trực tiếp:** Lý tưởng cho các tình huống bạn muốn trả về tin nhắn văn bản đơn giản, phản hồi API hoặc định dạng nội dung tùy chỉnh.

Lớp ContentResult cung cấp các thuộc tính chính sau để định hình phản hồi của bạn:

1. **Nội dung:** Đây là nơi bạn đặt nội dung thực tế mà bạn muốn gửi lại cho khách hàng. Nó có thể là một chuỗi đơn giản, một đối tượng tuần tự hoặc bất kỳ dữ liệu nào bạn muốn truyền.
2. **ContentType:** Thuộc tính này rất quan trọng. Nó chỉ định loại MIME (Phần mở rộng thư Internet đa năng) của nội dung. Loại MIME cho máy khách biết cách diễn giải dữ liệu bạn đang gửi. Dưới đây là một số ví dụ phổ biến:
   * text/plain: Văn bản thuần túy
   * text/html: Nội dung HTML
   * application/json: Dữ liệu JSON
   * text/csv: Dữ liệu CSV
   * application/xml: Dữ liệu XML
3. **StatusCode (Tùy chọn):** Bạn có thể tùy chọn đặt mã trạng thái HTTP của phản hồi (ví dụ: 200 OK, 404 Not Found). Nếu không được chỉ định, nó mặc định là 200 OK.

**Tạo ContentResult**

Bạn có một số tùy chọn để tạo ContentResult trong các phương thức hành động của mình:

1. **Khởi tạo ContentResult:**
   1. trả về ContentResult() mới
   2. {
   3. Nội dung = "Xin chào từ chỉ mục",
   4. ContentType = "văn bản/đơn giản"
   5. };
2. **Sử dụng phương thức trợ giúp content():**
   1. return Content("Hello from Index", "text/plain");

Phương thức Content() là một phím tắt được cung cấp bởi lớp cơ sở Controller để tạo ContentResult một cách thuận tiện.

Mã

1. HomeController.cs (sửa đổi)
2. [Tuyến đường ("nhà")]
3. [Tuyến ("/")]
4. chỉ mục ContentResult() công khai
5. {
6. return Content("<h1>Welcome</h1> <h2>Hello from Index</h2>", "text/html");
7. }

Trong hành động Chỉ mục đã sửa đổi này:

1. **Nội dung HTML:** Nội dung được trả về là một chuỗi HTML chứa các thẻ tiêu đề.
2. **Loại nội dung:** ContentType được đặt thành "text/html", hướng dẫn trình duyệt hiển thị phản hồi dưới dạng HTML.
3. **Trải nghiệm khách hàng:** Khi người dùng điều hướng đến tuyến đường /home hoặc /, họ sẽ thấy một trang web có tiêu đề được định dạng "Chào mừng" và tiêu đề phụ "Xin chào từ chỉ mục".

Mặc dù việc trả về các chuỗi thô trực tiếp từ các phương thức hành động thường thuận tiện, nhưng việc sử dụng ContentResult cho phép bạn kiểm soát rõ ràng tiêu đề ContentType, điều này rất quan trọng để đảm bảo máy khách diễn giải chính xác dữ liệu phản hồi.

Kết quả Json

Lớp JsonResult trong ASP.NET Core MVC là công cụ phù hợp khi bạn cần trả về dữ liệu có cấu trúc ở định dạng JSON (Ký hiệu đối tượng JavaScript) từ các hành động của bộ điều khiển. JSON đã trở thành tiêu chuẩn trên thực tế để trao đổi dữ liệu trong API web và các ứng dụng web hiện đại do tính đơn giản, dễ đọc và hỗ trợ rộng rãi trên các nền tảng và ngôn ngữ.

**Tại sao nên sử dụng JsonResult?**

* **Định dạng tiêu chuẩn:** JSON là một định dạng được thiết lập tốt để biểu diễn dữ liệu có cấu trúc, lý tưởng cho giao tiếp giữa các ứng dụng web và API.
* **Tuần tự hóa:** ASP.NET Core tuần tự hóa liền mạch các đối tượng của bạn thành JSON, giúp bạn không phải định dạng thủ công.
* **Loại nội dung:** JsonResult tự động đặt tiêu đề Content-Type thành application/json, đảm bảo rằng máy khách (ví dụ: trình duyệt hoặc ứng dụng khác) diễn giải chính xác phản hồi.
* **Thân thiện với API:** Hoàn hảo để xây dựng API RESTful hoặc trả về dữ liệu cho JavaScript phía máy khách để sử dụng.

**Tạo JsonResult**

Tương tự như ContentResult, bạn có một số cách thuận tiện để tạo JsonResult trong các phương thức hành động của mình:

1. **Khởi tạo JsonResult:**
   1. trả về JsonResult(person) mới;

Ở đây, bạn chuyển đối tượng (ví dụ: person) mà bạn muốn tuần tự hóa thành JSON trực tiếp đến hàm khởi tạo JsonResult.

1. **Sử dụng phương thức trợ giúp json():**
   1. trả về Json(người);

Phương thức Json() là một cách viết tắt được cung cấp bởi lớp cơ sở Controller, giúp việc tạo JsonResult trở nên dễ dàng hơn.

Mã

1. HomeController.cs
2. [Lộ trình ("người")]
3. công khai JsonResult Person()
4. {
5. Người = người mới()
6. {
7. Id = Guid.NewGuid(),
8. Tên = "James",
9. Họ = "Smith",
10. Tuổi = 25
11. };
13. trả về Json(người);
14. }

Trong hành động Người đã sửa đổi này:

1. **Đối tượng Person:** Đối tượng Person được tạo với một số dữ liệu mẫu (bao gồm cả ID duy nhất).
2. **Tuần tự hóa JSON:** Cuộc gọi Json(person) tuần tự hóa đối tượng person thành một chuỗi JSON.
3. **Phản hồi:** Chuỗi JSON kết quả được trả về dưới dạng JsonResult, với tiêu đề Content-Type tự động được đặt thành application/json.

**Ra:**

Phản hồi được gửi đến khách hàng sẽ giống như sau:

1. {
2. "ID": "123e4567-e89b-12d3-a456-426614174000",
3. "firstName": "James",
4. "họ": "Smith",
5. "tuổi": 25
6. }

Kết quả tệp

Trong ASP.NET Core MVC, kết quả tệp là kết quả hành động được thiết kế để phục vụ tệp cho máy khách. Chúng đặc biệt hữu ích khi bạn muốn ứng dụng của mình cung cấp các tệp như PDF, hình ảnh, tài liệu hoặc nội dung nhị phân khác.

**Các loại kết quả tệp**

1. **Kết quả VirtualFile:**
   * **Mục đích:** Phục vụ tệp từ thư mục gốc web của ứng dụng (wwwroot theo mặc định) hoặc đường dẫn ảo.
   * **Thông số:**
     + virtualPath: Đường dẫn đến tệp trong web root hoặc đường dẫn ảo.
     + contentType: Loại MIME của tệp (ví dụ: application/pdf).
   * **Sử dụng:**
     + trả về VirtualFileResult("/sample.pdf", "application/pdf");
     + return File("/sample.pdf", "application/pdf"); (Phiên bản viết tắt)
   * **Lợi ích:** Cung cấp bảo mật bằng cách hạn chế quyền truy cập tệp vào gốc web hoặc đường dẫn ảo được định cấu hình.
2. **PhysicalFileResult:**
   * **Mục đích:** Phục vụ một tệp từ một đường dẫn tệp tuyệt đối trên hệ thống tệp của máy chủ.
   * **Thông số:**
     + physicalPath: Đường dẫn tuyệt đối đến tệp.
     + contentType: Loại MIME của tệp.
   * **Sử dụng:**
     + trả về PhysicalFileResult mới(@"c:\aspnetcore\sample.pdf", "application/pdf");
     + trả về PhysicalFile(@"c:\aspnetcore\sample.pdf", "application/pdf"); (Phiên bản viết tắt)
   * **Lợi ích:** Cho phép phân phát tệp từ các vị trí bên ngoài root web, nhưng yêu cầu xử lý cẩn thận do rủi ro bảo mật tiềm ẩn.
3. **FileContentResult:**
   * **Mục đích:** Phục vụ một tệp từ một mảng byte trong bộ nhớ.
   * **Thông số:**
     + fileContents: Nội dung tệp dưới dạng mảng byte.
     + contentType: Loại MIME của tệp.
   * **Sử dụng:**
     + byte[] byte = System.IO.File.ReadAllBytes(@"c:\aspnetcore\sample.pdf");
     + trả về mới FileContentResult(bytes, "application/pdf");
     + return File(bytes, "application/pdf"); (Phiên bản viết tắt)
   * **Lợi ích:** Hữu ích cho các tệp được tạo động hoặc khi bạn không muốn hiển thị đường dẫn thực tế của tệp.

Mã

1. HomeController.cs
2. [Lộ trình ("tải xuống tệp")]
3. công khai VirtualFileResult FileDownload()
4. {
5. return File("/sample.pdf", "application/pdf"); Phục vụ từ wwwroot
6. }
8. [Route("file-download2")]
9. công khai PhysicalFileResult FileDownload2()
10. {
11. trả về PhysicalFile(@"c:\aspnetcore\sample.pdf", "application/pdf"); Đường dẫn đầy đủ
12. }
14. [Lộ trình ("file-download3")]
15. công khai FileContentResult FileDownload3()
16. {
17. byte[] byte = System.IO.File.ReadAllBytes(@"c:\aspnetcore\sample.pdf");
18. return File(bytes, "application/pdf"); Byte trong bộ nhớ
19. }

**Những cân nhắc chính**

* **Bảo mật:** Hãy cực kỳ thận trọng khi sử dụng PhysicalFileResult để ngăn chặn truy cập trái phép vào hệ thống tệp của máy chủ của bạn. Xác thực đường dẫn một cách nghiêm ngặt và tránh để lộ thông tin nhạy cảm.
* **Hiệu suất:** Xem xét lưu vào bộ nhớ đệm kết quả tệp để cải thiện hiệu suất, đặc biệt là đối với các tệp lớn hơn hoặc nội dung được yêu cầu thường xuyên.
* **Bố trí nội dung:** Sử dụng thuộc tính FileDownloadName của kết quả tệp để đề xuất tên tệp cho trình duyệt khi người dùng tải xuống tệp.

**Chọn kết quả tệp phù hợp**

* **VirtualFileResult:** Khi tệp nằm trong root web của bạn và bạn không cần phải hiển thị đường dẫn tuyệt đối của nó.
* **PhysicalFileResult:** Khi bạn cần phân phối một tệp từ một vị trí tùy ý trên hệ thống tệp của máy chủ (sử dụng một cách thận trọng).
* **FileContentResult:** Khi bạn có nội dung tệp trong bộ nhớ (ví dụ: được tạo động) hoặc khi bạn không muốn tiết lộ đường dẫn thực tế của tệp.

Kết quả IActionResult

Giao diện IActionResult là một khái niệm cốt lõi trong ASP.NET Core MVC. Nó đóng vai trò là kiểu trả về cho các phương thức hành động trong bộ điều khiển của bạn, mang lại sự linh hoạt và cho phép bạn trả về các loại phản hồi khác nhau tùy thuộc vào ngữ cảnh của yêu cầu.

Về cơ bản, đó là một hợp đồng xác định một phương thức duy nhất:

1. Nhiệm vụ ExecuteResultAsync(ngữ cảnh ActionContext);

Phương thức này chịu trách nhiệm thực thi logic cụ thể được liên kết với kết quả hành động, tạo phản hồi HTTP thích hợp được gửi trở lại máy khách.

Các loại kết quả hành động

Dưới đây là bảng phân tích về một số loại kết quả hành động quan trọng nhất có nguồn gốc từ IActionResult:

* **ContentResult:** Trả về một chuỗi dưới dạng nội dung thô (văn bản, HTML, XML, v.v.).
  + Ví dụ: trả về Content("Hello from Index", "text/plain");
* **EmptyResult:** Đại diện cho một phản hồi trống (204 Không có nội dung).
  + Ví dụ: trả về new EmptyResult();
* **FileResult:** Được sử dụng để gửi tệp đến khách hàng (PDF, hình ảnh, v.v.). Đây là lớp cơ sở cho một số loại kết quả tệp cụ thể hơn.
  + VirtualFileResult: Phục vụ tệp từ gốc web hoặc đường dẫn ảo.
  + PhysicalFileResult: Phục vụ tệp từ đường dẫn vật lý trên máy chủ.
  + FileContentResult: Phục vụ một tệp từ một mảng byte trong bộ nhớ.
* **JsonResult:** Tuần tự hóa một đối tượng thành định dạng JSON và gửi nó dưới dạng phản hồi.
  + Ví dụ: return Json(new { message = "Success" });
* **RedirectResult:** Chuyển hướng người dùng đến một URL khác.
  + Ví dụ: trả về Redirect("/home");
* **RedirectToActionResult:** Chuyển hướng đến một phương thức hành động cụ thể trong bộ điều khiển.
  + Ví dụ: trả về RedirectToAction("Index", "Home");
* **ViewResult:** Hiển thị chế độ xem, thường là trang HTML, với dữ liệu mô hình tùy chọn.
  + Ví dụ: return View("Index", model);
* **PartialViewResult:** Hiển thị chế độ xem một phần (một phần có thể tái sử dụng của chế độ xem).
  + Ví dụ: trả về PartialView("\_ProductCard", product);
* **StatusCodeResult:** Trả về mã trạng thái HTTP cụ thể với một thông báo tùy chọn.
  + Ví dụ: trả về StatusCode(404, "Không tìm thấy tài nguyên");
* **BadRequestResult:** Viết tắt để trả về phản hồi 400 Bad Request.
* **NotFoundResult:** Viết tắt để trả về phản hồi 404 Not Found.
* **OkResult:** Viết tắt để trả về phản hồi 200 OK.

Mã

1. HomeController.cs
2. [Lộ trình ("sách")]
3. Chỉ số IActionResult công khai()
4. {
5. Mã sách nên được áp dụng
6. nếu (! Request.Query.ContainsKey("bookid"))
7. {
8. Response.StatusCode = 400; Đặt mã trạng thái theo cách thủ công
9. return Content("Id sách không được cung cấp");
10. }
12. // ... các kiểm tra xác thực khác ...
14. Nếu tất cả các kiểm tra đều vượt qua
15. return File("/sample.pdf", "application/pdf");
16. }

Trong phương pháp hành động này:

1. **Xác thực:** Mã thực hiện một số kiểm tra xác thực đối với tham số truy vấn bookid:
   * Nó kiểm tra xem tham số có tồn tại hay không.
   * Nó kiểm tra xem giá trị tham số có rỗng hay trống không.
   * Nó kiểm tra xem giá trị tham số có nằm trong phạm vi hợp lệ (1-1000) hay không.
   * Nó kiểm tra xem tham số truy vấn isloggedin có đúng hay không.
2. **Phản hồi lỗi:** Nếu bất kỳ xác thực nào không thành công, ContentResult sẽ được trả về với thông báo lỗi thích hợp và mã trạng thái 400 Yêu cầu không hợp lệ hoặc 401 Không được phép.
3. **Phản hồi thành công:** Nếu tất cả các xác thực vượt qua, FileResult sẽ được trả về, phục vụ tệp sample.pdf từ root web.

**Ghi chú**

* **Tính linh hoạt:** IActionResult cho phép bạn trả về các loại phản hồi khác nhau dựa trên logic trong hành động của bạn.

Kết quả mã trạng thái

Trong giao tiếp web, điều quan trọng là phải thông báo cho khách hàng về kết quả yêu cầu của họ. Mã trạng thái cung cấp một cách tiêu chuẩn hóa để truyền tải thông tin này. ASP.NET Core MVC cung cấp một loạt các kết quả hành động được thiết kế đặc biệt để trả về các mã trạng thái này cùng với các thông báo tùy chọn.

**Kết quả mã trạng thái phổ biến**

* **OkResult:** Cho biết một yêu cầu thành công (HTTP 200).
* **BadRequestResult:** Cho biết lỗi máy khách (HTTP 400). Thường được sử dụng cho đầu vào không hợp lệ.
* **NotFoundResult:** Cho biết rằng không tìm thấy tài nguyên được yêu cầu (HTTP 404).
* **UnauthorizedResult:** Cho biết yêu cầu yêu cầu xác thực (HTTP 401).
* **ForbiddenResult:** Cho biết rằng người dùng không được phép truy cập tài nguyên (HTTP 403).
* **StatusCodeResult:** Cho phép bạn trả về bất kỳ mã trạng thái HTTP tùy ý nào.

Sử dụng kết quả mã trạng thái

1. **Khởi tạo trực tiếp:**
2. trả về BadRequestResult() mới; Trả về HTTP 400
3. trả về notFoundResult() mới; Trả về HTTP 404
4. **Phương pháp trợ giúp:**
5. trả về BadRequest(); Trả về HTTP 400
6. trả về NotFound(); Trả về HTTP 404
7. trả lại Unauthorized(); Trả về HTTP 401
8. trả về StatusCode(403); Trả về HTTP 403
9. **Với tin nhắn:**
10. return BadRequest("Dữ liệu đầu vào không hợp lệ");
11. return NotFound("Không tìm thấy tài nguyên");

Các phương thức trợ giúp này ngắn gọn và biểu cảm hơn so với việc thực hiện trực tiếp các đối tượng kết quả.

Mã

1. HomeController.cs
2. [Lộ trình ("sách")]
3. Chỉ số IActionResult công khai()
4. {
5. // ... (kiểm tra xác thực tương tự như ví dụ trước)...
7. nếu (bookId <= 0)
8. {
9. return BadRequest("Mã sách không được nhỏ hơn hoặc bằng không");
10. }
12. Lưu ý việc sử dụng NotFound ở đây
13. nếu (bookId > 1000)
14. {
15. return NotFound("Book id không được lớn hơn 1000");
16. }
18. if (Convert.ToBoolean(Request.Query["isloggedin"]) == false)
19. {
20. trả về StatusCode(401); Mã trạng thái có thể tùy chỉnh
21. }
23. trả về File("/sample.pdf", "application/json");
24. }

Trong ví dụ tinh chỉnh này, logic xác thực vẫn giữ nguyên. Tuy nhiên, chúng tôi đã thực hiện các thay đổi sau:

1. **Mã trạng thái cụ thể:**
   * Chúng tôi sử dụng BadRequest() cho đầu vào không hợp lệ (ví dụ: bookId nhỏ hơn hoặc bằng không).
   * Chúng tôi sử dụng NotFound() khi bookId nằm ngoài phạm vi hợp lệ (lớn hơn 1000), vì nó có thể ngụ ý rằng cuốn sách được yêu cầu không tồn tại.
2. **Mã trạng thái có thể tùy chỉnh:**
   * Đối với trường hợp xác thực thất bại (isloggedin là false), chúng tôi sử dụng StatusCode(401) để trả về mã trạng thái 401 Unauthorized tiêu chuẩn. Bạn cũng có thể sử dụng return Unauthorized(); như một lối tắt.

**Ghi chú**

* **Thông báo cho khách hàng:** Mã trạng thái rất cần thiết để truyền đạt kết quả của yêu cầu cho khách hàng.
* **Mã tiêu chuẩn:** Sử dụng mã trạng thái HTTP tiêu chuẩn bất cứ khi nào có thể để đảm bảo tính nhất quán và khả năng tương tác.
* **Phương thức trợ giúp:** Tận dụng các phương thức trợ giúp (BadRequest, NotFound, v.v.) để có mã gọn gàng và biểu cảm hơn.
* **Tùy chỉnh:** Kết quả StatusCode cho phép bạn trả về bất kỳ mã trạng thái HTTP nào bạn cần, nhưng hãy sử dụng nó một cách thận trọng.
* **Ngoài xác thực:** Mã trạng thái không chỉ để xác thực; hãy sử dụng chúng để báo hiệu kết quả của bất kỳ hành động nào trong API của bạn.

Kết quả chuyển hướng

Kết quả chuyển hướng là kết quả hành động trong ASP.NET Core MVC hướng dẫn trình duyệt của máy khách điều hướng đến một URL mới. Điều này thường được sử dụng sau các hành động như gửi biểu mẫu, đăng nhập hoặc các thao tác khác mà bạn muốn chuyển người dùng sang một trang khác.

**Các loại kết quả chuyển hướng**

1. **Kết quả chuyển hướng:**
   * **Mục đích:** Chuyển hướng đến một URL được chỉ định (tuyệt đối hoặc tương đối).
   * **Thông số:**
     + url: URL để chuyển hướng đến.
     + permanent: Một boolean cho biết chuyển hướng là vĩnh viễn (301 Đã di chuyển vĩnh viễn) hay tạm thời (302 Đã tìm thấy). Mặc định là false (tạm thời).
   * **Sử dụng:**
     + return Redirect("/home"); (Tạm thời)
     + trả về RedirectPermanent("/home"); (Vĩnh viễn)
2. **RedirectToActionResult:**
   * **Mục đích:** Chuyển hướng đến một phương thức hành động cụ thể trong bộ điều khiển.
   * **Thông số:**
     + actionName: Tên của phương thức hành động.
     + controllerName: Tên của bộ điều khiển (tùy chọn, mặc định là bộ điều khiển hiện tại).
     + routeValues: Một đối tượng chứa các giá trị tuyến đường để chuyển đến hành động (tùy chọn).
     + permanent: Một boolean cho biết chuyển hướng là vĩnh viễn (301) hay tạm thời (302).
   * **Sử dụng:**
     + trả về RedirectToAction("Index"); (Tạm thời, cùng một bộ điều khiển)
     + return RedirectToAction("Details", "Products", new { id = 123 }); (Tạm thời, với giá trị tuyến đường)
     + trả về RedirectToActionPermanent("Giới thiệu"); (Vĩnh viễn)
3. **LocalRedirectResult:**
   * **Mục đích:** Chuyển hướng đến một URL cục bộ trong cùng một ứng dụng.
   * **Thông số:**
     + localUrl: URL cục bộ để chuyển hướng đến.
     + permanent: Một boolean cho biết chuyển hướng là vĩnh viễn (301) hay tạm thời (302).
   * **Sử dụng:**
     + trả lại LocalRedirect("/products/details/456"); (Tạm thời)
     + trả về LocalRedirectPermanent("/about"); (Vĩnh viễn)

Mã

1. HomeController.cs
2. [Lộ trình ("hiệu sách")]
3. Chỉ số IActionResult công khai()
4. {
5. // ... logic xác thực (giống như ví dụ trước)...
7. Chuyển hướng có điều kiện
8. nếu (someConditionIsTrue)
9. {
10. return RedirectToAction("Books", "Store", new { id = bookId }); Tạm thời, cho một hành động khác
11. }
12. khác
13. {
14. trả lại LocalRedirectPermanent($"store/books/{bookId}"); Chuyển hướng cục bộ vĩnh viễn
15. }
17. // ... các ví dụ chuyển hướng khác ...
18. }

**Giải thích về các loại chuyển hướng**

* **302 Đã tìm thấy (RedirectResult hoặc RedirectToActionResult với vĩnh viễn: false):**
  + Chuyển hướng tạm thời tiêu chuẩn. Yêu cầu trình duyệt tìm nạp tài nguyên mới, nhưng các yêu cầu trong tương lai vẫn nên sử dụng URL gốc.
* **301 Đã di chuyển vĩnh viễn (RedirectResult, RedirectToActionResult hoặc LocalRedirectResult với vĩnh viễn: true):**
  + Cho biết tài nguyên đã được di chuyển vĩnh viễn. Trình duyệt nên cập nhật dấu trang/liên kết của nó và các yêu cầu trong tương lai nên sử dụng URL mới.
* **LocalRedirectResult:**
  + Cụ thể cho các chuyển hướng trong cùng một ứng dụng. Giúp ngăn chặn chuyển hướng mở, trong đó tác nhân độc hại có thể lừa trang web của bạn chuyển hướng đến một trang web có hại bên ngoài.

**Chọn chuyển hướng phù hợp**

* **Bên ngoài so với nội bộ:** Sử dụng RedirectResult cho URL bên ngoài và LocalRedirectResult cho URL nội bộ.
* **Tạm thời so với vĩnh viễn:** Sử dụng 301 cho các bước di chuyển vĩnh viễn, 302 cho các lần di chuyển tạm thời (ví dụ: sau khi gửi biểu mẫu).
* **Hành động cụ thể:** Sử dụng RedirectToActionResult khi bạn muốn chuyển hướng đến một hành động cụ thể trong ứng dụng của mình.
* **An toàn:** Ưu tiên LocalRedirectResult hơn RedirectResult cho chuyển hướng nội bộ để bảo vệ chống lại các cuộc tấn công chuyển hướng mở.

Những điểm chính cần nhớ:

**1. Bộ điều khiển**

* **Mục đích:**
  + Xử lý các yêu cầu HTTP.
  + Tương tác với mô hình (lớp dữ liệu).
  + Chọn chế độ xem thích hợp để hiển thị phản hồi.
* **Đặt tên:** Kết thúc bằng "Bộ điều khiển" (ví dụ: HomeController).
* **Kế thừa:** Kế thừa từ Controller (hoặc ControllerBase cho API).
* **Phương thức hành động:** Các phương thức công khai trong bộ điều khiển xử lý các yêu cầu cụ thể.
* **Định tuyến thuộc tính:** Sử dụng [Route], [HttpGet], [HttpPost], v.v., để xác định các tuyến đường.

**2. Kết quả IActionResult**

* **Mục đích:** Loại trả về linh hoạt cho các phương thức hành động, cho phép nhiều loại phản hồi khác nhau.
* **Loại:**
  + **Dựa trên nội dung:**
    - ContentResult: Nội dung thô (văn bản, HTML, JSON, v.v.).
    - JsonResult: Dữ liệu JSON tuần tự.
    - FileResult (và các loại phụ): Tệp (PDF, hình ảnh, v.v.).
  + **Redirection:**
    - RedirectResult: Chuyển hướng đến bất kỳ URL nào.
    - RedirectToActionResult: Chuyển hướng đến một hành động cụ thể trong ứng dụng của bạn.
    - LocalRedirectResult: Chuyển hướng đến một URL cục bộ trong cùng một ứng dụng.
  + **Mã trạng thái:**
    - StatusCodeResult: Bất kỳ mã trạng thái HTTP tùy ý nào.
    - BadRequestResult, NotFoundResult, UnauthorizedResult, v.v.: Mã trạng thái cụ thể.
  + **Views:**
    - ViewResult: Hiển thị chế độ xem đầy đủ.
    - PartialViewResult: Hiển thị chế độ xem một phần.

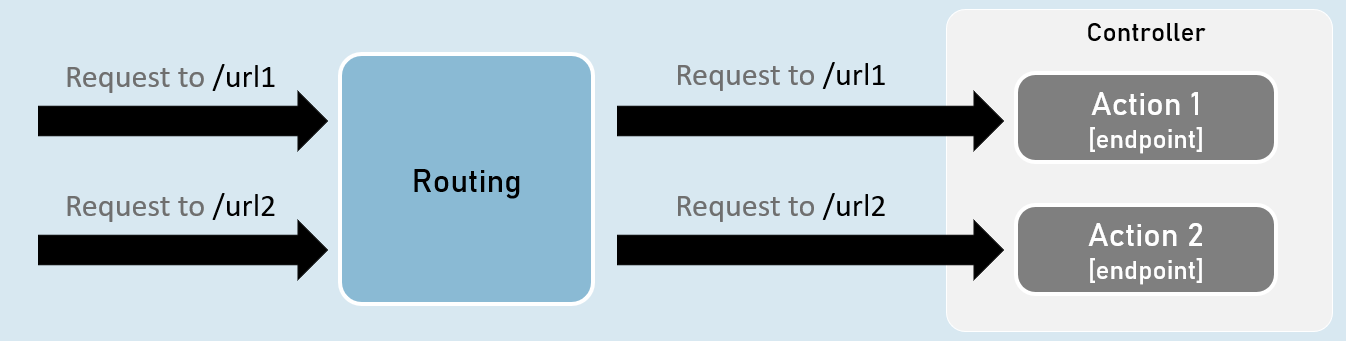
**Mẹo phỏng vấn chính**

* **Hiểu MVC:** Có thể giải thích vai trò của các mô hình, chế độ xem và bộ điều khiển.
* **Chọn kết quả hành động:** Giải thích lý do tại sao bạn chọn loại kết quả hành động này thay vì loại kết quả khác dựa trên kết quả mong muốn.
* **Mã trạng thái:** Biết các mã trạng thái HTTP phổ biến và ý nghĩa của chúng (200 OK, 404 Not Found, v.v.).
* **Định tuyến thuộc tính:** Thể hiện khả năng xác định tuyến đường của bạn bằng cách sử dụng các thuộc tính.

Giới thiệu về bộ điều khiển

Controller là một lớp được sử dụng để nhóm một tập hợp các hành động (*hoặc phương thức hành động*).

Các phương thức hành động thực hiện một số hoạt động nhất định khi nhận được yêu cầu và trả về kết quả (phản hồi).



**Tạo bộ điều khiển**

Bộ điều khiển phải là một trong hai hoặc cả hai:

* Tên lớp phải được đặt hậu tố là "Controller". Ví dụ: HomeController
* Thuộc tính [Controller] được áp dụng cho cùng một lớp hoặc cho lớp cơ sở của nó.

**Điều khiển**

1. [Bộ điều khiển]
2. lớp ClassNameController
3. {
4. Phương pháp hành động ở đây
5. }

**Tùy chọn:**

* Là một lớp học công cộng.
* Kế thừa từ Microsoft.AspNetCore.Mvc.Controller.

**Bật 'định tuyến' trong bộ điều khiển**

**Bộ điều khiển bổ sung( )**

chủ thầu. Services.AddControllers();

Thêm tất cả bộ điều khiển dưới dạng dịch vụ trong IServiceCollection.

Vì vậy, chúng có thể được truy cập khi một điểm cuối cụ thể cần nó.

**Bộ điều khiển bản đồ()**

App. Bộ điều khiển bản đồ();

Thêm tất cả các phương thức hành động làm điểm cuối.

Vì vậy, không cần sử dụng phương thức UseEndPoints() để thêm các phương thức hành động làm điểm cuối.

**Trách nhiệm của Kiểm soát viên**

**Yêu cầu đọc**

Trích xuất các giá trị dữ liệu từ yêu cầu như tham số chuỗi truy vấn, nội dung yêu cầu, cookie yêu cầu, tiêu đề yêu cầu, v.v.

**Gọi các mô hình**

Gọi các phương thức logic nghiệp vụ.

Nói chung, các hoạt động kinh doanh có sẵn dưới dạng 'dịch vụ'.

**Xác nhận**

Xác thực chi tiết yêu cầu đến (tham số chuỗi truy vấn, nội dung yêu cầu, cookie yêu cầu, tiêu đề yêu cầu, v.v.)

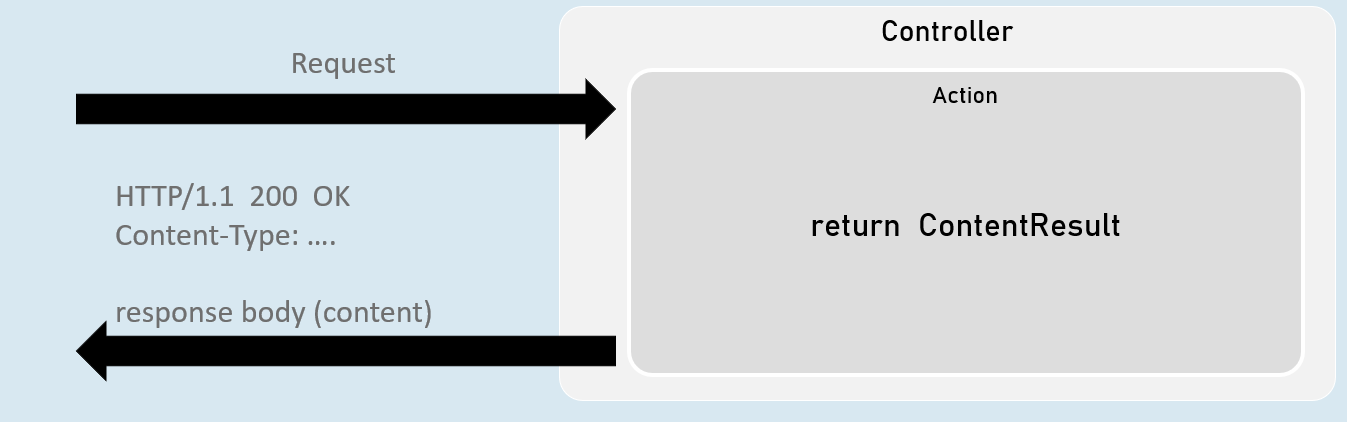
**Chuẩn bị phản hồi**

Chọn loại phản hồi nào phải được gửi đến khách hàng và cũng chuẩn bị phản hồi (*kết quả hành động*).

Kết quả nội dung

ContentResult có thể đại diện cho bất kỳ loại phản hồi nào, dựa trên loại MIME được chỉ định.

Loại MIME đại diện cho loại nội dung như text/plain, text/html, application/json, application, xml, application/pdf, v.v.



return new ContentResult() { Content = "content", ContentType = "content type" };

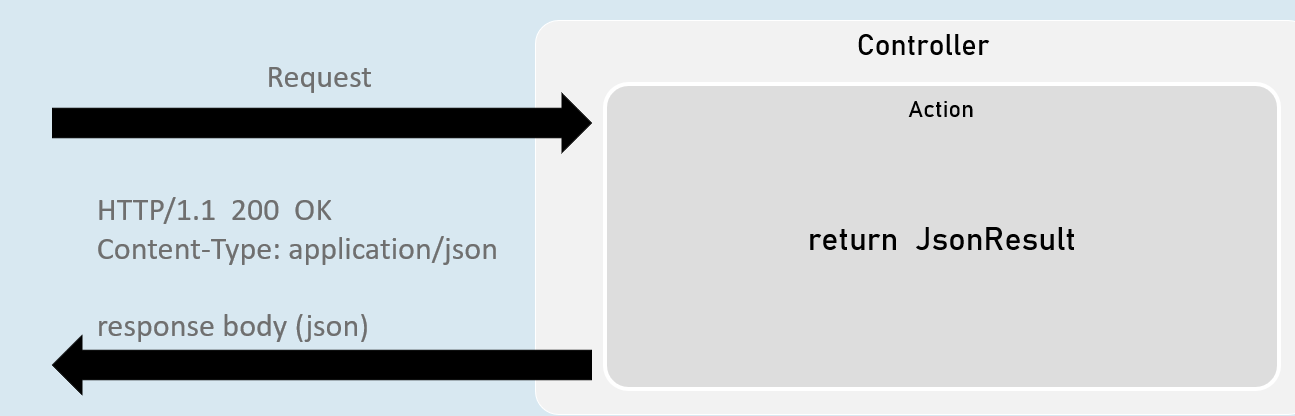
[hoặc]

trả về Content("content", "content type");

Kết quả Json

JsonResult có thể đại diện cho một đối tượng ở định dạng Ký hiệu đối tượng JavaScript (JSON).

Ví dụ: { "firstName": "James", "lastName": "Smith", "age": 25 }



trả về JsonResult(your\_object) mới;

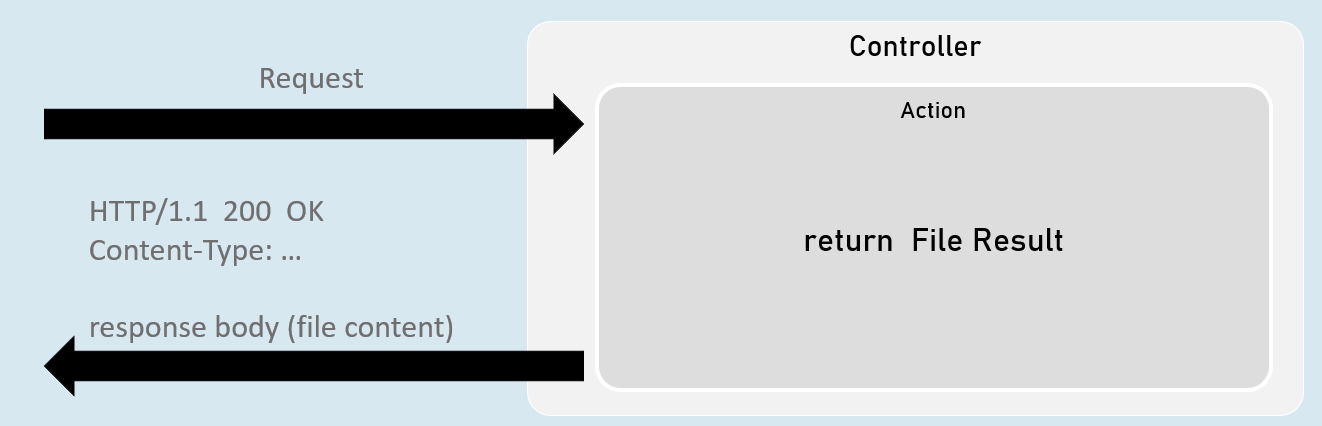
[hoặc]

trả về Json(your\_object);

Kết quả tệp

Kết quả tệp gửi nội dung của tệp dưới dạng phản hồi.

Ví dụ: tệp pdf, tệp txt, tệp exe, tệp zip, v.v.



**Kết quả VirtualFile**

trả về new VirtualFileResult("đường dẫn tương đối tệp", "loại nội dung");

hoặc

return File("đường dẫn tương đối tệp", "loại nội dung");

Đại diện cho một tệp trong thư mục WebRoot ('wwwroot' theo mặc định).

Được sử dụng khi tệp có trong thư mục WebRoot.

**Kết quả tệp vật lý**

Đại diện cho một tệp không nhất thiết phải là một phần của thư mục dự án.

Được sử dụng khi tệp có mặt bên ngoài thư mục WebRoot.

trả về new PhysicalFileResult("đường dẫn tuyệt đối tệp", "loại nội dung");

hoặc

return PhysicalFile("đường dẫn tuyệt đối tệp", "loại nội dung");

**FileContentResult**

Đại diện cho một tệp từ byte[ ].

Được sử dụng khi một phần của tệp hoặc byte[ ] từ nguồn dữ liệu khác phải được gửi dưới dạng phản hồi.

trả về mới FileContentResult(byte\_array, "loại nội dung");

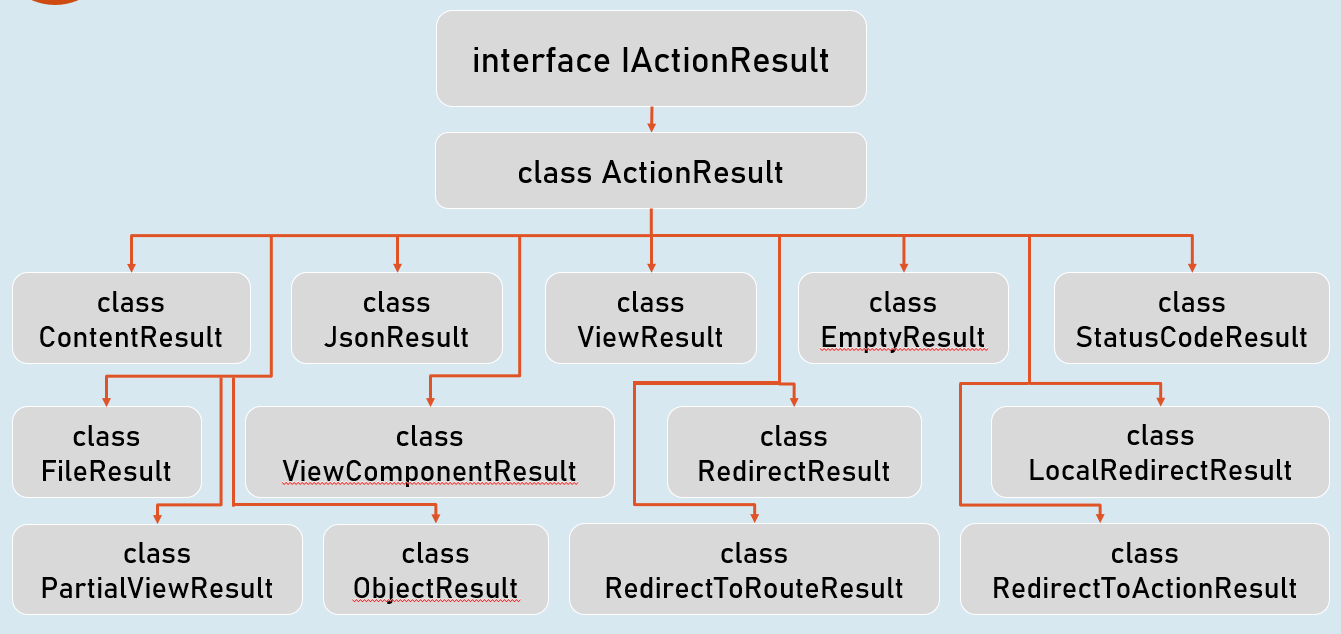
hoặc

return File(byte\_array, "loại nội dung");

Kết quả IActionResult

Nó là giao diện cha cho tất cả các lớp kết quả hành động như ContentResult, JsonResult, RedirectResult, StatusCodeResult, ViewResult, v.v.

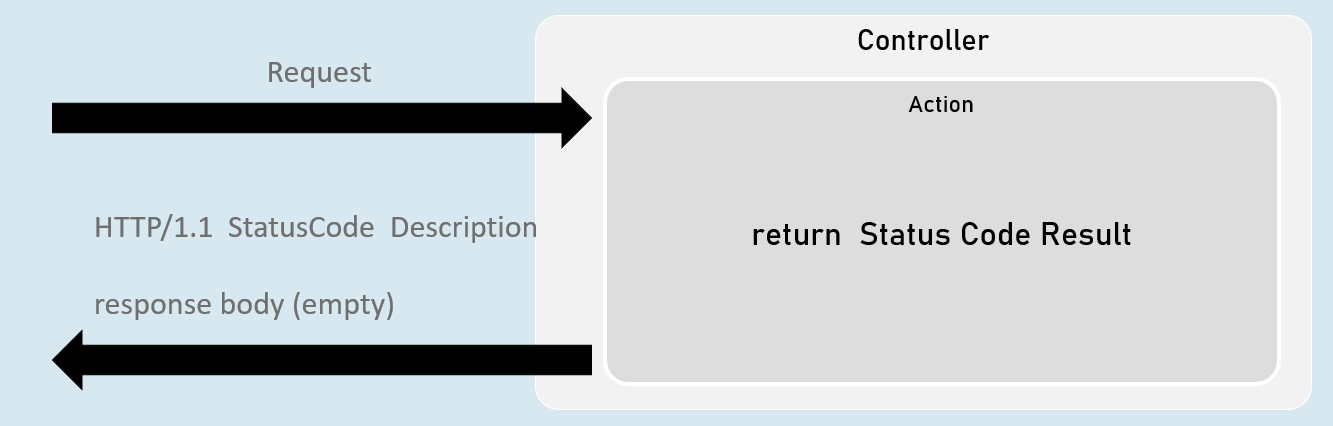
Bằng cách đề cập đến kiểu trả về là IActionResult, bạn có thể trả về một trong các kiểu con của IActionResult



Kết quả mã trạng thái

Kết quả mã trạng thái gửi một phản hồi trống với mã trạng thái được chỉ định.

Ví dụ: 200, 400, 401, 404, 500, v.v.



**StatusCodeResult**

trả về StatusCodeResult(status\_code) mới;

**Kết quả trái phép**

trả về UnauthorizedResult() mới;

Kết quả yêu cầu xấu

trả về BadRequestResult() mới;

**Kết quả không tìm thấy**

trả về notFoundResult() mới;

**StatusCodeResult**

* Thể hiện phản hồi với mã trạng thái được chỉ định.
* Được sử dụng khi bạn muốn gửi mã trạng thái HTTP cụ thể dưới dạng phản hồi.

trả về StatusCodeResult(status\_code) mới;

hoặc

trả về StatusCode(status\_code);

**Kết quả không được ủy quyền**

* Thể hiện phản hồi bằng mã trạng thái HTTP '401 Unauthorized'.
* Được sử dụng khi người dùng trái phép (không đăng nhập).

trả về UnauthorizedResult() mới;

hoặc

trả lại Unauthorized();

**Kết quả yêu cầu xấu**

* Thể hiện phản hồi bằng mã trạng thái HTTP '400 Yêu cầu không hợp lệ'.
* Được sử dụng khi các giá trị yêu cầu không hợp lệ (lỗi xác thực).

trả về BadRequestResult() mới;

hoặc

trả về BadRequest();

**Kết quả không tìm thấy**

* Thể hiện phản hồi với mã trạng thái HTTP '404 Not Found'.
* Được sử dụng khi thông tin được yêu cầu không có sẵn tại máy chủ.

trả về notFoundResult() mới;

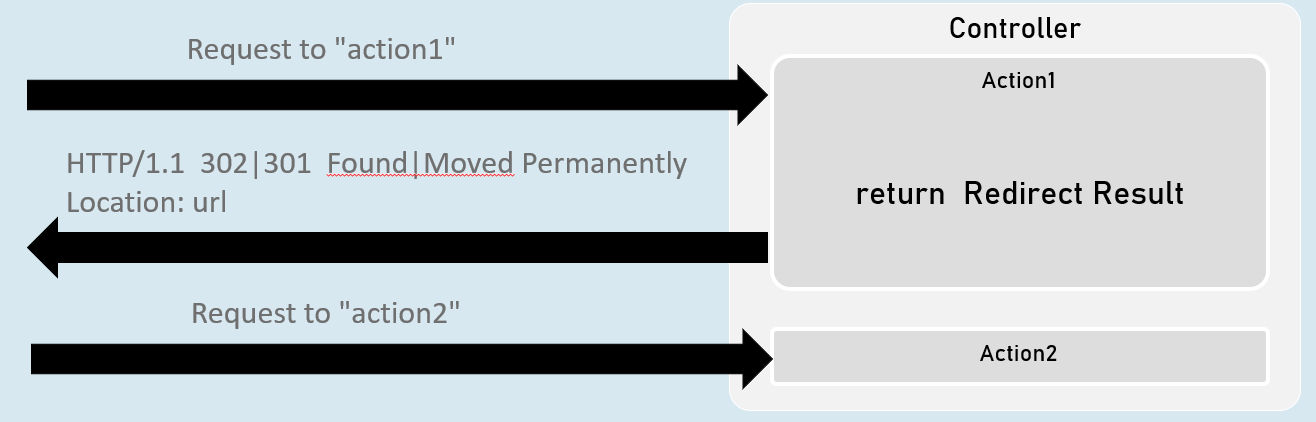
hoặc

trả về NotFound();

Kết quả chuyển hướng

Kết quả chuyển hướng gửi phản hồi HTTP 302 hoặc 301 đến trình duyệt, để chuyển hướng đến một hành động hoặc url cụ thể.

Ví dụ: chuyển hướng từ 'action1' sang 'action2'.



**Chuyển hướng đến kết quả hành động**

return new RedirectToActionResult("action", "controller", new { route\_values }, permanent);

**LocalRedirectResult**

trả về localRedirectResult("local\_url", vĩnh viễn);

**Kết quả chuyển hướng**

trả về RedirectResult("url", vĩnh viễn);

Chuyển hướng đến kết quả hành động

Thể hiện phản hồi để chuyển hướng từ phương thức hành động hiện tại sang phương thức hành động khác, dựa trên tên hành động và tên bộ điều khiển.

**302 - Đã tìm thấy**

trả về new RedirectToActionResult("action", "controller", new { route\_values });

hoặc

return RedirectToAction("action", "controller", new { route\_values });

**301 - Di chuyển vĩnh viễn**

trả lại new RedirectToActionResult("action", "controller", new { route\_values }, true);

hoặc

return RedirectToActionPermanent("action", "controller", new { route\_values });

LocalRedirectResult

•Thể hiện phản hồi để chuyển hướng từ phương thức hành động hiện tại sang phương thức hành động khác, dựa trên url được chỉ định.

**302 - Đã tìm thấy**

trả về LocalRedirectResult("url");

hoặc

trả về LocalRedirect("url);

**301 - Di chuyển vĩnh viễn**

return new LocalRedirectResult("url", true);

hoặc

trả về LocalRedirectPermanent("url");

Kết quả chuyển hướng

Thể hiện phản hồi để chuyển hướng từ phương thức hành động hiện tại đến bất kỳ url nào khác (trong cùng một ứng dụng web hoặc ứng dụng web khác).

**302 - Đã tìm thấy**

trả về RedirectResult("url");

hoặc

return Redirect("url);

**301 - Di chuyển vĩnh viễn**

trả về RedirectResult("url", true);

hoặc

trả về RedirectPermanent("url");

# **5. Liên kết và xác thực mô hình [MVC vs API]**

Liên kết mô hình

Liên kết mô hình là một tính năng mạnh mẽ trong ASP.NET Core MVC tự động hóa quá trình trích xuất dữ liệu từ các phần khác nhau của yêu cầu HTTP (dữ liệu biểu mẫu, giá trị tuyến đường, chuỗi truy vấn) và chuyển đổi nó thành các đối tượng C# được gõ mạnh mà bạn có thể sử dụng trực tiếp trong các phương thức hành động của mình.

**Khi liên kết mô hình thực thi**

Liên kết mô hình diễn ra **sau khi** định tuyến đã xác định phương thức hành động nào sẽ gọi. Hệ thống liên kết mô hình kiểm tra các tham số của phương thức hành động đã chọn và cố gắng điền chúng với các giá trị từ yêu cầu đến.

**Thứ tự liên kết mô hình**

ASP.NET Liên kết mô hình cốt lõi tuân theo một thứ tự cụ thể khi tìm kiếm nguồn dữ liệu:

1. **Dữ liệu biểu mẫu (yêu cầu POST):** Giá trị được gửi thông qua biểu mẫu HTML.
2. **Dữ liệu tuyến đường:** Các giá trị được trích xuất từ mẫu tuyến URL (ví dụ: /products/{id}).
3. **Chuỗi truy vấn:** Các giá trị được thêm vào URL sau dấu chấm hỏi (?).

**Các phần của liên kết mô hình**

* **Dữ liệu biểu mẫu:** Thường được sử dụng để gửi dữ liệu từ biểu mẫu HTML bằng cách sử dụng các yêu cầu POST.
* **Dữ liệu tuyến đường:** Giá trị được thu thập từ các phân đoạn URL được xác định trong mẫu tuyến đường của bạn.
* **Chuỗi truy vấn:** Các tham số được truyền trong URL sau dấu chấm hỏi (?).

**Chuỗi truy vấn chi tiết**

* **Mục đích:** Để chuyển các tham số đến ứng dụng của bạn thông qua URL.
* **Cú pháp:** ?key1=value1&key2=value2 (nhiều cặp khóa-giá trị được phân tách bằng dấu hiệu và
* **Cách sử dụng:** Hữu ích cho việc lọc, sắp xếp và phân trang.
* **Ví dụ:** /products?category=electronics&sort=price\_desc

**Các phương pháp hay nhất (chuỗi truy vấn):**

* **Giới hạn dữ liệu nhạy cảm:** Tránh chuyển thông tin nhạy cảm như mật khẩu hoặc chi tiết thẻ tín dụng trong chuỗi truy vấn.
* **Vệ sinh đầu vào:** Luôn vệ sinh và xác thực các giá trị chuỗi truy vấn để ngăn chặn các lỗ hổng bảo mật.
* **Giữ cho nó đơn giản:** Sử dụng tên tham số rõ ràng và có ý nghĩa. Tránh các chuỗi truy vấn quá dài.
* **Mã hóa:** Mã hóa đúng các ký tự đặc biệt trong các giá trị chuỗi truy vấn.

**Những điều cần tránh (chuỗi truy vấn):**

* **Dữ liệu nhạy cảm:** Không bao giờ chuyển dữ liệu nhạy cảm như mật khẩu hoặc mã thông báo xác thực trong chuỗi truy vấn.
* **Đối tượng phức tạp:** Tránh chuyển các đối tượng phức tạp dưới dạng chuỗi truy vấn do giới hạn độ dài URL.
* **Lạm dụng:** Không làm quá tải URL với quá nhiều thông số truy vấn.

**Dữ liệu tuyến đường chi tiết**

* **Mục đích:** Nắm bắt các giá trị động từ URL dựa trên mẫu tuyến đường.
* **Cú pháp:** /products/{id}, trong đó id là một tham số tuyến đường.
* **Cách sử dụng:** Cần thiết cho các API RESTful và để tạo URL sạch, dễ đọc.
* **Ví dụ:** /products/12345 (12345 sẽ là giá trị của thông số id).

**Các phương pháp hay nhất (Dữ liệu tuyến đường):**

* **Chọn Xóa tên:** Sử dụng tên mô tả cho các tham số tuyến đường.
* **Ràng buộc:** Áp dụng các ràng buộc tuyến đường (ví dụ: int, guid) để đảm bảo các kiểu dữ liệu hợp lệ.
* **Ràng buộc tùy chỉnh:** Tạo các ràng buộc tùy chỉnh để xác thực phức tạp hơn.

Mã

1. HomeController.cs
2. [Route("hiệu sách/{bookid?}/{isloggedin?}")] Định tuyến với các thông số tùy chọn
3. Public IActionResult Index(int? bookid, bool? isloggedin)
4. {
5. // ... logic xác thực (giống như ví dụ trước)...
6. return Content($"Book id: {bookid}, isloggedin: {isloggedin}", "text/plain");
7. }

Trong phương pháp hành động này:

* Các tham số bookID và isloggedin được tự động liên kết từ chuỗi truy vấn và dữ liệu tuyến đường.

Mã

1. StoreController.cs
2. [Route("store/books/{id}")] // Tuyến đường có tham số bắt buộc
3. IActionSổ kết quả công khai()
4. {
5. int id = Convert.ToInt32(Request.RouteValues["id"]);
6. trả lại Content($"<h1>Book Store {id}</h1>", "text/html");
7. }

Trong phương pháp hành động này:

* id là một tham số bắt buộc.
* id được ràng buộc từ dữ liệu tuyến đường (Request.RouteValues

[FromQuery] và [FromRoute]

Trong khi liên kết mô hình của ASP.NET Core tự động cố gắng khớp các tham số phương thức hành động với các phần khác nhau của yêu cầu (dữ liệu biểu mẫu, giá trị tuyến đường, chuỗi truy vấn), bạn có thể sử dụng các thuộc tính [FromQuery] và [FromRoute] để cho trình kết dính mô hình biết rõ ràng nơi tìm kiếm các giá trị cụ thể.

**[TừTruy vấn]**

* **Mục đích:** Hướng dẫn trình kết dính mô hình trích xuất giá trị tham số từ chuỗi truy vấn.
* **Cách sử dụng:** Áp dụng thuộc tính này cho các tham số phương thức hành động mà bạn mong đợi nhận được giá trị từ phần chuỗi truy vấn của URL (phần sau "?").
* **Ví dụ:**

1. public IActionResult Index([FromQuery] int page) { ... }

Trong ví dụ này, thông số trang sẽ được liên kết với giá trị của thông số truy vấn trang trong URL (ví dụ: /products?page=3).

**[TừTuyến đường]**

* **Mục đích:** Hướng dẫn chất kết dính mô hình trích xuất giá trị tham số từ dữ liệu tuyến đường.
* **Cách sử dụng:** Áp dụng thuộc tính này cho các tham số phương thức hành động mà bạn mong đợi nhận được giá trị từ mẫu định tuyến của URL.
* **Ví dụ:**

1. [Route("products/{id}")]
2. public IActionResult Details([FromRoute] int id) { ... }

Trong ví dụ này, thông số id sẽ được liên kết với giá trị của phân đoạn id trong URL (ví dụ: /products/123).

Mã

1. HomeController.cs
2. [Route("hiệu sách/{bookid?}/{isloggedin?}")] Định tuyến với các thông số tùy chọn
3. public IActionResult Index([FromQuery] int? bookid, [FromRoute] bool? isloggedin)
4. {
5. // ... (phần còn lại của logic xác thực và phản hồi)...
6. }

Trong mã này:

* **bookid (FromQuery):** Trình kết dính mô hình sẽ cố gắng truy xuất giá trị bookid độc quyền từ chuỗi truy vấn. Nếu nó không có trong chuỗi truy vấn, nó sẽ được đặt thành null do kiểu có thể rỗng (int?).
* **isloggedin (FromRoute):** Trình kết dính mô hình sẽ tìm kiếm cụ thể giá trị isloggedin trong dữ liệu tuyến đường. Nếu tham số không có trong tuyến đường, nó sẽ mặc định là null do kiểu boolean có thể rỗng (bool?).

**Ràng buộc kết hợp:**

Bằng cách sử dụng cả [FromQuery] và [FromRoute] trên các tham số khác nhau trong cùng một phương thức hành động, bạn có thể liên kết đồng thời các giá trị từ cả chuỗi truy vấn và dữ liệu định tuyến một cách hiệu quả.

**Ghi chú**

* **Liên kết rõ ràng:** Sử dụng [FromQuery] và [FromRoute] để kiểm soát rõ ràng nơi liên kết mô hình lấy giá trị cho các tham số phương thức hành động của bạn.
* **Tính linh hoạt:** Bạn có thể kết hợp cả hai thuộc tính trong cùng một hành động để liên kết từ nhiều nguồn.
* **Hành vi mặc định:** Ngay cả khi không có các thuộc tính này, liên kết mô hình của ASP.NET Core sẽ cố gắng xác định nguồn liên kết một cách thông minh. Tuy nhiên, việc sử dụng các thuộc tính này sẽ làm cho mã của bạn rõ ràng hơn và ít bị ảnh hưởng bởi hành vi không mong muốn hơn.
* **Chuyển đổi loại:** Trình kết dính mô hình tự động cố gắng chuyển đổi giá trị sang kiểu dữ liệu thích hợp cho các tham số phương thức hành động của bạn.

Lớp mô hình

Trong ASP.NET Core MVC, các lớp mô hình là nền tảng để biểu diễn dữ liệu mà ứng dụng của bạn hoạt động. Chúng thường phản ánh cấu trúc dữ liệu của bạn, cho dù nó đến từ cơ sở dữ liệu, API hay các nguồn khác.

* **Mục đích:**
  + **Cấu trúc:** Cung cấp cấu trúc được xác định rõ ràng cho dữ liệu của bạn, bao gồm các thuộc tính tương ứng với các trường hoặc thuộc tính của thực thể dữ liệu của bạn.
  + **Xác thực:** Thực thi các quy tắc xác thực dữ liệu bằng cách sử dụng các thuộc tính như [Bắt buộc], [StringLength] và [Phạm vi].
  + **Tổ chức:** Giữ cho logic dữ liệu của ứng dụng của bạn có tổ chức và có thể bảo trì.
* **Lớp mô hình ví dụ:**

1. Book.cs (Mô hình)
2. không gian tên IActionResultExample.Models
3. {
4. Lớp học công cộng Sách
5. {
6. int công khai? BookId { get; set; }
7. chuỗi công khai? Tác giả { get; set; }
9. public override string ToString() // Để dễ dàng hiển thị trong ví dụ này
10. {
11. return $"Đối tượng sách - Id sách: {BookId}, Tác giả: {Tác giả}";
12. }
13. }
14. }

Liên kết mô hình với các lớp mô hình

Liên kết mô hình với các lớp mô hình giúp đơn giản hóa quá trình điền dữ liệu từ các yêu cầu HTTP đến. Thay vì trích xuất các giá trị theo cách thủ công từ chuỗi truy vấn, dữ liệu tuyến đường hoặc dữ liệu biểu mẫu, bạn có thể trực tiếp sử dụng lớp mô hình làm tham số trong phương thức hành động của mình.

* **Cách thức hoạt động:**
  1. **Tham số hành động:** Khai báo tham số phương thức hành động của loại lớp mô hình của bạn.
  2. **Liên kết mô hình:** Trình liên kết mô hình tự động ánh xạ dữ liệu yêu cầu đến với các thuộc tính của lớp mô hình dựa trên tên của chúng.
  3. **Sử dụng thuộc tính:** Bạn có thể sử dụng các thuộc tính như [FromQuery], [FromRoute] và [FromBody] để chỉ định vị trí của trình kết dính mô hình sẽ tìm kiếm dữ liệu cho từng thuộc tính.

Mã

1. HomeController.cs
2. [Route("hiệu sách/{bookid?}/{isloggedin?}")]
3. Url: /bookstore/1/false?bookid=20&isloggedin=true&author=harsha
4. public IActionResult Index([FromQuery] int? bookid, [FromRoute] bool? isloggedin, Book book)
5. {
6. // ... logic xác thực và phản hồi ...
7. return Content($"Book id: {bookid}, Book: {book}", "text/plain");
8. }

Trong mã này:

1. **Model Class Parameter:** Phương thức hành động Index có một tham số tên là book.
2. **Thuộc tính [FromQuery]:** Thuộc tính BookId của lớp Book có thuộc tính [FromQuery], cho biết rằng giá trị của nó phải được truy xuất từ chuỗi truy vấn.
3. **Tự động ràng buộc:** Khi một yêu cầu như /bookstore/1/false?bookid=20&isloggedin=true&author=harsha xuất hiện:
   * bookid (int?) sẽ là 20 (từ chuỗi truy vấn, do [FromQuery]).
   * isloggedin (bool?) sẽ là true (từ dữ liệu tuyến đường, do [FromRoute]).
   * sách. Tác giả (chuỗi?) sẽ là "harsha" (từ chuỗi truy vấn, vì không có thuộc tính nào được chỉ định cho thuộc tính Tác giả nên nó mặc định tìm kiếm trong chuỗi truy vấn).

**Ghi chú**

* **Mã đơn giản:** Liên kết mô hình giảm mã nguyên mẫu để trích xuất dữ liệu từ các yêu cầu.
* **Nhập mạnh:** Bạn làm việc với các đối tượng mô hình được nhập mạnh trong hành động của mình.
* **Ý định rõ ràng:** Các thuộc tính như [FromQuery], [FromRoute] và [FromBody] làm cho mã của bạn rõ ràng hơn.
* **Chuyển đổi tự động:** Trình kết dính mô hình cố gắng chuyển đổi dữ liệu yêu cầu để phù hợp với các loại thuộc tính mô hình của bạn.
* **Loại phức tạp:** Bạn có thể liên kết các đối tượng phức tạp từ dữ liệu JSON hoặc XML trong nội dung yêu cầu (sử dụng [FromBody]).
* **Xác thực:** Tận dụng các thuộc tính xác thực mô hình để đảm bảo tính toàn vẹn của dữ liệu.

Mã hóa URL

Mã hóa URL (hoặc mã hóa phần trăm) là một cơ chế để mã hóa các ký tự đặc biệt trong các URL không được phép ở dạng thô. Mã hóa này rất cần thiết để đảm bảo rằng các URL được truyền chính xác và máy chủ diễn giải chúng một cách chính xác.

* **Ký tự đặc biệt:** Các ký tự như dấu cách, ký hiệu và (&), dấu chấm hỏi (?) và các ký tự không phải ASCII cần được mã hóa.
* **Định dạng mã hóa:** Các ký tự đặc biệt được thay thế bằng dấu phần trăm (%) theo sau là hai chữ số thập lục phân đại diện cho mã ASCII của chúng.
* **Ví dụ:** Khoảng trắng được mã hóa là %20 và dấu hiệu và được mã hóa là %26.

**Các loại nội dung để gửi biểu mẫu**

1. **application/x-www-form-urlđược mã hóa:**
   * **Mục đích:** Mã hóa mặc định cho các biểu mẫu HTML. Nó mã hóa dữ liệu biểu mẫu dưới dạng các cặp khóa-giá trị được phân tách bằng dấu hiệu và (&) và có dấu bằng nhau (=) giữa các khóa và giá trị. Khoảng trắng được chuyển đổi thành dấu cộng (+).
   * **Cách sử dụng:** Thích hợp cho các biểu mẫu đơn giản với dữ liệu văn bản.
   * **Hạn chế:** Không hiệu quả đối với lượng lớn dữ liệu hoặc dữ liệu nhị phân (như tải lên tệp).
2. **multipart/form-data:**
   * **Mục đích:** Được thiết kế để gửi biểu mẫu có tệp hoặc lượng lớn dữ liệu. Mỗi trường biểu mẫu được gửi dưới dạng một phần riêng biệt, với loại nội dung và tiêu đề riêng.
   * **Cách sử dụng:** Cần thiết cho việc tải lên tệp.
   * **Lợi ích:** Xử lý dữ liệu nhị phân một cách hiệu quả và có thể hỗ trợ tải trọng lớn hơn.
3. **dữ liệu biểu mẫu:**
   * **Mục đích:** Một định dạng mới hơn và linh hoạt hơn để gửi biểu mẫu có thể xử lý cả dữ liệu đơn giản và phức tạp, bao gồm cả tệp.
   * **Cách sử dụng:** Cung cấp một giải pháp thay thế hiện đại hơn cho dữ liệu đa phần/biểu mẫu.
   * **Lợi ích:** Tương tự như dữ liệu đa phần/biểu mẫu nhưng có cấu trúc hợp lý hơn.

**Ghi chú**

* **Mã hóa URL:** Cần thiết để đảm bảo rằng các URL được định hình và diễn giải đúng cách.
* **Gửi biểu mẫu:**
  + application/x-www-form-urlencoded: Mặc định cho các biểu mẫu đơn giản.
  + multipart/form-data hoặc form-data: Bắt buộc để tải lên tệp và tải trọng lớn hơn.
* **Liên kết mô hình:** ASP.NET Core MVC tự động xử lý dữ liệu biểu mẫu liên kết (được gửi qua POST) với các lớp mô hình của bạn dựa trên tiêu đề Content-Type.

Xác thực mô hình

Xác thực mô hình là quá trình xác minh rằng dữ liệu được gửi đến ứng dụng ASP.NET Core MVC của bạn đáp ứng các tiêu chí đã xác định của bạn. Điều này ngăn dữ liệu không hợp lệ hoặc độc hại xâm nhập vào hệ thống của bạn và giúp duy trì tính toàn vẹn của dữ liệu ứng dụng.

**Tại sao xác thực mô hình lại quan trọng**

* **Bảo mật:** Bảo vệ chống lại các cuộc tấn công phổ biến như SQL injection, cross-site scripting (XSS) và overpost.
* **Tính toàn vẹn của dữ liệu:** Đảm bảo rằng dữ liệu được lưu trữ trong cơ sở dữ liệu của bạn hoặc được sử dụng trong logic ứng dụng của bạn là hợp lệ.
* **Trải nghiệm người dùng:** Cung cấp phản hồi ngay lập tức cho người dùng, hướng dẫn họ sửa lỗi nhập liệu.

**Các phương pháp hay nhất**

1. **Xác thực cả hai phía:** Xác thực dữ liệu cả ở phía máy khách (sử dụng JavaScript) để có phản hồi ngay lập tức và ở phía máy chủ để bảo mật (vì xác thực phía máy khách có thể bị bỏ qua).
2. **Sử dụng chú thích dữ liệu:** Tận dụng các thuộc tính chú thích dữ liệu tích hợp được cung cấp bởi không gian tên System.ComponentModel.DataAnnotations để thể hiện các quy tắc xác thực một cách ngắn gọn.
3. **Thuộc tính xác thực tùy chỉnh:** Tạo thuộc tính xác thực tùy chỉnh cho các quy tắc phức tạp hơn hoặc dành riêng cho miền.
4. **Trạng thái mô hình:** Luôn kiểm tra thuộc tính ModelState.IsValid trong các hành động của bộ điều khiển trước khi xử lý dữ liệu. Nếu không hợp lệ, hãy trả về phản hồi lỗi thích hợp.
5. **Hiển thị thông báo lỗi:** Hiển thị rõ ràng thông báo lỗi cho người dùng, cho biết trường nào không hợp lệ và tại sao.

**Chú thích dữ liệu cần thiết**

Dưới đây là một số thuộc tính chú thích dữ liệu được sử dụng phổ biến nhất:

* **[Bắt buộc]:** Trường không được rỗng hoặc trống.
* **[StringLength]:** Giới hạn độ dài tối đa hoặc tối thiểu của chuỗi.
* **[Phạm vi]:** Chỉ định một phạm vi số trong đó giá trị phải nằm trong đó.
* **[RegularExpression]:** Xác thực giá trị dựa trên mẫu biểu thức chính quy.
* **[EmailAddress]:** Xác minh rằng giá trị là định dạng địa chỉ email hợp lệ.
* **[So sánh]:** So sánh giá trị của tài sản này với tài sản khác (ví dụ: xác nhận mật khẩu).
* **[Điện thoại]:** Xác thực định dạng số điện thoại.
* **[Url]:** Xác thực định dạng URL.

Trạng thái mô hình trong bộ điều khiển

Đối tượng ModelState trong bộ điều khiển của bạn rất quan trọng để xác thực. Nó theo dõi trạng thái xác thực của mô hình của bạn sau khi trình kết dính mô hình đã cố gắng điền nó từ yêu cầu.

* **ModelState.IsValid:** Một thuộc tính boolean cho biết liệu tất cả các quy tắc xác thực đã vượt qua (true) hay có bất kỳ lỗi nào (false) hay không.
* **ModelState.AddModelError:** Thêm lỗi mô hình theo cách thủ công cho một thuộc tính cụ thể.
* **Thông báo lỗi:** Truy xuất thông báo lỗi được liên kết với các thuộc tính cụ thể.

Mã

1. Person.cs (Mô hình)
2. người lớp công cộng
3. {
4. // ... Các thuộc tính khác
6. [Bắt buộc (ErrorMessage = "{0} không thể để trống")]
7. [So sánh ("Mật khẩu", ErrorMessage = "{0} và {1} không khớp")]
8. [Hiển thị (Tên = "Nhập lại mật khẩu")]
9. chuỗi công khai? ConfirmPassword { get; set; }
11. // ... Các thuộc tính khác
12. }
14. (Trong hành động bộ điều khiển của bạn)
15. public IActionResult Create(Người người)
16. {
17. nếu (! ModelState.IsValid)
18. {
19. trả lại View(người); Quay lại chế độ xem có lỗi xác thực
20. }
22. Mô hình hợp lệ, tiến hành lưu dữ liệu
23. }

Trong mã này:

1. **Chú thích dữ liệu:** Mô hình Người sử dụng chú thích dữ liệu để thực thi các quy tắc xác thực.
2. **Kiểm tra trạng thái mô hình:** Hành động của bộ điều khiển kiểm tra ModelState.IsValid. Nếu sai, chế độ xem ban đầu được hiển thị lại với đối tượng mô hình có chứa lỗi xác thực, cho phép người dùng sửa chúng.
3. **Hiển thị lỗi:** Chế độ xem thường sử dụng các phương thức trợ giúp @Html.ValidationSummary() và @Html.ValidationMessageFor() để hiển thị thông báo lỗi cho người dùng.

Xác thực tùy chỉnh với ValidationAttribute

Mặc dù các thuộc tính xác thực tích hợp của ASP.NET Core bao gồm nhiều tình huống, nhưng chắc chắn bạn sẽ gặp phải các quy tắc xác thực cụ thể cho logic nghiệp vụ của ứng dụng của mình. Các thuộc tính xác thực tùy chỉnh, bắt nguồn từ lớp ValidationAttribute, cho phép bạn tạo các xác thực phù hợp này.

**Các bước chính**

1. **Kế thừa từ ValidationAttribute:** Tạo một lớp kế thừa từ ValidationAttribute.
2. **Ghi đè IsValid:** Cốt lõi của logic xác thực tùy chỉnh nằm ở phương thức IsValid. Phương thức này nhận giá trị cần xác thực và đối tượng ValidationContext (chứa thông tin bổ sung về mô hình).
3. **Trả về ValidationResult:**
   * Nếu giá trị hợp lệ, hãy trả về ValidationResult.Success.
   * Nếu giá trị không hợp lệ, hãy trả về một đối tượng ValidationResult mới với thông báo lỗi tùy chỉnh của bạn.

Mã

1. lớp công khai DateRangeValidatorAttribute : ValidationAttribute
2. {
3. chuỗi công khai OtherPropertyName { get; set; }
5. Constructor
6. công khai DateRangeValidatorAttribute(chuỗi otherPropertyName)
7. {
8. OtherPropertyName = tên tài sản khác;
9. }
11. được bảo vệ ghi đè ValidationResult? IsValid(giá trị đối tượng, ValidationContext validationContext)
12. {
13. if (giá trị != null)
14. {
15. Nhận "to\_date"
16. DateTime toDate = Convert.ToDateTime(giá trị);
18. Nhận "from\_date"
19. var otherProperty = validationContext.ObjectType.GetProperty(OtherPropertyName);
21. nếu (otherProperty!= null)
22. {
23. DateTime fromDate = Convert.ToDateTime(otherProperty.GetValue(validationContext.ObjectInstance));
25. if (từNgày > đến ngày)
26. {
27. trả lại new ValidationResult(ErrorMessage, new string[] { OtherPropertyName, validationContext.MemberName }); Cho biết các thuộc tính cụ thể liên quan đến lỗi
28. }
29. khác
30. {
31. trả về ValidationResult.Success;
32. }
33. }
34. trả về null; Trả về null nếu otherProperty là null
35. }
36. trả về null; Trả về null nếu giá trị là null
37. }
38. }

* **Mục đích:** Đảm bảo rằng một ngày (ví dụ: ToDate) không sớm hơn một ngày khác (FromDate).
* **OtherPropertyName:** Chỉ định tên của thuộc tính để so sánh (trong trường hợp này là FromDate).
* **IsValid:**
  + Nó truy xuất các giá trị của cả hai thuộc tính bằng cách sử dụng phản xạ.
  + Nó so sánh các ngày và trả về thông báo lỗi nếu toDate sớm hơn fromDate.
  + Thông báo lỗi bao gồm tên của cả hai thuộc tính, cung cấp phản hồi rõ ràng cho người dùng.

Mã

1. lớp công khai MinimumYearValidatorAttribute : ValidationAttribute
2. {
3. public int MinimumYear { get; set; } = 2000;
4. chuỗi công khai DefaultErrorMessage { get; set; } = "Năm không được nhỏ hơn {0}";
6. // ... (người xây dựng)...
8. được bảo vệ ghi đè ValidationResult? IsValid(giá trị đối tượng, ValidationContext validationContext)
9. {
10. if (giá trị != null)
11. {
12. DateTime date = (DateTime)giá trị;
13. nếu (ngày. Năm >= Năm tối thiểu)
14. {
15. trả về ValidationResult.Success;
16. }
17. khác
18. {
19. trả về ValidationResult(string. Định dạng (MessageError ?? DefaultErrorMessage, MinimumYear)); Sử dụng thông báo lỗi tùy chỉnh hoặc mặc định
20. }
21. }
23. trả về null;
24. }
25. }

* **Mục đích:** Đảm bảo rằng ngày (ví dụ: Ngày sinh) không sớm hơn một năm cụ thể.
* **MinimumYear:** Đặt năm tối thiểu cho phép (mặc định là 2000).
* **DefaultErrorMessage:** Cung cấp thông báo lỗi mặc định nếu không cung cấp thông báo tùy chỉnh.
* **IsValid:**
  + Nó kiểm tra xem năm của ngày nhất định có lớn hơn hoặc bằng năm tối thiểu hay không.

IValidatableObject

Mặc dù chú thích dữ liệu ([Bắt buộc], [StringLength], v.v.) cung cấp một cách ngắn gọn để xác định các quy tắc xác thực trên các thuộc tính mô hình riêng lẻ, giao diện IValidatableObject cho phép bạn thực hiện logic xác thực cấp mô hình phức tạp hơn bao gồm nhiều thuộc tính hoặc phụ thuộc vào trạng thái của toàn bộ mô hình.

**Ghi chú**

* **Giao diện:** IValidatableObject là một giao diện có một phương thức duy nhất: Validate(ValidationContext context).
* **Phương pháp xác thực:** Phương thức này được gọi bởi trình kết dính mô hình sau khi xác thực cấp thuộc tính riêng lẻ (chú thích dữ liệu) đã được kiểm tra.
* **Yielding Errors:** Trong phương thức Validate, bạn có thể tạo ra các đối tượng ValidationResult cho bất kỳ lỗi nào bạn tìm thấy. Điều này cho phép bạn báo cáo nhiều lỗi cho toàn bộ mô hình cùng một lúc.
* **Tích hợp trạng thái mô hình:** Các lỗi bạn tạo ra sẽ tự động được thêm vào đối tượng ModelState, giúp chúng có sẵn để hiển thị lỗi trong chế độ xem của bạn.

**Khi nào nên sử dụng IValidatableObject**

* **Xác thực chéo thuộc tính:** Khi logic xác thực phụ thuộc vào giá trị của nhiều thuộc tính (ví dụ: "Ngày bắt đầu" phải trước "Ngày kết thúc").
* **Quy tắc công việc phức tạp:** Khi các quy tắc xác thực của bạn liên quan đến logic phức tạp hoặc tra cứu cơ sở dữ liệu.
* **Lỗi có thể tùy chỉnh:** Khi bạn muốn kiểm soát nhiều hơn các thông báo lỗi hiển thị cho người dùng.

Mã

1. Person.cs (Mô hình)
2. public class Người : IValidatableObject
3. {
4. // ... (thuộc tính có chú thích dữ liệu)...
6. Ngày giờ công khai? DateOfBirth { get; set; }
7. int công khai? Age { get; set; } // Thuộc tính mới
9. // ... (các tính chất và phương pháp khác)...
11. public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
12. {
13. if (DateOfBirth.HasValue == false && Age.HasValue == false)
14. {
15. yield return new ValidationResult("Ngày sinh hoặc Tuổi phải được cung cấp", new[] { nameof(Tuổi) }); Đưa ra lỗi
16. }
17. }
18. }

Trong ví dụ này:

1. **Triển khai IValidatableObject:** Lớp Person hiện triển khai IValidatableObject.
2. **Phương pháp xác thực:**
   * Nó kiểm tra xem DateOfBirth hoặc Age có được cung cấp hay không. Nếu cả hai đều không có, nó sẽ tạo ra một ValidationResult cho biết rằng ít nhất một trong các thuộc tính này phải được cung cấp.
   * Lưu ý cách thông báo lỗi được liên kết cụ thể với thuộc tính Age bằng cách sử dụng new[] { nameof(Age) }. Điều này giúp nhắm mục tiêu thông báo lỗi đến đúng trường trong chế độ xem của bạn.
3. **Cập nhật trạng thái mô hình:** Khi bạn sử dụng mô hình này trong bộ điều khiển của mình, các lỗi xác thực từ phương thức Validate sẽ tự động được thêm vào ModelState và bạn có thể kiểm tra ModelState.IsValid để xác định xem mô hình có hợp lệ hay không.

**Ghi chú**

* **Xác thực cấp mô hình:** IValidatableObject lý tưởng cho logic xác thực vượt ra ngoài các thuộc tính riêng lẻ.
* **Lỗi mang lại:** Sử dụng câu lệnh yield return để trả về nhiều lỗi xác thực từ phương thức Validate.
* **Nhắm mục tiêu lỗi:** Liên kết thông báo lỗi với các thuộc tính cụ thể để có phản hồi rõ ràng của người dùng.
* **Tích hợp với Chú thích dữ liệu:** IValidatableObject hoạt động cùng với chú thích dữ liệu, cung cấp một cách tiếp cận xác thực toàn diện.

Thuộc tính [Bind] và [BindNever]

Liên kết mô hình rất mạnh mẽ, nhưng đôi khi bạn muốn kiểm soát chi tiết hơn đối với các thuộc tính được điền từ dữ liệu yêu cầu đến. Đây là nơi [Bind] và [BindNever] xuất hiện.

**[Ràng buộc] Thuộc tính**

* **Mục đích:** Bao gồm rõ ràng các thuộc tính cụ thể để liên kết mô hình.
* **Cách sử dụng:** Áp dụng thuộc tính này cho tham số phương thức hành động của bạn (ví dụ: lớp mô hình) và cung cấp danh sách tên thuộc tính làm đối số.
* **Ví dụ:**

1. [Bài đăng]
2. public IActionResult Create([Bind("Title", "Description")] Sản phẩm sản phẩm)
3. {
4. Chỉ các thuộc tính Tiêu đề và Mô tả mới được ràng buộc từ yêu cầu.
5. }

Trong ví dụ này, ngay cả khi yêu cầu đến chứa dữ liệu cho các thuộc tính khác của lớp Product (như Price hoặc Category), chúng sẽ bị bỏ qua trong quá trình liên kết mô hình.

**[Ràng buộc không bao giờ] Thuộc tính**

* **Mục đích:** Loại trừ các thuộc tính cụ thể khỏi liên kết mô hình.
* **Cách sử dụng:** Áp dụng thuộc tính này trực tiếp cho các thuộc tính mô hình mà bạn không bao giờ muốn bị ràng buộc khỏi yêu cầu.
* **Ví dụ:**

1. sản phẩm lớp công cộng
2. {
3. // ... Các thuộc tính khác
5. [Ràng buộc không bao giờ]
6. public DateTime CreatedAt { get; set; } Không bao giờ liên kết từ yêu cầu
7. }

Trong ví dụ này, thuộc tính CreatedAt sẽ luôn giữ nguyên giá trị mặc định của nó, bất kể yêu cầu đến có chứa dữ liệu cho nó hay không.

Mã

1. Person.cs (Mô hình)
2. public class Người : IValidatableObject
3. {
4. // ... (các thuộc tính khác)
6. [BindNever] // Thuộc tính này sẽ không bị ràng buộc trong quá trình liên kết mô hình
7. Ngày giờ công khai? DateOfBirth { get; set; }
9. // ... (các tính chất và phương pháp khác)...
10. }
12. HomeController.cs
13. [Lộ trình ("đăng ký")]
14. IActionChỉ số kết quả công khai(Người người)
15. {
16. // ... (logic xác nhận và phản hồi)...
17. }

Trong mã này:

1. **DateOfBirth (BindNever):** Thuộc tính [BindNever] trên thuộc tính DateOfBirth yêu cầu trình kết nối mô hình bỏ qua hoàn toàn bất kỳ dữ liệu nào cho thuộc tính này đến từ yêu cầu. Ngay cả khi yêu cầu đến chứa giá trị cho DateOfBirth, nó sẽ không được gán cho thuộc tính model.

**Ghi chú**

* **Bảo mật:** [BindNever] là một công cụ có giá trị để ngăn chặn các cuộc tấn công đăng tải quá mức, trong đó kẻ tấn công cố gắng gửi dữ liệu cho các thuộc tính không thể sửa đổi từ máy khách.
* **Kiểm soát rõ ràng:** [Bind] và [BindNever] cho phép bạn kiểm soát chính xác các thuộc tính mô hình được điền từ các yêu cầu đến.
* **Hành vi mặc định:** Nếu không có các thuộc tính này, trình kết dính mô hình sẽ cố gắng liên kết tất cả các thuộc tính công khai của lớp mô hình của bạn.
* **Các loại phức tạp:** Bạn có thể sử dụng [Bind] trên các thuộc tính phức tạp lồng nhau để chỉ định thuộc tính nào trong các đối tượng đó sẽ được liên kết.

[Từ cơ thể] Thuộc tính

Thuộc tính [FromBody] là một công cụ quan trọng trong kho vũ khí liên kết mô hình của ASP.NET Core MVC, được thiết kế để xử lý các tình huống trong đó dữ liệu đến được chứa trong nội dung của yêu cầu HTTP. Điều này đặc biệt phổ biến khi làm việc với API và các ứng dụng web hiện đại thường xuyên trao đổi dữ liệu ở các định dạng như JSON hoặc XML.

**Cách thức hoạt động**

1. **Nhận dạng nội dung yêu cầu:** Khi một yêu cầu đến, phần mềm trung gian liên kết mô hình sẽ kiểm tra tiêu đề Content-Type để xác định định dạng của dữ liệu trong nội dung yêu cầu. Thông thường, đây là application/json cho dữ liệu JSON hoặc application/xml cho dữ liệu XML.
2. **Lựa chọn định dạng đầu vào:** Dựa trên Content-Type, phần mềm trung gian chọn một bộ định dạng đầu vào thích hợp. Các định dạng đầu vào chịu trách nhiệm giải tuần tự hóa nội dung yêu cầu thô thành định dạng mà trình kết dính mô hình có thể hiểu được.
3. **Liên kết mô hình:** Trình kết dính mô hình lấy dữ liệu đã giải tuần tự và cố gắng ánh xạ dữ liệu đó với các thuộc tính của lớp mô hình của bạn. Ánh xạ này thường được thực hiện dựa trên tên thuộc tính, nhưng bạn có thể tùy chỉnh nó bằng cách sử dụng các thuộc tính liên kết mô hình khác nhau.
4. **Xác thực:** Sau khi liên kết, mô hình trải qua quá trình xác thực để đảm bảo tuân thủ các quy tắc được xác định bởi chú thích dữ liệu hoặc logic xác thực tùy chỉnh.

**Lợi ích của [FromBody]**

* **Xử lý dữ liệu phức tạp:** Dễ dàng liên kết các đối tượng phức tạp với các thuộc tính lồng nhau từ tải trọng JSON hoặc XML.
* **Phân tách các mối quan tâm:** Trình định dạng đầu vào xử lý việc giải tuần tự, giữ cho các hành động của bộ điều khiển của bạn sạch sẽ.
* **Thân thiện với API:** Phù hợp với các phương pháp API RESTful, trong đó dữ liệu thường được truyền trong nội dung yêu cầu.

Mã

1. HomeController.cs
2. [Lộ trình ("đăng ký")]
3. Ví dụ JSON: { "PersonName": "william", "email": "william@example.com", "Phone": "123456", "Password": "william123", "ConfirmPassword": "william123" }
4. public IActionResult Index([FromBody] Person person)
5. {
6. nếu (! ModelState.IsValid)
7. {
8. // ... (xử lý lỗi xác thực)...
9. }
11. trả lại Content($"{person}");
12. }

Trong mã này:

1. **Thuộc tính [FromBody]:** Thuộc tính [FromBody] trên tham số person hướng dẫn trình kết dính mô hình tìm kiếm dữ liệu trong nội dung yêu cầu.
2. **JSON Deserialization:** Nếu yêu cầu có Content-Type là application/json, thì trình định dạng đầu vào JSON tích hợp sẽ giải tuần tự hóa dữ liệu JSON trong nội dung yêu cầu thành đối tượng Person.
3. **Xác thực mô hình:** Kiểm tra ModelState.IsValid đảm bảo đối tượng Person được giải tuần tự đáp ứng các tiêu chí xác thực của bạn.
4. **Phản hồi thành công:** Nếu mô hình hợp lệ, kết quả Nội dung trả về biểu diễn chuỗi của đối tượng Person.

**Những cân nhắc quan trọng**

* **Tham số [FromBody] đơn:** ASP.NET Liên kết mô hình cốt lõi chỉ cho phép một tham số cho mỗi phương thức hành động được trang trí bằng [FromBody]. Điều này là do nội dung yêu cầu thường là một luồng dữ liệu duy nhất.
* **Content-Type:** Tiêu đề Content-Type của yêu cầu phải khớp với định dạng dự kiến (ví dụ: application/json) để sử dụng đúng định dạng đầu vào.
* **Bảo mật:** Luôn xác thực và làm sạch dữ liệu từ cơ quan yêu cầu để bảo vệ khỏi các lỗ hổng như đăng tải quá mức và tấn công tiêm.

Định dạng đầu vào

Input formatter là các thành phần chuyên biệt trong ASP.NET Core MVC chịu trách nhiệm giải tuần tự hóa dữ liệu từ nội dung của các yêu cầu HTTP. Khi một yêu cầu đến với tải trọng, trình định dạng đầu vào sẽ giải mã dữ liệu này thành một định dạng (ví dụ: đối tượng C#, bộ sưu tập) mà các phương thức hành động của bạn có thể dễ dàng hoạt động.

**Cách thức hoạt động của Input Formatters**

1. **Đàm phán nội dung:** Quá trình liên kết mô hình bắt đầu bằng đàm phán nội dung, trong đó ASP.NET Core kiểm tra tiêu đề Content-Type của yêu cầu để xác định định dạng của dữ liệu đến (ví dụ: JSON, XML).
2. **Lựa chọn định dạng đầu vào:** Dựa trên Loại nội dung, ASP.NET Core chọn một định dạng đầu vào thích hợp biết cách xử lý định dạng dữ liệu cụ thể đó.
3. **Deserialization:** Trình định dạng đầu vào đã chọn giải tuần tự dữ liệu thô từ nội dung yêu cầu thành các đối tượng C#, bộ sưu tập hoặc các loại được hỗ trợ khác.
4. **Liên kết mô hình:** Dữ liệu được giải tuần tự sau đó được chuyển đến trình kết dính mô hình, điền các tham số của phương thức hành động của bạn.

**Các định dạng đầu vào phổ biến**

* **NewtonsoftJsonInputFormatter:** Xử lý dữ liệu JSON (Ký hiệu đối tượng JavaScript) bằng cách sử dụng thư viện Newtonsoft.Json phổ biến.
* **SystemTextJsonInputFormatter:** Xử lý dữ liệu JSON bằng trình tuần tự System.Text.Json tích hợp sẵn.
* **XmlSerializerInputFormatter:** Xử lý dữ liệu XML (Ngôn ngữ đánh dấu mở rộng) bằng cách sử dụng XmlSerializer.

**Cấu hình định dạng đầu vào**

1. **Định dạng mặc định:** ASP.NET Core MVC bao gồm NewtonsoftJsonInputFormatter làm bộ định dạng mặc định.
2. **Các định dạng bổ sung:** Bạn có thể thêm hỗ trợ cho các trình định dạng khác (như XmlSerializerInputFormatter) bằng cách đăng ký chúng một cách rõ ràng trong cấu hình khởi động của ứng dụng của bạn.

Mã

1. var builder = WebApplication.CreateBuilder(args);
2. chủ thầu. Services.AddControllers(). AddXmlSerializerFormatters();
3. // ... cấu hình khác ...

Trong mã này, tệp . AddXmlSerializerFormatters() đăng ký XmlSerializerInputFormatter, cho phép ứng dụng của bạn xử lý các yêu cầu với kiểu nội dung application/xml.

**Sử dụng định dạng đầu vào**

* **Liên kết ngầm:** Nếu tiêu đề Content-Type của yêu cầu khớp với định dạng được hỗ trợ, thì trình định dạng đầu vào tương ứng sẽ tự động được sử dụng. Bạn không cần phải chỉ định rõ ràng trình định dạng nào sẽ sử dụng trong phương thức hành động của mình.
* **Liên kết rõ ràng ([FromBody]):** Bạn có thể sử dụng thuộc tính [FromBody] trên tham số phương thức hành động để yêu cầu trình kết dính mô hình tìm kiếm dữ liệu trong nội dung yêu cầu một cách rõ ràng. Điều này thường được sử dụng khi bạn có các đối tượng phức tạp cần được giải tuần tự.

**Những cân nhắc quan trọng**

* **Đàm phán nội dung:** Sự thành công của liên kết mô hình phụ thuộc vào việc máy khách gửi tiêu đề Content-Type hợp lệ mà ứng dụng của bạn hỗ trợ.
* **Xử lý lỗi:** Xử lý các lỗi deserialization tiềm ẩn một cách duyên dáng. Nếu trình định dạng đầu vào không thể phân tích cú pháp nội dung yêu cầu, hãy trả về phản hồi lỗi thích hợp (ví dụ: 400 Bad Request).
* **Bảo mật:** Luôn xác thực và vệ sinh dữ liệu được giải trình từ nội dung yêu cầu để bảo vệ khỏi các lỗ hổng bảo mật.
* **Định dạng đầu vào tùy chỉnh:** Đối với các tình huống chuyên biệt cao hoặc định dạng dữ liệu tùy chỉnh, bạn có thể tạo các định dạng đầu vào của riêng mình bằng cách triển khai giao diện IInputFormatter.

Chất kết dính mô hình tùy chỉnh

Mặc dù chất kết dính mô hình mặc định của ASP.NET Core khá linh hoạt, nhưng nó có thể không phải lúc nào cũng đáp ứng nhu cầu cụ thể của bạn. Đây là lúc các liên kết mô hình tùy chỉnh tham gia, cho phép bạn xác định chính xác cách dữ liệu được trích xuất từ các yêu cầu đến và ánh xạ vào thuộc tính mô hình của bạn.

**Mục đích**

* **Tính linh hoạt:** Xử lý các định dạng dữ liệu phức tạp hoặc tùy chỉnh mà trình kết dính mô hình mặc định không hiểu.
* **Logic tùy chỉnh:** Triển khai các quy tắc kinh doanh cụ thể hoặc chuyển đổi dữ liệu trong quá trình ràng buộc.
* **Kiểm soát hoàn toàn:** Kiểm soát hoàn toàn quá trình liên kết, từ phân tích cú pháp dữ liệu thô đến điền đối tượng mô hình của bạn.

**Triển khai IModelBinder**

Để tạo một chất kết dính mô hình tùy chỉnh, bạn triển khai giao diện IModelBinder:

1. giao diện công cộng IModelBinder
2. {
3. Nhiệm vụ BindModelAsync(ModelBindingContext bindingContext);
4. }

Cốt lõi của logic tùy chỉnh nằm trong phương thức BindModelAsync, nơi bạn:

1. Truy xuất dữ liệu thô từ bindingContext.ValueProvider.
2. Phân tích cú pháp và xác thực dữ liệu theo yêu cầu của bạn.
3. Tạo một thực thể của lớp mô hình của bạn và điền các thuộc tính của nó.
4. Đặt bindingContext.Result thành ModelBindingResult.Success(yourModelInstance).

Mã

1. PersonModelBinder.cs
2. lớp công cộng PersonModelBinder : IModelBinder
3. {
4. tác vụ công khai BindModelAsync(ModelBindingContext bindingContext)
5. {
6. Person person = new Person();
8. // ... (Logic để trích xuất và điền các thuộc tính từ ValueProvider)...
10. bindingContext.Result = ModelBindingResult.Success(người);
11. trả về Task.CompletedTask;
12. }
13. }

Trong ví dụ này, PersonModelBinder:

1. Tạo một đối tượng Person mới.
2. Trích xuất các giá trị cho các thuộc tính khác nhau (ví dụ: PersonName, Email, Phone) từ ValueProvider (có thể truy cập dữ liệu biểu mẫu, chuỗi truy vấn, dữ liệu tuyến đường, v.v.).
3. Thực hiện một số biến đổi cơ bản (nối FirstName và LastName).
4. Đặt kết quả liên kết mô hình để cho biết thành công và trả về đối tượng Person đã điền.

Nhà cung cấp chất kết dính mô hình

Để thông báo cho ASP.NET Core rằng bạn muốn sử dụng chất kết dính mô hình tùy chỉnh của mình cho một loại cụ thể, bạn tạo một nhà cung cấp kết dính mô hình. Nhà cung cấp này triển khai giao diện IModelBinderProvider.

Mã

1. (Mã này không được cung cấp trong yêu cầu ban đầu của bạn, nhưng đó là một cách phổ biến để đăng ký một chất kết dính mô hình tùy chỉnh)
3. lớp công cộng PersonBinderProvider: IModelBinderProvider
4. {
5. IModelBinder công khai? GetBinder(Ngữ cảnh ModelBinderProviderContext)
6. {
7. if (ngữ cảnh. Metadata.ModelType == typeof(Person))
8. {
9. trả về mới BinderTypeModelBinder(typeof(PersonModelBinder));
10. }
11. trả về null;
12. }
13. }

Nhà cung cấp này kiểm tra xem loại mô hình có phải là Person hay không và nếu có, nó sẽ trả về một thực thể của PersonModelBinder của bạn.

**Đăng ký và sử dụng**

1. Program.cs (hoặc Startup.cs)
2. chủ thầu. Services.AddControllers(tùy chọn => {
3. Tùy chọn. ModelBinderProviders.Insert(0, new PersonBinderProvider());
4. });

Bằng cách chèn PersonBinderProvider của bạn vào chỉ mục 0, bạn đảm bảo nó được ưu tiên hơn các liên kết mô hình mặc định.

**Dữ liệu yêu cầu mẫu (Postman)**

Để kiểm tra điều này, bạn có thể sử dụng Postman để gửi yêu cầu POST đến hành động của bộ điều khiển với nội dung JSON sau:

JSON

1. {
2. "Tên": "John",
3. "Họ": "Doe",
4. "Email": "john.doe@example.com",
5. "Điện thoại": "1234567890",
6. "Mật khẩu": "mật khẩu123",
7. "Xác nhận mật khẩu": "mật khẩu123",
8. "Giá": 59.99,
9. "Ngày sinh": "2000-01-01"
10. }

**Những cân nhắc quan trọng**

* **Độ phức tạp:** Các chất kết dính mô hình tùy chỉnh có thể trở nên phức tạp, vì vậy hãy sử dụng chúng một cách thận trọng khi hành vi mặc định không đủ.
* **Khả năng kiểm tra:** Viết kiểm thử đơn vị cho các liên kết mô hình tùy chỉnh của bạn để đảm bảo chúng hoạt động chính xác trong các tình huống khác nhau.
* **Hiệu suất:** Hãy chú ý đến hiệu suất khi triển khai logic phân tích cú pháp hoặc xác thực phức tạp trong kết dính của bạn.
* **Xử lý lỗi:** Xử lý các ngoại lệ tiềm ẩn trong quá trình trích xuất và xác thực dữ liệu để cung cấp phản hồi lỗi thông tin.
* **Phương pháp tiếp cận thay thế:** Trong một số trường hợp, sử dụng trình định dạng đầu vào tùy chỉnh kết hợp với trình kết dính mô hình đơn giản hơn có thể là một cách tiếp cận phù hợp hơn.

Ràng buộc bộ sưu tập

Liên kết mô hình của ASP.NET Core không giới hạn ở các thuộc tính đơn giản; Nó có thể xử lý các bộ sưu tập như danh sách và mảng trong các lớp mô hình của bạn một cách duyên dáng. Điều này đặc biệt hữu ích khi xử lý các biểu mẫu cho phép người dùng nhập nhiều giá trị cho một trường duy nhất (ví dụ: chọn nhiều sở thích từ danh sách hộp kiểm) hoặc khi làm việc với dữ liệu từ các API trả về các bộ sưu tập một cách tự nhiên.

**Cách hoạt động của liên kết bộ sưu tập**

1. **Thuộc tính bộ sưu tập trong mô hình:** Lớp mô hình của bạn phải có thuộc tính là kiểu bộ sưu tập (ví dụ: List<T>, T[]).
2. **Quy ước đặt tên:** Các tham số yêu cầu đến phải tuân theo một quy ước đặt tên cụ thể để cho biết giá trị nào thuộc về bộ sưu tập.
3. **Model Binder Magic:** Trình kết dính mô hình tự động nhận dạng quy ước đặt tên và điền thuộc tính bộ sưu tập cho phù hợp.

**Quy ước đặt tên cho liên kết bộ sưu tập**

* **Được lập chỉ mục:** items[0], items[1], items[2], ... (Được sử dụng cho danh sách và mảng)
* **Cùng tên:** vật phẩm, vật phẩm, vật phẩm,... (Được sử dụng cho các bộ sưu tập như ICollection<T>)

Mã

1. Person.cs (Mô hình)
2. người lớp công cộng
3. {
4. // ... Các thuộc tính khác
5. public List<string?> Tags { get; set; } = mới List<string?>(); Thuộc tính bộ sưu tập
6. }
8. HomeController.cs
9. IActionChỉ số kết quả công khai(Người người)
10. {
11. // ... logic xác thực và phản hồi ...
12. return Content($"Person: {person}, Tags: {string. Join(",", người. Thẻ)}", "văn bản/đơn giản");
13. }

**Dữ liệu yêu cầu mẫu (Postman)**

Để kiểm tra điều này, bạn có thể gửi dữ liệu yêu cầu JSON sau đến điểm cuối đăng ký của mình bằng Postman:

JSON

1. {
2. "PersonName": "Alice",
3. "Email": "alice@example.com",
4. "Điện thoại": "1234567890",
5. "Mật khẩu": "alicepassword",
6. "Xác nhận mật khẩu": "alicepassword",
7. "Giá": 59.99,
8. "Ngày sinh": "1995-03-15",
9. "tags": ["âm nhạc", "đọc", "mã hóa"]
10. }

**Phản hồi** Phản hồi sẽ giống như sau:

1. Đối tượng người - Tên người: Alice, Email: alice@example.com, Điện thoại: 1234567890, Mật khẩu: alicepassword, Xác nhận mật khẩu: alicepassword, Giá: 59.99, Tags: âm nhạc, đọc, mã hóa

**Giải thích chi tiết**

1. **Thuộc tính bộ sưu tập:** Mô hình Person có thuộc tính Tags, là List<string?>.
2. **Dữ liệu JSON:** Nội dung yêu cầu bao gồm một mảng Thẻ với các giá trị chuỗi.
3. **Liên kết mô hình:** Trình kết dính mô hình tự động nhận dạng mảng Tags trong dữ liệu JSON và điền danh sách Person.Tags với các giá trị tương ứng.

**Ghi chú**

* **Đặt tên:** Thực hiện theo quy ước đặt tên chính xác (được lập chỉ mục hoặc cùng tên) cho các tham số bộ sưu tập của bạn trong dữ liệu yêu cầu.
* **Tính linh hoạt:** Bạn có thể liên kết với nhiều loại bộ sưu tập, bao gồm danh sách, mảng và bộ sưu tập tùy chỉnh triển khai ICollection<T>.
* **Xác thực:** Áp dụng các thuộc tính xác thực cho thuộc tính bộ sưu tập của bạn (ví dụ: [Bắt buộc], [MaxLength]) để đảm bảo tính toàn vẹn của dữ liệu.
* **Chất kết dính mô hình tùy chỉnh:** Nếu bạn có các tình huống liên kết bộ sưu tập phức tạp, bạn có thể tạo các liên kết mô hình tùy chỉnh để xử lý chúng.

[TừTiêu đề] Thuộc tính

Trong ASP.NET Core MVC, thuộc tính [FromHeader] được sử dụng để hướng dẫn trình kết dính mô hình tìm nạp giá trị cho các tham số phương thức hành động trực tiếp từ các tiêu đề yêu cầu HTTP. Tiêu đề HTTP là các cặp khóa-giá trị cung cấp siêu dữ liệu về yêu cầu, chẳng hạn như loại trình duyệt của máy khách (User-Agent), loại nội dung được chấp nhận (Chấp nhận) và mã thông báo ủy quyền.

**Cách thức hoạt động của [FromHeader]**

1. **Nhận dạng tiêu đề:** Khi một yêu cầu đến, hệ thống liên kết mô hình của ASP.NET Core xác định các tham số hành động được đánh dấu bằng thuộc tính [FromHeader].
2. **Trích xuất tiêu đề:** Sau đó, nó kiểm tra các tiêu đề yêu cầu để xác định các tiêu đề khớp với tên được chỉ định trong thuộc tính [FromHeader].
3. **Gán giá trị:** Nếu tìm thấy tiêu đề phù hợp, giá trị của nó sẽ được gán cho tham số hành động tương ứng. Nếu tiêu đề không xuất hiện hoặc giá trị của nó không thể được chuyển đổi thành loại của tham số, trạng thái mô hình sẽ được đánh dấu là không hợp lệ.

**Tại sao nên sử dụng [FromHeader]?**

* **Truy cập vào Siêu dữ liệu:** Tiêu đề HTTP chứa thông tin có giá trị về máy khách, yêu cầu và dữ liệu đang được truyền.
* **Thông số tùy chỉnh:** Bạn có thể xác định tiêu đề tùy chỉnh để chuyển dữ liệu bổ sung đến API của mình.
* **Bảo mật:** Tiêu đề thường được sử dụng để truyền mã thông báo xác thực và các thông tin liên quan đến bảo mật khác.
* **Đàm phán nội dung:** Các tiêu đề như Chấp nhận được sử dụng để xác định định dạng ưu tiên cho phản hồi (ví dụ: JSON, XML).

Mã

1. HomeController.cs
2. [Lộ trình ("đăng ký")]
3. public IActionResult Index(Person person, [FromHeader(Name = "User-Agent")] chuỗi UserAgent)
4. {
5. // ... (logic xác thực mô hình)...
7. trả lại Content($"{person}, {UserAgent}"); Bao gồm User-Agent trong phản hồi
8. }

Trong mã này:

1. **Tham số UserAgent:** Phương thức hành động hiện bao gồm một tham số UserAgent với thuộc tính [FromHeader]. Thuộc tính Tên của thuộc tính được đặt thành "User-Agent", cho biết rằng tham số này phải được liên kết với giá trị của tiêu đề User-Agent.
2. **Trích xuất tiêu đề:** Khi một yêu cầu được thực hiện đến điểm cuối /register, trình kết dính mô hình sẽ tìm kiếm tiêu đề User-Agent trong yêu cầu và gán giá trị của nó cho tham số UserAgent.
3. **Phản hồi:** Kết quả Nội dung hiện bao gồm cả thông tin của người đó (từ nội dung yêu cầu) và giá trị của tiêu đề User-Agent trong phản hồi.

**Dữ liệu yêu cầu mẫu (Postman)**

Để kiểm tra điều này, bạn sẽ gửi yêu cầu POST đến điểm cuối /register bằng cách sử dụng Postman, với cùng nội dung JSON như trước đây, nhưng lần này, bạn cũng cần thêm tiêu đề User-Agent trong tab Headers với giá trị như:

1. Mozilla/5.0 (Windows NT 10.0; Thắng64; x64) AppleWebKit / 537.36 (KHTML, như Gecko) Chrome / 58.0.3029.110 Safari / 537.3

**Những cân nhắc quan trọng**

* **Không phân biệt chữ hoa chữ thường:** Tên tiêu đề không phân biệt chữ hoa chữ thường, vì vậy bạn có thể sử dụng [FromHeader(Name = "user-agent")] hoặc [FromHeader(Name = "USER-AGENT")].
* **Nhiều tiêu đề:** Bạn có thể sử dụng [FromHeader] trên nhiều tham số để liên kết các giá trị từ các tiêu đề khác nhau.
* **Giá trị mặc định:** Nếu tiêu đề không có trong yêu cầu, bạn có thể chỉ định giá trị mặc định cho tham số bằng cách sử dụng toán tử ? (ví dụ: chuỗi? UserAgent).
* **Cách thay thế:** Nếu bạn cần truy cập nhiều tiêu đề hoặc có logic phân tích cú pháp tiêu đề phức tạp hơn, hãy cân nhắc sử dụng Request.Headers trực tiếp trong phương thức hành động của bạn.

Những điểm chính cần nhớ

**1. Liên kết mô hình: Cầu nối HTTP và C#**

* **Mục đích:** Tự động ánh xạ dữ liệu từ các yêu cầu HTTP (dữ liệu biểu mẫu, giá trị tuyến đường, chuỗi truy vấn, tiêu đề, nội dung) đến các tham số phương thức hành động hoặc thuộc tính mô hình.
* **Lợi ích:** Giảm mã nguyên mẫu, cung cấp khả năng nhập mạnh và đơn giản hóa việc xử lý dữ liệu trong các hành động.
* **Quá trình:**
  1. **Phân tích yêu cầu:** Kiểm tra loại nội dung và phương thức của yêu cầu.
  2. **Nhà cung cấp giá trị:** Tạo nhà cung cấp giá trị để truy cập dữ liệu từ các nguồn khác nhau.
  3. **Lựa chọn chất kết dính mô hình:** Chọn chất kết dính mô hình thích hợp dựa trên loại tham số và thuộc tính.
  4. **Ánh xạ thuộc tính:** Ánh xạ các giá trị từ nhà cung cấp giá trị sang các thuộc tính mô hình dựa trên đối sánh tên và thuộc tính.

**2. Xác thực mô hình: Đảm bảo tính toàn vẹn của dữ liệu**

* **Mục đích:** Đảm bảo rằng dữ liệu được gửi đến đơn đăng ký của bạn đáp ứng các tiêu chí được xác định trước trước khi xử lý.
* **Tại sao nó lại quan trọng:** Tăng cường bảo mật, duy trì tính toàn vẹn của dữ liệu và cải thiện trải nghiệm người dùng.
* **Phương pháp:**
  + **Chú thích dữ liệu:** Sử dụng các thuộc tính như [Bắt buộc], [StringLength], [Phạm vi], v.v. (từ System.ComponentModel.DataAnnotations) để trang trí các thuộc tính mô hình.
  + **IValidatableObject:** Triển khai giao diện này để thực hiện logic xác thực cấp mô hình tùy chỉnh.
  + **Thuộc tính xác thực tùy chỉnh:** Tạo các thuộc tính của riêng bạn kế thừa từ ValidationAttribute cho các quy tắc phức tạp hơn.

**3. Trạng thái mô hình:**

* **Xác thực tập trung:** Đối tượng ModelState theo dõi trạng thái xác thực của mô hình của bạn sau khi liên kết.
* **ModelState.IsValid:** Một thuộc tính boolean cho biết mô hình hợp lệ hay chứa lỗi.
* **ModelState.AddModelError:** Thêm thông báo lỗi tùy chỉnh vào ModelState.

**4. Thuộc tính: Tinh chỉnh liên kết và xác thực mô hình**

* **[FromQuery]:** Liên kết các tham số từ chuỗi truy vấn.
* **[FromRoute]:** Liên kết các tham số từ dữ liệu tuyến đường (phân đoạn URL).
* **[FromBody]:** Liên kết các đối tượng phức tạp từ nội dung yêu cầu (JSON, XML).
* **[FromHeader]:** Liên kết các tham số từ tiêu đề HTTP.
* **[Bind]:** Bao gồm rõ ràng các thuộc tính cụ thể để liên kết.
* **[BindNever]:** Loại trừ các thuộc tính cụ thể khỏi liên kết (ngăn đăng quá mức).

**5. Chất kết dính mô hình tùy chỉnh**

* **Mục đích:** Tạo logic của riêng bạn để trích xuất và ánh xạ dữ liệu vào các mô hình khi hành vi mặc định không đủ.
* **Giao diện IModelBinder:**  Triển khai giao diện này để xác định logic liên kết tùy chỉnh của bạn.
* **ModelBindingContext:** Đối tượng ngữ cảnh này cung cấp quyền truy cập vào các nhà cung cấp giá trị, siêu dữ liệu mô hình và các thông tin liên quan khác.

**Mẹo bổ sung**

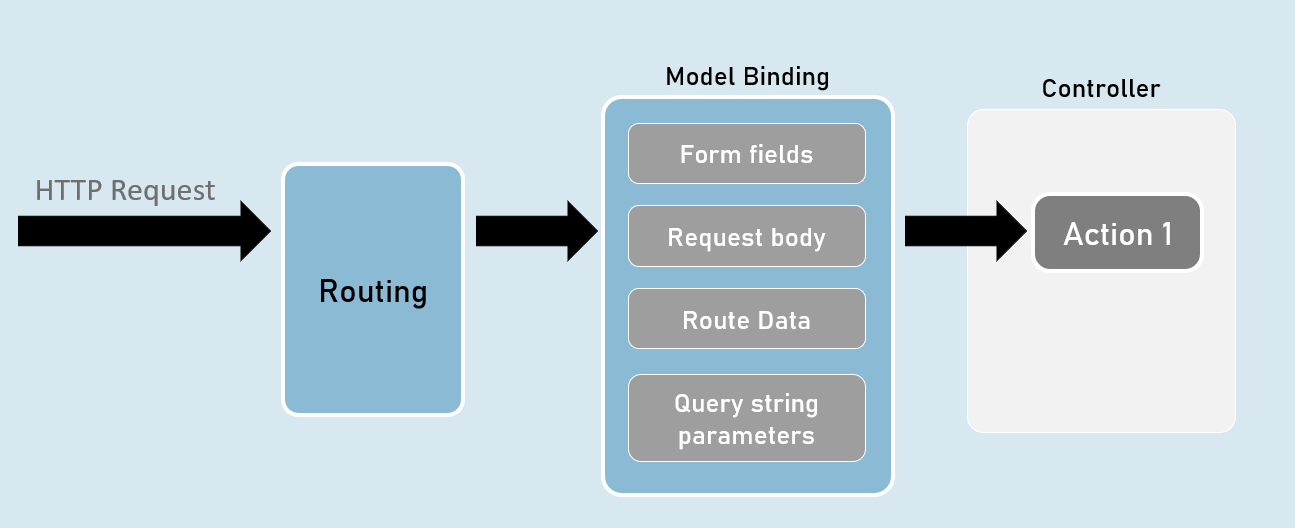
* **Trình kết dính mô hình mặc định:** Hiểu hành vi liên kết mô hình mặc định và thời điểm bạn cần tùy chỉnh.
* **Trình định dạng đầu vào:** Biết cách thức hoạt động của các bộ định dạng đầu vào để giải tuần tự hóa nội dung yêu cầu ở các định dạng khác nhau (JSON, XML).
* **Liên kết bộ sưu tập:** Làm quen với cách liên kết các bộ sưu tập (danh sách, mảng) bằng cách sử dụng các quy ước đặt tên thích hợp.
* **Xử lý lỗi:** Luôn kiểm tra ModelState.IsValid trong các hành động của bạn và xử lý các trạng thái mô hình không hợp lệ một cách dễ dàng.
* **Bảo mật:** Ưu tiên bảo mật bằng cách xác thực và vệ sinh dữ liệu đầu vào để ngăn chặn các cuộc tấn công.

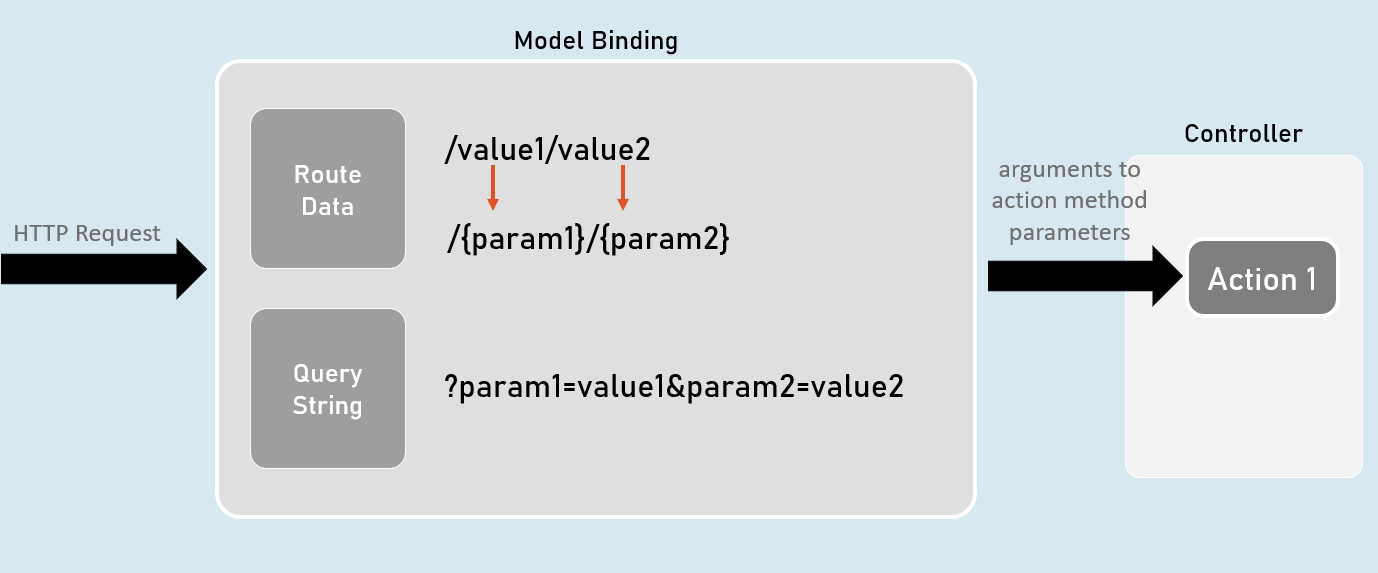
Liên kết mô hình

Ràng buộc mô hình là một tính năng của lõi asp.net đọc các giá trị từ các yêu cầu http và chuyển chúng dưới dạng đối số cho phương thức hành động.

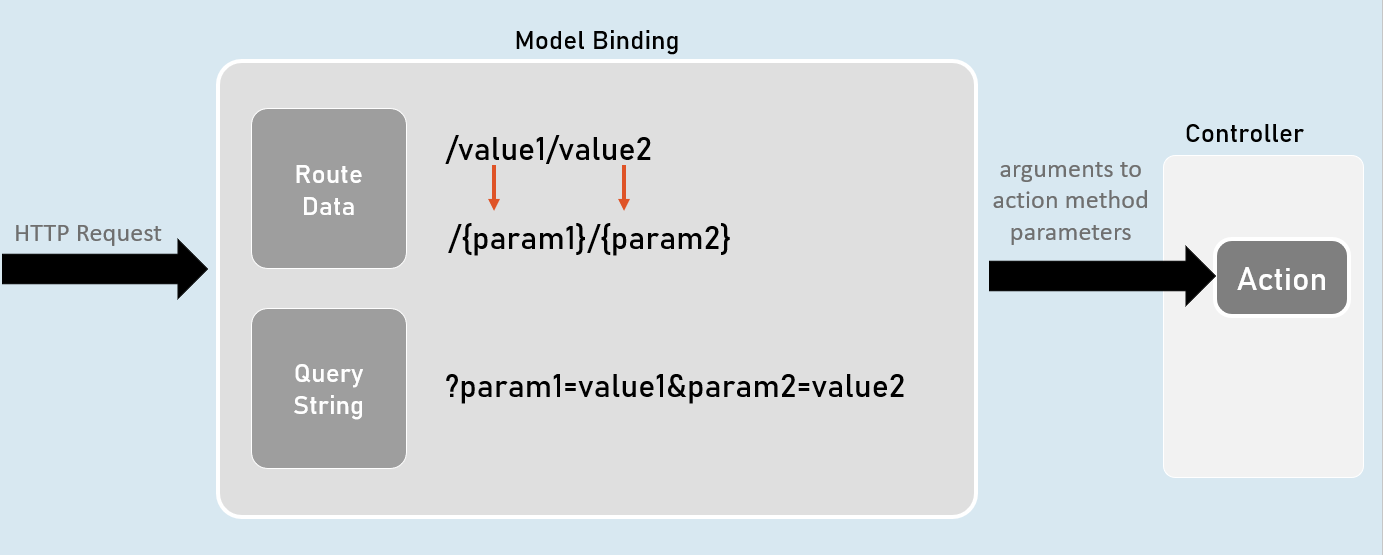


QueryString so với RouteData





[FromQuery] và [FromRoute]

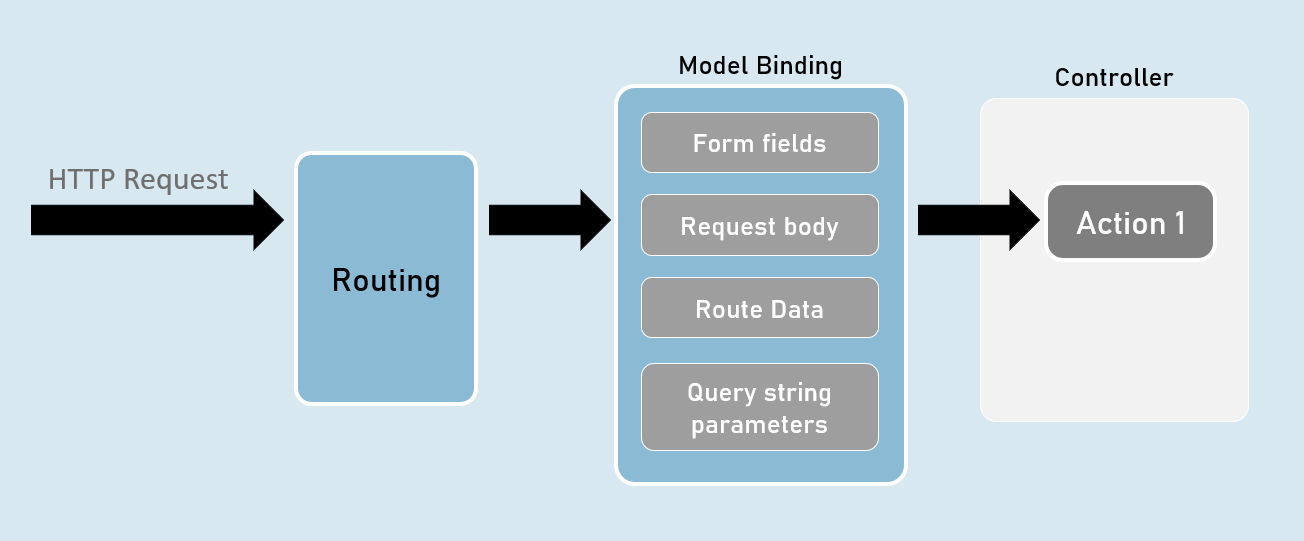


[TừTruy vấn]

1. Chỉ lấy giá trị từ chuỗi truy vấn
2. public IActionResult ActionMethodName( [FromQuery] tham số loại)
3. {
4. }

[TừTuyến đường]

1. chỉ lấy giá trị từ các tham số tuyến đường
2. public IActionResult ActionMethodName( [FromRoute] tham số loại)
3. {
4. }



Mô hình

Model là một lớp đại diện cho cấu trúc của dữ liệu (dưới dạng thuộc tính) mà bạn muốn nhận từ yêu cầu và/hoặc gửi đến phản hồi.

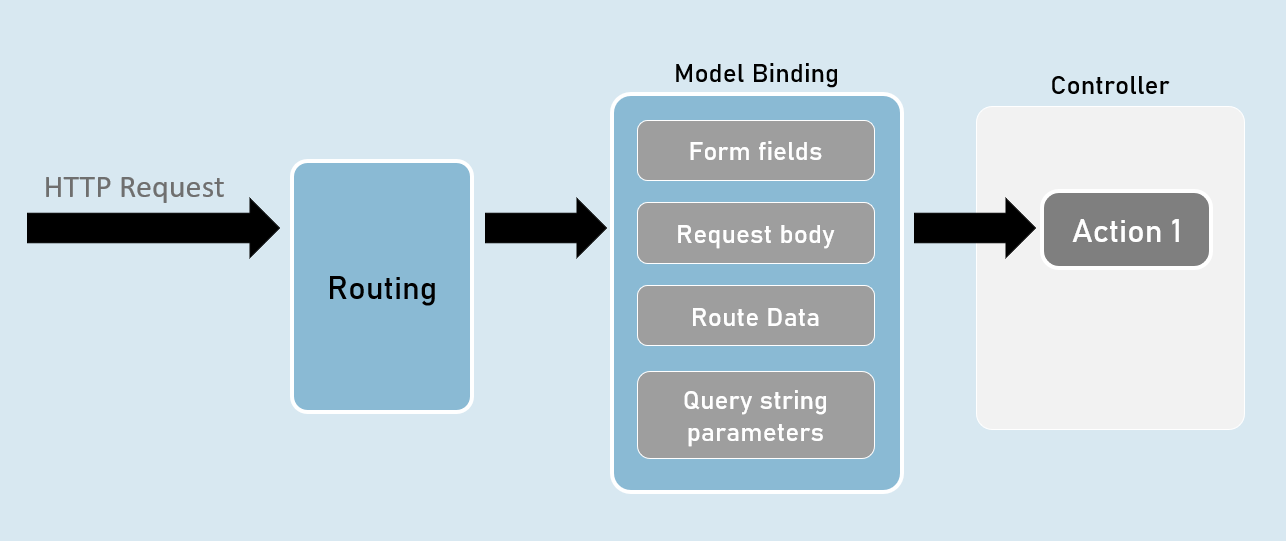
Còn được gọi là POCO (Plain Old CLR Objects).



Mẫu

1. class ClassName
2. {
3. loại công khai PropertyName { get; set; }
4. }

form-urlencoded và form-data



form-urlencoded (mặc định)

**Tiêu đề yêu cầu**

Loại nội dung: application/x-www-form-urlencoded

**Nội dung yêu cầu**

param1 = giá trị 1 & param2 = giá trị 2

dữ liệu biểu mẫu

**Tiêu đề yêu cầu**

Loại nội dung: multipart/form-data

**Nội dung yêu cầu**

--------------------------d74496d66958873e

Nội dung-Bố trí: biểu mẫu-dữ liệu; name="param1"

giá trị1

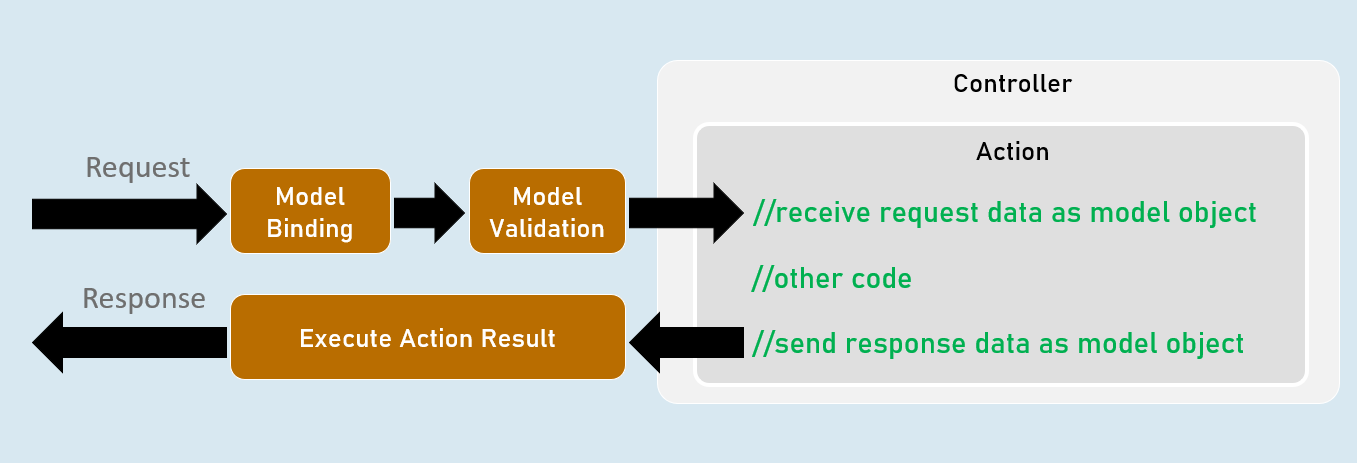
--------------------------d74496d66958873e

Nội dung-Bố trí: biểu mẫu-dữ liệu; name="param2"

giá trị2

Xác thực mô hình

1. class ClassName
2. {
3. [Thuộc tính] // áp dụng quy tắc xác thực cho thuộc tính này
4. loại công khai PropertyName { get; set; }
5. }



Trạng thái mô hình

**Có giá trị**

Chỉ định xem có ít nhất một lỗi xác thực hay không (đúng hoặc sai).

**Giá trị**

Chứa từng giá trị thuộc tính mô hình với thuộc tính "Lỗi" tương ứng chứa danh sách các lỗi xác thực của thuộc tính mô hình đó.

**Đếm lỗi**

Trả về số lỗi.

Xác thực mô hình

**[Bắt buộc(ErrorMessage = "giá trị")]**

Chỉ định rằng giá trị thuộc tính là bắt buộc (không được để trống hoặc trống).

**[StringLength (int maximumLength, MinimumLength = value, ErrorMessage = "value")]**

Chỉ định độ dài tối thiểu và tối đa (số ký tự) được phép trong chuỗi.

**[Phạm vi (int tối thiểu, int tối đa, ErrorMessage = "value")]**

Chỉ định giá trị số tối thiểu và tối đa cho phép.

**[RegularExpression(mẫu chuỗi, ErrorMessage = "value")]**

Chỉ định mẫu hợp lệ (biểu thức chính quy).

**[EmailAddress(ErrorMessage = "value")]**

Chỉ định rằng giá trị phải là một địa chỉ email hợp lệ.

**[Điện thoại (ErrorMessage = "giá trị")]**

Chỉ định rằng giá trị phải là một số điện thoại hợp lệ).

Ví dụ: (999) -999-9999 hoặc 9876543210

**[So sánh (chuỗi otherProperty, ErrorMessage = "value")]**

Chỉ định rằng các giá trị của thuộc tính hiện tại và thuộc tính khác phải giống nhau.

**[Url (ErrorMessage = "giá trị")]**

Chỉ định rằng giá trị phải là một url hợp lệ (địa chỉ trang web).

Ví dụ: http://www.example.com

**[Xác thựcKhông bao giờ]**

Chỉ định rằng thuộc tính không nên được xác thực (loại trừ thuộc tính khỏi xác thực mô hình).

Xác thực tùy chỉnh

1. class ClassName : ValidationAttribute
2. {
3. public ghi đè ValidationResult? IsValid(giá trị đối tượng, ValidationContext validationContext)
4. {
5. trả về ValidationResult.Success;
6. [hoặc] trả về ValidationResult("thông báo lỗi");
7. }
8. }

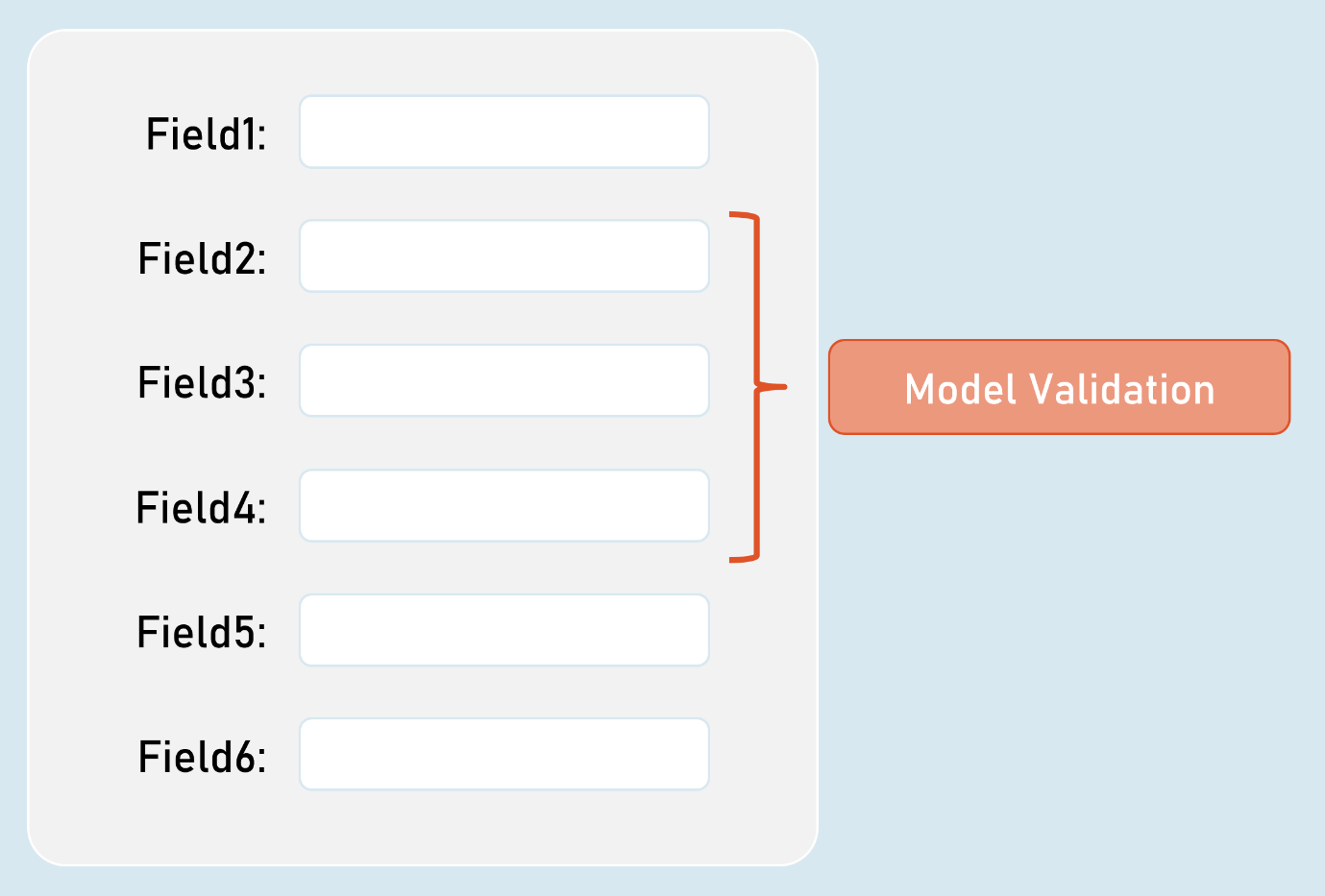
**Thuộc tính xác thực**

* Lớp cơ sở cho tất cả các thuộc tính xác thực như RequiredAttribute, RegularExpressionAttribute, RangeAttribute, StringLengthAttribute, CompareAttribute, v.v.
* Cung cấp các thuộc tính như ErrorMessage và các phương thức như Validate(), IsValid(), v.v.

**Ngữ cảnh xác thực**

* Hoạt động như một tham số cho phương thức "IsValid()" của các lớp thuộc tính xác thực tùy chỉnh.
* Cung cấp các thuộc tính như ObjectType, ObjectInstance.

Xác thực tùy chỉnh với nhiều thuộc tính



IValidatableObject

1. class Tên lớp: IValidatableObject
2. {
3. thuộc tính mô hình ở đây
5. public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
6. {
7. if (điều kiện)
8. {
9. yield return new ValidationResult("thông báo lỗi");
10. }
11. }
12. }

* Lớp cơ sở cho các lớp mô hình có xác thực.
* Cung cấp một phương thức được gọi là Validate() để xác định logic xác thực cấp lớp.
* Phương thức Validate() thực thi sau khi xác thực tất cả các xác thực cấp thuộc tính được thực thi; nhưng không thực thi nếu ít nhất một lỗi xác thực cấp thuộc tính.

**Ngữ cảnh xác thực**

* Hoạt động như một tham số cho phương thức "Validate()" của các lớp mô hình với IValidatableObject.
* Cung cấp các thuộc tính như ObjectType, ObjectInstance.

[Bind] và [BindNever]

**[Ràng buộc]**

1. lớp ClassNameController
2. {
3. công khai IActionResult ActionMethodName( [Bind(nameof(ClassName.PropertyName), nameof(ClassName.PropertyName)) )] ClassName parameterName)
4. {
5. }
6. }

* [Bind] chỉ định rằng chỉ các thuộc tính được chỉ định mới được đưa vào liên kết mô hình.
* Ngăn chặn việc đăng quá mức (đăng giá trị vào các thuộc tính không mong muốn), đặc biệt là trong các tình huống 'Tạo'.

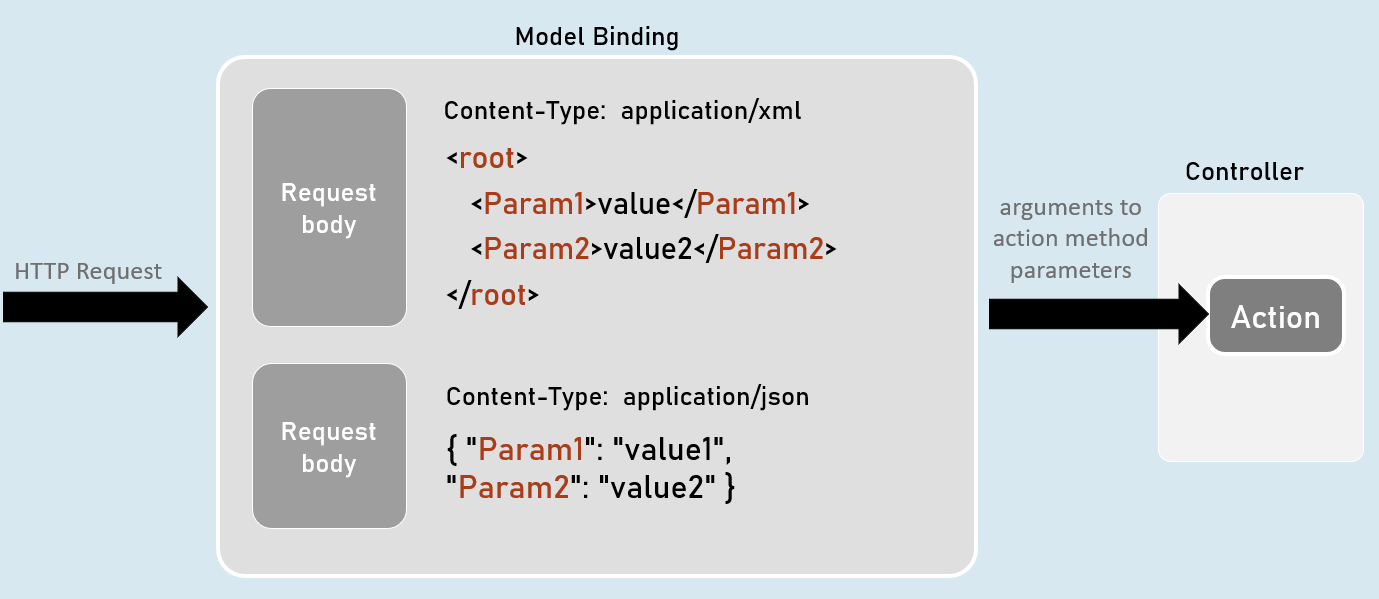
**[Ràng buộc không bao giờ]**

1. lớp ModelClassName
2. {
3. [Ràng buộc không bao giờ]
4. loại công khai PropertyName { get; set; }
5. }

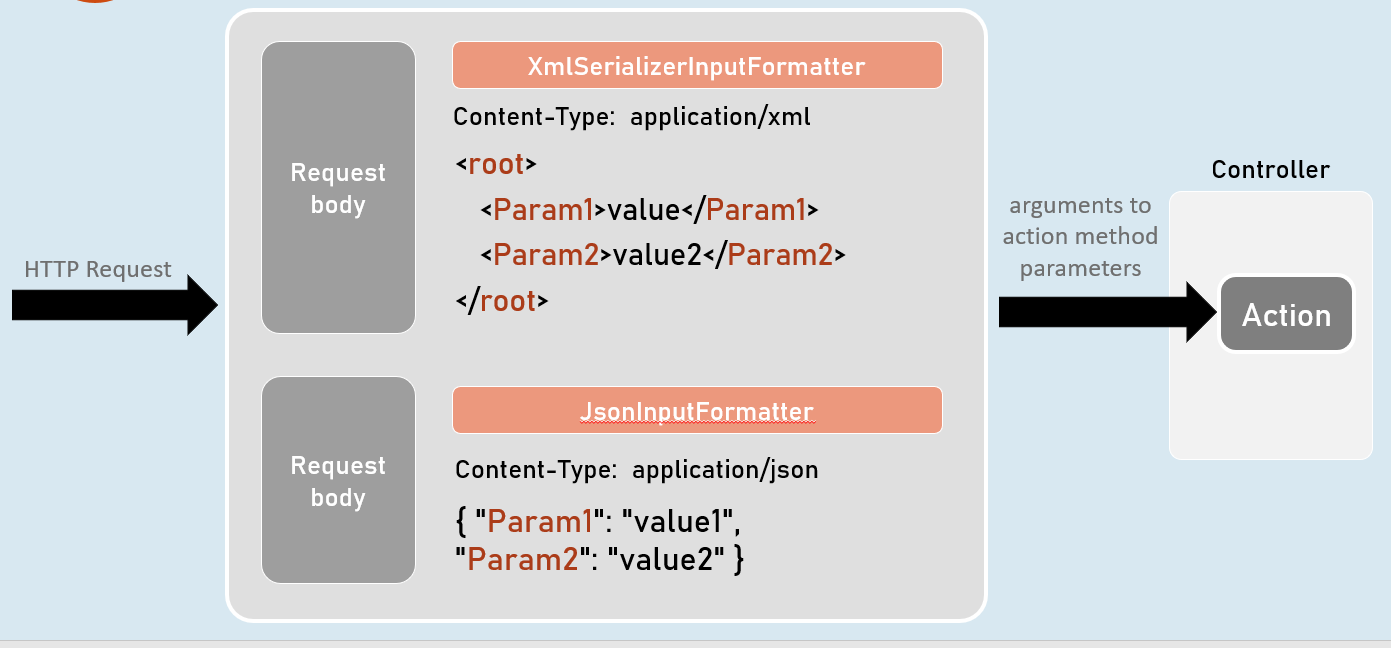
* [BindNever] chỉ định rằng thuộc tính được chỉ định KHÔNG nên được đưa vào liên kết mô hình.
* Hữu ích khi bạn có ít thuộc tính hơn để loại bỏ khỏi liên kết mô hình.

[Từ cơ thể]

1. chỉ cho phép các trình định dạng đầu vào đọc dữ liệu từ nội dung yêu cầu (dưới dạng JSON hoặc XML hoặc tùy chỉnh)
2. public IActionResult ActionMethodName( [FromBody] tham số loại)
3. {
4. }



Định dạng đầu vào



Chất kết dính mô hình tùy chỉnh

**Chất kết dính mô hình tùy chỉnh**

1. class Tên lớp: IModelBinder
2. {
3. tác vụ công khai BindModelAsync(ModelBindingContext bindingContext)
4. {
5. Nhận giá trị từ yêu cầu
6. bindingContext.ValueProvider.GetValue("Tên đầu tiên");
8. Trả về đối tượng mô hình sau khi đọc dữ liệu từ yêu cầu
9. bindingContext.Result = ModelBindingResult.Success(your\_object);
10. }
11. }

**Chất kết dính IModel**

* Giao diện cơ sở cho tất cả các chất kết dính mô hình tùy chỉnh.
* Cung cấp một phương thức có tên là BindModelAsync, để xác định logic để liên kết (đọc) dữ liệu từ yêu cầu và tạo một đối tượng mô hình đã được nhận dưới dạng tham số trong phương thức hành động.

**Ngữ cảnh ModelBindingContext**

* Hoạt động như một tham số cho phương thức "BindModelAsync()" của các lớp kết dính mô hình tùy chỉnh.
* Cung cấp các thuộc tính như HttpContext, ModelState, ValueProvider, Result, v.v.

Nhà cung cấp chất kết dính mô hình tùy chỉnh

1. class Tên lớp: IModelBinderProvider
2. {
3. IModelBinder công cộng GetBinder(ModelBinderProviderContext providerContext)
4. {
5. Trả về kiểu của lớp kết dính mô hình tùy chỉnh sẽ được gọi
6. trả về mới BinderTypeModelBinder(typeof(YourModelBinderClassName));
7. }
8. }

**IModelBinderProvider**

* Giao diện cơ sở cho tất cả các nhà cung cấp chất kết dính mô hình tùy chỉnh.
* Cung cấp một phương thức có tên là GetBinder, để trả về loại lớp kết dính mô hình tùy chỉnh.

**ModelBinderProviderContext**

* Hoạt động như một tham số cho phương thức "GetBinder()" của các lớp cung cấp kết dính mô hình tùy chỉnh.
* Cung cấp các thuộc tính như BindingInfo, Services, v.v.

# **6. Chế độ xem dao cạo [MVC]**

Kiến trúc MVC

Model-View-Controller (MVC) là một mẫu thiết kế tổ chức ứng dụng của bạn thành ba thành phần riêng biệt, mỗi thành phần có một trách nhiệm cụ thể:

1. **Mẫu:**
   * Đại diện cho dữ liệu và logic nghiệp vụ của ứng dụng của bạn.
   * Đóng gói quyền truy cập dữ liệu, quy tắc xác thực và mọi hoạt động kinh doanh cốt lõi.
   * **Các loại mô hình:**
     + **Mô hình kinh doanh (Mô hình miền):** Đại diện cho các thực thể và mối quan hệ trong thế giới thực của miền ứng dụng của bạn (ví dụ: Sản phẩm, Khách hàng, Đơn đặt hàng).
     + **Mô hình chế độ xem:** Một mô hình được thiết kế riêng để cung cấp dữ liệu cho chế độ xem. Nó có thể tổng hợp dữ liệu từ nhiều mô hình kinh doanh hoặc chứa các thuộc tính cho các mối quan tâm cụ thể về giao diện người dùng.
2. **Cảnh:**
   * Hiển thị giao diện người dùng (UI) dựa trên dữ liệu do bộ điều khiển cung cấp.
   * Thông thường, chế độ xem là các mẫu HTML với mã phía máy chủ được nhúng (cú pháp Razor trong ASP.NET Core) để tự động tạo nội dung.
3. **Điều khiển:**
   * Hoạt động như trung gian giữa mô hình và chế độ xem.
   * Nhận đầu vào của người dùng (từ các yêu cầu), tương tác với mô hình để thực hiện các hành động hoặc truy xuất dữ liệu, sau đó chọn chế độ xem thích hợp để trình bày kết quả.

Quy trình thực thi một yêu cầu trong MVC

1. **Định tuyến:** Yêu cầu đến được xử lý bởi phần mềm trung gian định tuyến, xác định bộ điều khiển và phương thức hành động thích hợp để xử lý nó dựa trên mẫu URL và phương thức HTTP.
2. **Liên kết mô hình:** Nếu phương thức hành động có tham số, trình kết dính mô hình sẽ trích xuất dữ liệu từ yêu cầu (chuỗi truy vấn, giá trị tuyến đường, dữ liệu biểu mẫu, nội dung) và cố gắng chuyển đổi dữ liệu đó thành các loại mà các tham số mong đợi.
3. **Xác thực mô hình:** Trình kết dính mô hình xác thực dữ liệu liên kết bằng cách sử dụng chú thích dữ liệu và bất kỳ logic xác thực tùy chỉnh nào mà bạn đã triển khai. Nếu xác thực không thành công, lỗi sẽ được thêm vào đối tượng ModelState.
4. **Thực thi hành động của bộ điều khiển:** Nếu mô hình hợp lệ, phương thức hành động của bộ điều khiển sẽ thực thi logic của nó, có thể liên quan:
   * Tương tác với mô hình kinh doanh để truy xuất hoặc cập nhật dữ liệu.
   * Thực hiện tính toán hoặc các hoạt động kinh doanh khác.
   * Chuẩn bị mô hình chế độ xem để chuyển dữ liệu đến chế độ xem.
5. **Lựa chọn chế độ xem:** Bộ điều khiển chọn chế độ xem thích hợp để hiển thị và chuyển mô hình chế độ xem đến nó.
6. **Kết xuất chế độ xem:** Công cụ xem (Razor) tạo đầu ra HTML dựa trên mẫu dạng xem và dữ liệu trong mô hình chế độ xem.
7. **Phản hồi:** HTML được hiển thị được gửi trở lại máy khách dưới dạng phản hồi HTTP.

Trách nhiệm của các thành phần MVC

* **Điều khiển:**
  + Xử lý các yêu cầu đến và định tuyến chúng đến các phương thức hành động chính xác.
  + Tương tác với mô hình để tìm nạp hoặc sửa đổi dữ liệu.
  + Chọn chế độ xem thích hợp và chuyển dữ liệu cần thiết cho nó.
  + Xử lý lỗi và chuyển hướng.
* **Mô hình (Mô hình kinh doanh):**
  + Đóng gói logic nghiệp vụ và quyền truy cập dữ liệu của ứng dụng.
  + Đại diện cho các thực thể và mối quan hệ cốt lõi trong miền của bạn.
  + Xác định các quy tắc xác thực để đảm bảo tính toàn vẹn của dữ liệu.
* **Xem mô hình:**
  + Được thiết kế riêng cho một chế độ xem cụ thể, chỉ chứa dữ liệu theo yêu cầu của chế độ xem đó.
  + Đơn giản hóa việc trình bày dữ liệu trong chế độ xem và tránh hiển thị các chi tiết không cần thiết.
* **Cảnh:**
  + Hiển thị giao diện người dùng dựa trên dữ liệu do bộ điều khiển cung cấp.
  + Sử dụng cú pháp Razor để kết hợp đánh dấu HTML với mã C# để tạo nội dung động.

Lợi ích và mục tiêu của MVC

* **Tách biệt các mối quan tâm (SoC):** Việc phân chia trách nhiệm rõ ràng làm cho mã có tổ chức, dễ bảo trì và kiểm tra hơn.
* **Khả năng kiểm tra:** Các thành phần riêng lẻ (mô hình, chế độ xem, bộ điều khiển) có thể được kiểm tra riêng biệt.
* **Khả năng tái sử dụng:** Chế độ xem và mô hình có thể được sử dụng lại trong các ngữ cảnh khác nhau.
* **Phát triển song song:** Các thành viên khác nhau trong nhóm có thể làm việc đồng thời trên các phần khác nhau của ứng dụng.
* **Khả năng bảo trì:** Các thay đổi đối với một thành phần ít có khả năng ảnh hưởng đến các thành phần khác.
* **Khả năng mở rộng:** Các tính năng mới có thể được thêm vào dễ dàng hơn do cấu trúc mô-đun.
* **Khả năng mở rộng:** Kiến trúc MVC phù hợp với việc mở rộng quy mô khi ứng dụng phát triển.

Mô hình MVC cung cấp một cách tiếp cận có cấu trúc và có tổ chức để xây dựng các ứng dụng web phức tạp, tạo điều kiện cộng tác, khả năng tái sử dụng mã và khả năng bảo trì lâu dài. Hiểu được những khái niệm cốt lõi này sẽ giúp bạn trở thành một nhà phát triển ASP.NET Core hiệu quả hơn.

Views

Trong ASP.NET Core MVC, các chế độ xem chịu trách nhiệm hiển thị giao diện người dùng (UI) mà ứng dụng của bạn trình bày cho người dùng. Chúng thường là các mẫu HTML với cú pháp Razor được nhúng, đây là một công cụ tạo mẫu mạnh mẽ cho phép bạn kết hợp liền mạch đánh dấu HTML với mã C#.

**Ghi chú**

* **Nội dung động:** Chế độ xem có thể tạo nội dung động dựa trên dữ liệu được truyền đến chúng từ bộ điều khiển (mô hình). Điều này cho phép bạn hiển thị thông tin được tìm nạp từ cơ sở dữ liệu, đầu vào của người dùng và các nguồn khác.
* **Cú pháp Razor:** Chế độ xem sử dụng cú pháp Razor, bắt đầu bằng ký hiệu @, để nhúng mã C# vào HTML. Mã này có thể thực hiện các tác vụ như lặp lại các bộ sưu tập dữ liệu, hiển thị có điều kiện và truy cập các thuộc tính mô hình.

Phương thức AddControllersWithViews()

Phương thức mở rộng này được sử dụng để đăng ký các dịch vụ cần thiết cho MVC (mô hình, chế độ xem, bộ điều khiển) với vùng chứa chèn phụ thuộc. Đó là một phím tắt để thêm nhiều dịch vụ cùng một lúc, bao gồm:

* **Khám phá bộ điều khiển:** Tự động phát hiện các lớp bộ điều khiển trong dự án của bạn.
* **Công cụ xem:** Định cấu hình Razor làm công cụ chế độ xem.
* **Liên kết mô hình:** Thiết lập liên kết mô hình để xử lý việc gửi biểu mẫu.
* **Xác thực:** Cho phép xác thực mô hình.

Kết quả xem

Lớp ViewResult là một loại kết quả hành động trong ASP.NET Core MVC đại diện cho một view sẽ được hiển thị. Khi một hành động của bộ điều khiển trả về ViewResult, khung MVC sẽ định vị tệp view tương ứng, chuyển dữ liệu mô hình (nếu có) cho nó và hiển thị nội dung của view dưới dạng HTML.

**Vị trí xem mặc định**

ASP.NET Core MVC tuân theo một quy ước để xác định nơi tìm các tệp xem:

* **Theo quy ước:** Theo mặc định, công cụ xem tìm kiếm các dạng xem trong đường dẫn Views/[ControllerName]/[ActionName].cshtml. Ví dụ: phương thức hành động Index trong HomeController sẽ tìm kiếm tệp dạng xem tại Views/Home/Index.cshtml.
* **Ghi đè bằng ViewName:** Bạn có thể chỉ định rõ ràng tên chế độ xem bằng cách sử dụng thuộc tính ViewName của ViewResult hoặc phương thức trợ giúp View().
* **Chế độ xem được chia sẻ:** Chế độ xem được chia sẻ được lưu trữ trong thư mục Chế độ xem/Chia sẻ và có thể được sử dụng bởi nhiều bộ điều khiển.

Mã

1. Program.cs
2. chủ thầu. Services.AddControllersWithViews(); Bật các tính năng MVC
4. // ...
5. HomeController.cs
6. [Tuyến đường ("nhà")]
7. Chỉ số IActionResult công khai()
8. {
9. trả về View(); Hiển thị Views/Home/Index.cshtml
10. }
12. Lượt xem/Trang chủ/Index.cshtml
13. <! DOCTYPE html>
14. <html>
15. < đầu>
16. <title>Asp.Net Core App< / title>
17. < / đầu>
18. <thân>
19. Chào mừng
20. < / cơ thể>
21. </html>
22. **Bật MVC:** Phương thức AddControllersWithViews() định cấu hình ứng dụng cho MVC, bao gồm cả hỗ trợ chế độ xem.
23. **Hành động của Bộ điều khiển:** Phương thức hành động Index trong HomeController trả về ViewResult mà không chỉ định tên chế độ xem.
24. **Vị trí chế độ xem:** Vì không có tên dạng xem nào được cung cấp rõ ràng, khung MVC tuân theo quy ước và tìm kiếm dạng xem tại Views/Home/Index.cshtml.
25. **Xem Nội dung:** Chế độ xem Index.cshtml chứa HTML đơn giản sẽ được hiển thị dưới dạng phản hồi.

Dao cạo View Công cụ

Razor là công cụ xem mặc định trong ASP.NET Core MVC. Nó cung cấp một cách ngắn gọn và thanh lịch để tạo các trang web động bằng cách kết hợp mã C# với đánh dấu HTML.

**Các yếu tố cú pháp chính của Razor**

1. **Khối mã (@{...}):**
   * Mục đích: Bao gồm các câu lệnh mã C# nhiều dòng.
   * Các trường hợp sử dụng:
     + Khai báo biến
     + Xác định logic phức tạp
     + Thực hiện truy vấn cơ sở dữ liệu hoặc các hoạt động khác
2. **Biểu thức (@variable hoặc @method()):**
   * Mục đích: Nhúng các biểu thức C# trực tiếp vào đầu ra HTML.
   * Các trường hợp sử dụng:
     + Hiển thị giá trị từ các biến hoặc thuộc tính mô hình
     + Gọi các phương thức hoặc hàm trợ giúp
3. **Nghĩa đen (@: hoặc <text>):**
   * Mục đích: Xuất văn bản thô mà không cần mã hóa HTML.
   * Các trường hợp sử dụng:
     + Hiển thị văn bản thuần túy, đoạn mã HTML hoặc giá trị có thể chứa ký tự HTML

Kiểm soát luồng trong chế độ xem dao cạo

1. **@if, @else, @elseif:**
   * Mục đích: Hiển thị có điều kiện của các khối HTML dựa trên các điều kiện C#.
   * Ví dụ:
2. @if (người. DateOfBirth.HasValue)
3. {
4. <p>Tuổi: @(Math.Round((DateTime.Now - người. DateOfBirth). Value.TotalDays / 365.25)) tuổi</p>
5. }
6. khác
7. {
8. <p>không rõ ngày sinh</p>
9. }
10. **@switch:**
    * Mục đích: Chọn một trong một số khối mã để thực thi dựa trên giá trị của biểu thức.
    * Ví dụ:
11. @switch (người. PersonGender)
12. {
13. trường hợp Giới tính.Nam:
14. <p>19 tháng 11 là Ngày Quốc tế Nam giới< / tr>
15. phá vỡ;
16. trường hợp Giới tính.Nữ:
17. <p>ngày 8 tháng 3 là Ngày Quốc tế Phụ nữ</tr>
18. phá vỡ;
19. // ... các trường hợp khác ...
20. }
21. **@foreach:**
    * Mục đích: Lặp lại một bộ sưu tập (ví dụ: danh sách hoặc mảng) và hiển thị HTML cho từng mục.
    * Ví dụ:
22. @foreach (var người trong người)
23. {
24. <div>@person. Tên, @person. PersonGender</div>
25. }
26. **@for:**
    * Mục đích: Thực thi một khối mã một số lần cụ thể.
    * Ví dụ:
27. @for (int i = 0; i < 5; i ++)
28. {
29. <p>Lặp lại: @i</p>
30. }

**Ghi chú**

* **Cú pháp Razor:** Làm quen với cú pháp Razor (khối mã, biểu thức, ký ngạch) để nhúng mã C# vào chế độ xem của bạn.
* **Luồng điều khiển:** Hiểu cách sử dụng if, else, switch, foreach và for để kiểm soát luồng logic và tạo chế độ xem động.
* **Liên kết mô hình:** Chế độ xem thường hoạt động với các mô hình được truyền từ bộ điều khiển. Sử dụng biểu thức Razor để truy cập và hiển thị các thuộc tính mô hình.

Các chức năng cục bộ trong chế độ xem dao cạo

Hàm cục bộ là các hàm C# được xác định trong phạm vi của khối mã Razor (một khối mã được bao bọc trong @{ ... }). Chúng cho phép bạn đóng gói logic có thể tái sử dụng trực tiếp trong chế độ xem của mình, làm cho mã chế độ xem của bạn trở nên mô-đun hơn, có tổ chức và dễ đọc hơn.

**Cú pháp và tính năng**

* **Khai báo:** Các hàm cục bộ được khai báo bằng cú pháp phương thức C# tiêu chuẩn, thường nằm trong một khối @functions { ... }.
* **Phạm vi:** Chúng chỉ có thể được gọi trong khối mã hoặc phần nơi chúng được xác định.
* **Tham số và kiểu trả về:** Các hàm cục bộ có thể nhận tham số và giá trị trả về, giống như các phương thức C# thông thường.
* **Khả năng tiếp cận:** Chúng ngầm riêng tư đối với chế độ xem.
* **Truy cập vào View dữ liệu:** Các hàm cục bộ có thể truy cập các biến và thuộc tính mô hình được khai báo trong cùng một khối mã.

**Tại sao nên sử dụng chức năng cục bộ?**

* **Đóng gói:** Nhóm logic liên quan thành các chức năng khép kín, cải thiện khả năng đọc mã.
* **Khả năng đọc:** Chia nhỏ mã phức tạp thành các phần nhỏ hơn, dễ quản lý hơn.
* **Khả năng tái sử dụng:** Gọi các hàm cục bộ nhiều lần trong cùng một chế độ xem, tránh trùng lặp mã.
* **Chế độ xem sạch hơn:** Giảm lượng mã C# nội tuyến nằm rải rác trong đánh dấu HTML của bạn.

Mã

1. @using ViewsExample.Models
3. @{
4. string appTitle = "Asp.Net ứng dụng demo cốt lõi";
5. Danh sách<Người> người = Danh sách mới<Người>()
6. {
7. // ... (dữ liệu người)
8. };
9. }
11. @functions {
12. đôi? GetAge(Ngày/giờ? ngày sinh)
13. {
14. nếu (dateOfBirth không phải là null)
15. {
16. trả về Math.Round((DateTime.Now - dateOfBirth.Value). Tổng số ngày / 365,25);
17. }
18. khác
19. {
20. trả về null;
21. }
22. }
24. int x = 10; Ví dụ về một biến cục bộ
26. string Name { get; set; } = "Hello name"; Ví dụ về tài sản địa phương
27. }
29. <! DOCTYPE html>
30. <html>
31. < đầu>
32. <tiêu đề>@appTitle</tiêu đề>
33. <meta charset="utf-8" />
34. < / đầu>
35. <thân>
36. <h1>Chào mừng bạn đến với @Name</h1>
37. @for (int i = 0; i < 2; i ++)
38. {
39. Người = người[i];
40. <div>
41. @person. Tên
42. < nhịp>, < / nhịp>
43. <Span>@person. PersonGender</span>
44. @if (người. DateOfBirth != null)
45. {
46. <Span>@person. DateOfBirth.Value.ToString("MM/dd/yyyy")</span>
47. <span>@GetAge(người. Ngày sinh) tuổi</nhịp>
48. }
49. < / div>
50. }
51. < / cơ thể>
52. </html>

Trong mã này:

1. **Hàm cục bộ:** GetAge(DateTime? dateOfBirth) tính tuổi của một người dựa trên ngày sinh của họ.
2. **Biến cục bộ:** int x = 10; Điều này không được sử dụng trong chế độ xem nhưng trình bày cách khai báo các biến cục bộ trong khối mã Razor.
3. **Thuộc tính cục bộ:** tên chuỗi { get; set; } = "Hello name"; Điều này được sử dụng để đặt tiêu đề của trang.

Hàm GetAge được gọi trong vòng lặp @foreach để hiển thị tuổi của mỗi người nếu ngày sinh của họ có sẵn.

**Những cân nhắc quan trọng**

* **Lạm dụng:** Tránh lạm dụng các chức năng cục bộ. Chúng rất tốt để đóng gói, nhưng quá nhiều có thể khiến quan điểm của bạn khó theo dõi hơn.
* **Các lựa chọn thay thế:** Xem xét các phương thức trợ giúp hoặc xem các thành phần để có logic phức tạp hoặc có thể tái sử dụng hơn.

Html.Raw()

Trong chế độ xem Razor, hành vi mặc định là tự động mã hóa nội dung HTML để ngăn chặn các lỗ hổng bảo mật tiềm ẩn như tấn công kịch bản chéo trang web (XSS). Tuy nhiên, đôi khi bạn có nội dung HTML hợp pháp (ví dụ: từ trình soạn thảo văn bản đa dạng thức hoặc nguồn đáng tin cậy) mà bạn muốn hiển thị nguyên trạng mà không cần mã hóa. Đây là lúc Html.Raw() xuất hiện.

**Mục đích**

* **Bỏ qua mã hóa HTML:** Html.Raw() ngăn Razor mã hóa nội dung HTML mà bạn cung cấp, cho phép nó được hiển thị trực tiếp trong trình duyệt.
* **Hiển thị HTML đáng tin cậy:** Sử dụng nó khi bạn tin tưởng nguồn của nội dung HTML và muốn giữ nguyên định dạng và cấu trúc của nó.
* **Tạo nội dung động:** Html.Raw() có thể được sử dụng để tự động chèn các đoạn HTML vào chế độ xem của bạn dựa trên dữ liệu hoặc logic.

**Cú pháp**

1. @Html.Raw(nội dung html)

trong đó htmlContent là một biến chuỗi hoặc biểu thức chứa HTML bạn muốn hiển thị mà không cần mã hóa.

**Những cân nhắc quan trọng**

* **Rủi ro bảo mật: Sử dụng Html.Raw() hết sức thận trọng.** Nếu htmlContent bắt nguồn từ đầu vào của người dùng hoặc một nguồn không đáng tin cậy, nó có thể dẫn đến các lỗ hổng XSS. Luôn làm sạch và xác thực nội dung do người dùng tạo trước khi hiển thị bằng Html.Raw().
* **Chính sách bảo mật nội dung (CSP):** Cân nhắc triển khai CSP để giảm thiểu hơn nữa rủi ro XSS. CSP xác định một tập hợp các quy tắc chi phối cách trình duyệt xử lý các tập lệnh, kiểu và các tài nguyên khác bên ngoài.

Mã

1. @{
2. // ... (các biến số và logic khác)...
3. string alertMessage = $"<script>alert('{people. Đếm} người tìm thấy')</script>";
4. }
6. <! DOCTYPE html>
7. <html>
8. < đầu>
9. < / đầu>
10. <thân>
11. @Html.Raw(thông báo cảnh báo)
12. <h1>Chào mừng</h1>
13. @for (int i = 0; i < 2; i ++)
14. {
15. // ... (phần còn lại của mã xem)...
16. }
17. < / cơ thể>
18. </html>

Trong mã này:

1. **Chuỗi HTML thô:** Biến alertMessage chứa một chuỗi HTML bao gồm thẻ <script> nhằm hiển thị thông báo cảnh báo với số lượng người được tìm thấy.
2. **Cách sử dụng Html.Raw():** Dòng @Html.Raw(alertMessage) yêu cầu Razor hiển thị nội dung của alertMessage trực tiếp vào đầu ra HTML mà không cần mã hóa. Điều này sẽ dẫn đến HTML sau trong đầu ra:
3. <script>alert('3 người đã tìm thấy')</script>
4. **Thực thi phía máy khách:** Khi trình duyệt phân tích cú pháp HTML này, nó sẽ thực thi mã JavaScript bên trong thẻ <script>, kích hoạt hộp cảnh báo.
5. **Không vệ sinh:** Trong ví dụ này, vì số lượng người không phải là đầu vào của người dùng nên không có nguy cơ bị tấn công XSS. Tuy nhiên, nếu nội dung của thông báo cảnh báo đến từ đầu vào của người dùng, điều cần thiết là phải khử thông tin trước khi hiển thị để ngăn chặn các cuộc tấn công XSS.

**Ghi chú**

* **Tin cậy nhưng xác minh:** Chỉ sử dụng Html.Raw() khi bạn hoàn toàn chắc chắn nội dung là an toàn và từ một nguồn đáng tin cậy.
* **Rủi ro bảo mật:** Nhận thức được khả năng xảy ra các lỗ hổng XSS khi hiển thị HTML thô.
* **Phương pháp tiếp cận thay thế:** Nếu bạn cần chèn nội dung HTML động, hãy xem xét các lựa chọn thay thế an toàn hơn như chế độ xem một phần hoặc thành phần chế độ xem.
* **Chính sách bảo mật nội dung (CSP):** Triển khai CSP để thêm một lớp bảo vệ bổ sung chống lại các cuộc tấn công XSS.

ViewData và ViewBag

Cả ViewData và ViewBag đều là các cơ chế trong ASP.NET Core MVC để truyền dữ liệu từ các hành động của bộ điều khiển đến các chế độ xem của bạn. Mặc dù chúng phục vụ cùng một mục đích, nhưng chúng khác nhau về cách triển khai và cú pháp.

* **ViewData (dựa trên từ điển):**
  + Đây là một đối tượng giống như từ điển (ViewDataDictionary) lưu trữ các cặp khóa-giá trị.
  + Phím là chuỗi và giá trị có thể thuộc bất kỳ loại nào.
  + Truy cập dữ liệu bằng khóa chuỗi: ViewData["keyName"].
  + Yêu cầu truyền khi truy xuất các giá trị không phải chuỗi.
* **ViewBag (Động):**
  + Đây là một trình bao bọc động xung quanh từ điển ViewData.
  + Cho phép bạn truy cập dữ liệu bằng ký hiệu chấm: ViewBag.keyName.
  + Không cần truyền rõ ràng cho các giá trị không phải chuỗi.

**Tính khả dụng trong bộ điều khiển và chế độ xem**

Cả ViewData và ViewBag đều có thể truy cập được:

* **Bộ điều khiển:** Bạn đặt các giá trị trong hành động của bộ điều khiển và chúng được chuyển đến chế độ xem.
* **Chế độ xem:** Bạn truy xuất các giá trị từ từ điển ViewData hoặc đối tượng động ViewBag trong chế độ xem để hiển thị nội dung.

**Mã của ViewData:**

1. HomeController.cs (Bộ điều khiển)
2. ViewData["appTitle"] = "Asp.Net ứng dụng demo cốt lõi";
3. ViewData["people"] = người;
5. Index.cshtml (Xem)
6. <title>@ViewData["appTitle"]</title>
7. Danh sách<Người>? people = (danh sách<người>?)ViewData["người"];

**Mã của ViewBag:**

1. HomeController.cs (Bộ điều khiển)
2. ViewBag.appTitle = "Asp.Net ứng dụng demo cốt lõi";
3. ViewBag.people = người;
4. Index.cshtml (Xem)
5. <title>@ViewBag.appTitle</title>
6. @foreach (Người trong ViewBag.people)

**Các phương pháp hay nhất**

* **Chọn một:** Thông thường, bạn nên chọn ViewData hoặc ViewBag và sử dụng nó một cách nhất quán trong suốt dự án của mình để duy trì cách tiếp cận rõ ràng và thống nhất.
* **Nhập dữ liệu mạnh mẽ với các mô hình View:** Mặc dù ViewData và ViewBag mang lại sự linh hoạt, nhưng chúng thiếu sự an toàn về kiểu trong thời gian biên dịch. Đối với các ứng dụng lớn hơn, hãy ưu tiên các mô hình chế độ xem được nhập mạnh để chuyển dữ liệu đến chế độ xem.
* **Phạm vi hạn chế:** Hiểu rằng dữ liệu ViewData và ViewBag chỉ tồn tại trong khoảng thời gian của chu kỳ yêu cầu và phản hồi hiện tại. Nó không có nghĩa là để duy trì dữ liệu giữa các yêu cầu.
* **Khóa có ý nghĩa:** Sử dụng các khóa mô tả và có ý nghĩa cho dữ liệu của bạn để cải thiện khả năng đọc mã.

**Những điều cần tránh**

* **Lạm dụng:** Không nhồi nhét quá nhiều dữ liệu vào ViewData hoặc ViewBag. Giữ cho nó ngắn gọn và tập trung vào thông tin cụ thể cần thiết cho chế độ xem.
* **Magic Strings:** Tránh mã hóa cứng các chuỗi cho các phím. Sử dụng hằng số hoặc enum để có khả năng bảo trì tốt hơn.
* **Dữ liệu nhạy cảm:** Không bao giờ chuyển dữ liệu nhạy cảm như mật khẩu hoặc mã thông báo xác thực thông qua ViewData hoặc ViewBag.

Chế độ xem được gõ mạnh

Trong ASP.NET Core MVC, một chế độ xem được gõ mạnh là một chế độ xem được liên kết với một lớp mô hình cụ thể. Điều này có nghĩa là dạng xem có quyền truy cập trực tiếp vào các thuộc tính và phương thức của mô hình đó, cung cấp kiểm tra kiểu thời gian biên dịch và hỗ trợ IntelliSense trong chế độ xem.

**Chỉ thị @model**

Chỉ thị @model được sử dụng ở đầu chế độ xem để chỉ định loại mô hình cho chế độ xem đó. Ví dụ: @model Person chỉ ra rằng view mong muốn làm việc với dữ liệu của lớp Person.

**Khi nào nên sử dụng chế độ xem gõ mạnh**

* **Dữ liệu phức tạp:** Khi chế độ xem của bạn cần hiển thị hoặc tương tác với các cấu trúc dữ liệu phức tạp có nhiều thuộc tính hoặc mối quan hệ.
* **Type Safety:** Khi bạn muốn phát hiện các lỗi liên quan đến type trong quá trình phát triển, thay vì trong thời gian chạy.
* **IntelliSense:** Khi bạn muốn toàn bộ sức mạnh của IntelliSense của Visual Studio giúp bạn viết mã nhanh hơn và ít lỗi hơn.
* **Tái cấu trúc:** Khi bạn cần thay đổi mô hình, các chế độ xem được gõ mạnh giúp tự động cập nhật các chế độ xem tương ứng dễ dàng hơn.

**Các phương pháp hay nhất**

* **Sử dụng mô hình chế độ xem:** Thay vì chuyển trực tiếp các mô hình kinh doanh của bạn đến chế độ xem, hãy tạo các mô hình chế độ xem chuyên biệt phù hợp với dữ liệu cần thiết của mỗi chế độ xem. Điều này giúp giữ cho chế độ xem của bạn sạch sẽ và tập trung.
* **Quy ước đặt tên:** Thực hiện theo quy ước đặt tên tiêu chuẩn cho các mô hình chế độ xem của bạn (ví dụ: ProductViewModel, OrderDetailsViewModel).
* **Giữ chế độ xem đơn giản:** Chế độ xem nên chủ yếu tập trung vào logic trình bày. Tránh logic nghiệp vụ phức tạp trong chế độ xem.
* **Chế độ xem một phần:** Tận dụng chế độ xem một phần cho các thành phần có thể tái sử dụng, chuyển các mô hình chế độ xem cho chúng khi cần thiết.
* **Tận dụng Trình trợ giúp thẻ:** Khám phá cách sử dụng Trình trợ giúp thẻ, cung cấp một cách thân thiện hơn với HTML để tương tác với các mô hình của bạn trong chế độ xem.

Mã:

**Bộ điều khiển (HomeController.cs)**

1. // ...
2. [Tuyến đường ("nhà")]
3. [Tuyến ("/")]
4. Chỉ số IActionResult công khai()
5. {
6. Danh sách<Người> người = Danh sách mới<Người>()
7. {
8. new Person() { Name = "John", DateOfBirth = DateTime.Parse("2000-05-06"), PersonGender = Gender.Male},
9. new Person() { Name = "Linda", DateOfBirth = DateTime.Parse("2005-01-09"), PersonGender = Gender.Female},
10. new Person() { Name = "Susan", DateOfBirth = null, PersonGender = Gender.Other}
11. };
13. return View("Index", người); Chuyển danh sách 'người' làm mô hình
14. }
16. [Tuyến đường("chi tiết người/{tên}")]
17. chi tiết IActionResult công khai(chuỗi? tên)
18. {
19. // ... (Truy xuất người dựa trên tên)...
20. Người? matchingPerson = người. Ở đâu (nhiệt độ => nhiệt độ. Tên == tên). FirstOrDefault();
21. trả về View(matchingPerson); Chuyển một đối tượng 'Person' duy nhất làm mô hình
22. }

Cả hai phương thức hành động đều chuyển (các) mô hình đến chế độ xem bằng cách sử dụng return View(model).

**Chế độ xem (Index.cshtml)**

1. @using ViewsExample.Models
2. @model IEnumerable<Person> // Cho biết mô hình là một tập hợp các đối tượng Person
4. <! DOCTYPE html>
5. <html>
6. < đầu>
7. <title>@ViewBag.appTitle</title>
8. < / đầu>
9. <thân>
10. <div class="page-content">
11. <h1>Người</h1>
13. @foreach (Người trong người mẫu)
14. {
15. <div class="hộp float-left w-50">
16. <h3>@person. Tên</h3>
17. <table class="bảng w-100">
18. <tbody>
19. <tr>
20. <td>Giới tính</td>
21. <td>@person. NgườiGiới tính< / td>
22. < / tr>
23. <tr>
24. <td>Ngày sinh</td>
25. <td>@person. Ngày sinh?. ToString("MM/dd/yyyy")</td>
26. < / tr>
27. < / tbody>
28. </bảng>
29. <a href="/person-details/@person. Tên">Chi tiết</a>
30. < / div>
31. }
32. < / div>
34. < / cơ thể>
35. </html>

Chế độ xem này lặp lại Model (là danh sách các đối tượng Person) bằng cách sử dụng vòng lặp foreach. Bên trong vòng lặp, nó truy cập trực tiếp các thuộc tính của từng đối tượng Person (ví dụ: @person. Tên, @person. PersonGender) nhờ gõ mạnh.

**Xem (Details.cshtml)**

1. @using ViewsExample.Models
2. @model Person // Cho biết mô hình là một đối tượng Person duy nhất
4. <! DOCTYPE html>
5. <html>
6. < đầu>
7. <chức danh>Chi tiết người</chức danh>
8. < / đầu>
9. <thân>
10. <div class="page-content">
11. <h1>Chi tiết người</h1>
12. <div class="box">
13. <h3>@Model.Name</h3>
14. <table class="bảng w-100">
15. <tbody>
16. <tr>
17. <td>Giới tính</td>
18. <td>@Model.PersonGender</td>
19. < / tr>
20. <tr>
21. <td>Ngày sinh</td>
22. <td>@Model.Ngày sinh?. ToString("MM/dd/yyyy")</td>
23. < / tr>
24. < / tbody>
25. </bảng>
26. < / div>
27. <a href="/home">Trở lại người</a>
28. < / div>
29. < / cơ thể>
30. </html>

Chế độ xem này hiển thị chi tiết của một đối tượng Person. Lưu ý rằng bạn có thể truy cập trực tiếp vào các thuộc tính như @Model.Name mà không cần bất kỳ giao dịch hoặc tra cứu từ điển nào.

Chế độ xem được gõ mạnh với nhiều mô hình

Khi chế độ xem của bạn yêu cầu dữ liệu từ nhiều lớp mô hình, cách tiếp cận phổ biến và được đề xuất nhất là tạo mô *hình chế độ xem*. Mô hình xem là một lớp tùy chỉnh đóng gói tất cả dữ liệu cần thiết cho một chế độ xem cụ thể, tổng hợp các thuộc tính từ nhiều mô hình hoặc cung cấp các thuộc tính bổ sung phù hợp với nhu cầu trình bày của chế độ xem.

**Tại sao nên sử dụng View Models?**

* **Đóng gói:** Giữ cho logic chế độ xem của bạn có tổ chức và ngăn các chế độ xem của bạn được kết hợp chặt chẽ với các mô hình kinh doanh cơ bản của bạn.
* **Tính linh hoạt:** Cho phép bạn kết hợp dữ liệu từ các nguồn khác nhau (nhiều mô hình, cài đặt cấu hình, v.v.) thành một đối tượng duy nhất cho chế độ xem.
* **An toàn kiểu:** Chế độ xem được gõ mạnh với các mô hình chế độ xem cung cấp kiểm tra kiểu thời gian biên dịch và IntelliSense, cải thiện hiệu quả phát triển.
* **Định hình dữ liệu:** Bạn có thể tạo các thuộc tính trong mô hình xem được thiết kế đặc biệt cho cách bạn muốn hiển thị dữ liệu trong chế độ xem, chẳng hạn như chuỗi được định dạng hoặc giá trị được tính toán.

**Tạo và sử dụng các mô hình chế độ xem**

1. **Xác định Mô hình Xem:** Tạo một lớp mới trong thư mục Mô hình của bạn để đại diện cho mô hình chế độ xem. Lớp này sẽ có các thuộc tính để lưu trữ dữ liệu từ các mô hình khác nhau mà chế độ xem của bạn cần.
2. **Điền vào Bộ điều khiển:** Trong hành động của bộ điều khiển, hãy truy xuất dữ liệu từ các mô hình khác nhau của bạn và sử dụng chúng để tạo một phiên bản của mô hình chế độ xem của bạn.
3. **Chuyển đến Chế độ xem:** Chuyển đối tượng mô hình chế độ xem đã điền vào chế độ xem bằng phương thức View.
4. **Truy cập trong chế độ xem:** Trong chế độ xem, hãy sử dụng lệnh @model ở trên cùng để chỉ định loại mô hình chế độ xem. Sau đó, bạn có thể truy cập các thuộc tính của mô hình dạng xem bằng cách sử dụng Model.<PropertyName>.

**Ví dụ về mã: PersonAndProductWrapperModel**

1. PersonAndProductWrapperModel.cs (Xem mô hình)
2. lớp công cộng PersonAndProductWrapperModel
3. {
4. public Person PersonData { get; set; }
5. public Product ProductData { get; set; }
6. }
8. HomeController.cs (Bộ điều khiển)
9. IActionResult công khai SomeAction()
10. {
11. Người = GetPersonData(); Tìm nạp dữ liệu người
12. Sản phẩm sản phẩm = GetProductData(); Tìm nạp dữ liệu sản phẩm
14. var viewModel = PersonAndProductWrapperModel mới
15. {
16. PersonData = người,
17. ProductData = sản phẩm
18. };
20. trả về View(viewModel);
21. }

**Xem (PersonAndProduct.cshtml)**

1. @using ViewsExample.Models
2. @model PersonAndProductWrapperModel
4. <! DOCTYPE html>
5. <html>
6. < đầu>
7. < / đầu>
8. <thân>
9. <div class="page-content">
10. <h1>Người và Sản phẩm</h1>
12. <div class="hộp w-100">
13. <h3>Người</h3>
14. <table class="bảng w-50">
15. <tbody>
16. <tr>
17. <td>Tên người</td>
18. <td>@Model.PersonData.Name</td>
19. < / tr>
20. <tr>
21. <td>Giới tính</td>
22. <td>@Model.PersonData.PersonGender</td>
23. < / tr>
24. < / tbody>
25. </bảng>
26. < / div>
28. <div class="hộp w-100">
29. <h3>Sản phẩm</h3>
30. <table class="bảng w-50">
31. <tbody>
32. <tr>
33. <td>ID sản phẩm</td>
34. <td>@Model.ProductData.ProductId</td>
35. < / tr>
36. <tr>
37. <td>Tên sản phẩm< / td>
38. <td>@Model.ProductData.ProductName</td>
39. < / tr>
40. < / tbody>
41. </bảng>
42. < / div>
43. < / div>
44. < / cơ thể>
45. </html>

**Những điểm chính:**

* **Tổ chức:** Các mô hình xem giúp giữ cho các chế độ xem của bạn tập trung vào bản trình bày và bộ điều khiển của bạn tập trung vào việc truy xuất và xử lý dữ liệu.
* **Khả năng bảo trì:** Khi các mô hình cơ bản của bạn thay đổi, bạn chỉ cần cập nhật mô hình chế độ xem của mình, không phải mọi chế độ xem sử dụng dữ liệu đó.
* **Tính linh hoạt:** Bạn có thể bao gồm các thuộc tính bổ sung trong mô hình chế độ xem không phải là một phần của mô hình gốc. Đây có thể là các tùy chọn định dạng, giá trị được tính toán hoặc cờ để kiểm soát hành vi của chế độ xem.

\_ViewImports.cshtml

Tệp \_ViewImports.cshtml là một tệp đặc biệt trong ASP.NET Core MVC cho phép bạn tập trung các lệnh và cài đặt phổ biến áp dụng cho nhiều chế độ xem trong ứng dụng của bạn. Tệp này thường nằm trong thư mục Views ở thư mục gốc của dự án của bạn, nhưng bạn cũng có thể đặt nó trong các thư mục con để áp dụng cài đặt cho các view trong các thư mục cụ thể đó.

**Bạn có thể đặt gì vào \_ViewImports.cshtml?**

* **@using Chỉ thị:** Nhập không gian tên mà bạn thường sử dụng trong chế độ xem của mình. Điều này giúp loại bỏ nhu cầu đưa các chỉ thị này vào mọi chế độ xem riêng lẻ.
* **Chỉ thị @addTagHelper:** Cung cấp Trình trợ giúp thẻ cho các chế độ xem của bạn. Tag Helpers là các đoạn mã phía máy chủ tạo hoặc sửa đổi các phần tử HTML theo cách trực quan hơn so với cú pháp Razor truyền thống.
* **Chỉ thị @inject:** Chèn các dịch vụ vào chế độ xem của bạn, làm cho chúng có thể truy cập được để sử dụng trong mã Razor của bạn.
* **Chỉ thị @model (Tùy chọn):** Chỉ định loại mô hình mặc định cho tất cả các chế độ xem trong thư mục hoặc thư mục con (không được khuyến nghị cho các dự án phức tạp).
* **Chỉ thị @layout (Tùy chọn):** Đặt bố cục mặc định cho tất cả các chế độ xem trong thư mục hoặc thư mục con.

**Cách thức hoạt động**

Khi ứng dụng của bạn xử lý yêu cầu chế độ xem, ứng dụng sẽ tìm kiếm tệp \_ViewImports.cshtml theo thứ tự sau:

1. **Cùng thư mục với chế độ xem:** Nếu tìm thấy, các chỉ thị và cài đặt trong tệp này sẽ được áp dụng.
2. **Thư mục mẹ:** Sau đó, nó kiểm tra thư mục cha của chế độ xem, v.v., cho đến thư mục Views gốc.
3. **Thư mục Root Views:** Cuối cùng, nó kiểm tra tệp \_ViewImports.cshtml trong thư mục Views gốc.

Chỉ thị trong tệp \_ViewImports.cshtml của thư mục con có thể ghi đè các cài đặt kế thừa từ thư mục mẹ.

**Ví dụ mã: \_ViewImports.cshtml**

1. @using System.Collections.Generic // Nhập không gian tên bộ sưu tập chung
2. @using YourProject.Models // Nhập không gian tên mô hình của dự án
3. @addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers // Thêm ASP.NET Core Tag Helpers tích hợp sẵn

**Lợi ích của việc sử dụng \_ViewImports.cshtml**

* **Giảm dự phòng:** Tránh lặp lại các chỉ thị giống nhau trong nhiều chế độ xem.
* **Cấu hình nhất quán:** Dễ dàng áp dụng các cài đặt chung trên các chế độ xem của bạn.
* **Cải thiện khả năng đọc:** Giữ cho các tệp chế độ xem cá nhân của bạn gọn gàng hơn và tập trung hơn vào nội dung cụ thể của chúng.

**Mã chế độ xem:**

1. @model YourProject.Models.Product
2. <! DOCTYPE html>
3. <html>
4. < đầu>
5. <tiêu đề>Chi tiết sản phẩm</tiêu đề>
6. < / đầu>
7. <thân>
8. <h1>@Model.Name</h1>
9. < / cơ thể>
10. </html>

Trong ví dụ này, chỉ thị @model không bắt buộc trong dạng xem vì nó đã được chỉ định trong tệp \_ViewImports.cshtml. Tương tự, bạn không cần phải bao gồm chỉ thị @using cho không gian tên của mô hình của bạn.

**Những cân nhắc quan trọng**

* **Phạm vi:** Các lệnh trong \_ViewImports.cshtml được giới hạn trong thư mục và thư mục con của nó.
* **Ghi đè:** Cài đặt trong tệp \_ViewImports.cshtml của thư mục con ghi đè các cài đặt trong thư mục mẹ.
* **Tính linh hoạt:** Sử dụng nhiều tệp \_ViewImports.cshtml trong các thư mục khác nhau để quản lý cài đặt ở cấp độ chi tiết hơn.

**Những điểm chính cần nhớ**

* **Tập trung:** Sử dụng \_ViewImports.cshtml để tập trung các lệnh và cài đặt chế độ xem chung.
* **Nguyên tắc DRY:** Tuân thủ nguyên tắc Don't Repeat Yourself (DRY) bằng cách tránh mã thừa.
* **Phạm vi và Ghi đè:** Hiểu cách hoạt động của bản chất xếp tầng của tệp \_ViewImports.cshtml.

Chế độ xem được chia sẻ

Trong ASP.NET Core MVC, chế độ xem được chia sẻ là tệp dạng xem (.cshtml) không được liên kết với một bộ điều khiển hoặc hành động cụ thể. Các chế độ xem này thường nằm trong thư mục Views/Shared và được thiết kế để sử dụng lại trên các bộ điều khiển khác nhau hoặc thậm chí nhiều dự án.

**Tại sao nên sử dụng chế độ xem được chia sẻ?**

* **Giảm trùng lặp:** Tránh viết cùng một mã lặp đi lặp lại cho các yếu tố giao diện người dùng phổ biến trên ứng dụng của bạn.
* **Giao diện người dùng nhất quán:** Duy trì giao diện thống nhất trên các trang và phần khác nhau của trang web của bạn.
* **Bảo trì đơn giản:** Cập nhật chế độ xem được chia sẻ một lần và các thay đổi sẽ tự động áp dụng cho tất cả các chế độ xem sử dụng chế độ xem đó.

**Cách hoạt động của chế độ xem được chia sẻ**

1. **Vị trí:** Theo mặc định, ASP.NET Core MVC tìm kiếm các dạng xem được chia sẻ trong thư mục Xem/Chia sẻ. Bạn cũng có thể tạo các thư mục con trong Dạng xem/Được chia sẻ để sắp xếp thêm các chế độ xem được chia sẻ của mình.
2. **Đặt tên:** Đặt tên cho các chế độ xem được chia sẻ của bạn theo cách phản ánh mục đích hoặc nội dung của chúng. Các ví dụ phổ biến bao gồm:
   * \_Layout.cshtml: Bố cục chính cho các trang ứng dụng của bạn.
   * \_PartialName.cshtml: Các thành phần nhỏ hơn, có thể tái sử dụng (dạng xem một phần) có thể được nhúng vào các dạng xem khác.
3. **Kết xuất:** Bạn có thể hiển thị chế độ xem được chia sẻ bằng cách sử dụng các phương pháp sau:
   * **PartialView():** Hiển thị chế độ xem một phần.
   * **View():** Có thể được sử dụng để hiển thị trực tiếp chế độ xem được chia sẻ, nhưng phổ biến hơn là sử dụng PartialView() cho chế độ xem một phần và đặt trước View() cho toàn bộ trang.

**Ví dụ mã: Chế độ xem được chia sẻ All.cshtml**

1. <! DOCTYPE html>
2. <html>
3. < đầu>
4. <title>Tất cả sản phẩm</title>
5. <meta charset="UTF-8" />
6. <link href="~/StyleSheet.css" rel="stylesheet" />
7. < / đầu>
8. <thân>
9. <div class="page-content">
10. <h1>Tất cả các sản phẩm< / h1>
11. < / div>
12. < / cơ thể>
13. </html>

Chế độ xem được chia sẻ này (All.cshtml) có thể được sử dụng lại bởi nhiều bộ điều khiển để hiển thị trang "Tất cả sản phẩm" với cùng cấu trúc và kiểu dáng.

**Ví dụ mã: Bộ điều khiển sử dụng chế độ xem được chia sẻ**

1. HomeController.cs
2. [Lộ trình ("trang chủ/tất cả sản phẩm")]
3. IActionResult công khai Tất cả ()
4. {
5. trả về View(); Trước tiên, sẽ tìm kiếm Views/Home/All.cshtml, sau đó là Views/Shared/All.cshtml
6. }
8. ProductsController.cs
9. [Lộ trình ("sản phẩm/tất cả")]
10. IActionResult công khai Tất cả ()
11. {
12. trả về View(); Trước tiên, sẽ tìm kiếm Views/Products/All.cshtml, sau đó là Views/Shared/All.cshtml
13. }

Trong cả HomeController và ProductsController, phương thức hành động All trả về ViewResult. Trước tiên, ASP.NET Core sẽ tìm kiếm một dạng xem có tên All.cshtml trong thư mục dạng xem của bộ điều khiển cụ thể (ví dụ: Dạng xem/Trang chủ hoặc Chế độ xem/Sản phẩm). Nếu nó không tìm thấy nó ở đó, nó sẽ tìm kiếm nó trong thư mục Xem/Chia sẻ.

**Những cân nhắc quan trọng**

* **Quy ước đặt tên:** Đặt tiền tố tên của chế độ xem một phần bằng dấu gạch dưới (ví dụ: \_ProductCard.cshtml) để phân biệt chúng với chế độ xem đầy đủ.
* **Truyền dữ liệu:** Sử dụng mô hình chế độ xem để chuyển dữ liệu đến chế độ xem được chia sẻ, đảm bảo rằng chế độ xem có quyền truy cập vào tất cả thông tin cần thiết để hiển thị chính xác.
* **Tính linh hoạt:** Bạn có thể ghi đè các chế độ xem được chia sẻ trong các bộ điều khiển cụ thể nếu bạn cần tùy chỉnh chúng cho một trường hợp sử dụng cụ thể.

Các chế độ xem được chia sẻ là nền tảng của các ứng dụng MVC cốt lõi ASP.NET có thể bảo trì và mở rộng. Bằng cách tận dụng khả năng tái sử dụng của chúng, bạn có thể hợp lý hóa đáng kể quy trình phát triển của mình và đảm bảo trải nghiệm người dùng nhất quán trên trang web hoặc ứng dụng của bạn.

Những điểm chính cần nhớ

**1. Lượt xem**

* **Mục đích:** Hiển thị giao diện người dùng (UI) của ứng dụng MVC của bạn.
* **Razor View Engine:** Công cụ mặc định để kết hợp đánh dấu HTML với mã C#.
* **Vị trí tệp xem:** Thường được tìm thấy trong thư mục Chế độ xem, được sắp xếp theo bộ điều khiển.
* **View Selection:** Bộ điều khiển trả về ViewResult hoặc các kết quả hành động khác để hiển thị chế độ xem.
* **Liên kết mô hình:** Chế độ xem thường nhận dữ liệu từ bộ điều khiển (mô hình) để hiển thị động.

**2. Yếu tố cần thiết về cú pháp dao cạo**

* **Khối mã (@{...}):** Đối với mã C# nhiều dòng.
* **Biểu thức (@variable hoặc @method()):** Nhúng biểu thức C#.
* **Literals (@: hoặc <text>):** Đầu ra văn bản thô (không mã hóa HTML).
* **Luồng điều khiển:** Sử dụng @if, @else, @switch, @foreach, @for cho logic có điều kiện và lặp lại.
* **Bình luận:** @\*...\*@ cho nhận xét phía máy chủ.

**3. Chức năng cục bộ**

* **Mục đích:** Đóng gói các chức năng C# có thể tái sử dụng trong chế độ xem Razor.
* **Cú pháp:** Định nghĩa trong @functions { ... }.
* **Lợi ích:** Cải thiện tổ chức, khả năng đọc và khả năng tái sử dụng trong chế độ xem.

**4. Html.Raw ()**

* **Mục đích:** Hiển thị nội dung HTML thô mà không cần mã hóa.
* **Cảnh báo:** Sử dụng thận trọng do các lỗ hổng XSS tiềm ẩn.
* **Các lựa chọn thay thế:** Xem xét các chế độ xem một phần hoặc các thành phần chế độ xem để tạo HTML động.

**5. ViewData và ViewBag**

* **Mục đích:** Chuyển dữ liệu từ bộ điều khiển đến chế độ xem.
* **ViewData:** Dựa trên từ điển (ViewData["key"]).
* **ViewBag: Trình**  bao bọc động xung quanh ViewData (ViewBag.key).
* **Khuyến nghị:** Ưu tiên các chế độ xem hoặc mô hình dạng xem được gõ mạnh để có tính an toàn và khả năng bảo trì kiểu tốt hơn.

**6. Chế độ xem gõ mạnh**

* **Mục đích:** Liên kết chế độ xem với một lớp mô hình cụ thể bằng lệnh @model.
* **Lợi ích:** An toàn kiểu loại, IntelliSense và cải thiện khả năng bảo trì.
* **Mô hình xem:** Kết hợp dữ liệu từ nhiều mô hình thành một lớp mô hình chế độ xem duy nhất.

**7. \_ViewImports.cshtml**

* **Mục đích:** Tập trung các @using chung, @addTagHelper, @inject và các chỉ thị khác.
* **Vị trí:** Trong thư mục gốc Views hoặc các thư mục con.
* **Lợi ích:** Giảm dư thừa, đảm bảo tính nhất quán và cải thiện khả năng đọc.

**8. Lượt xem được chia sẻ**

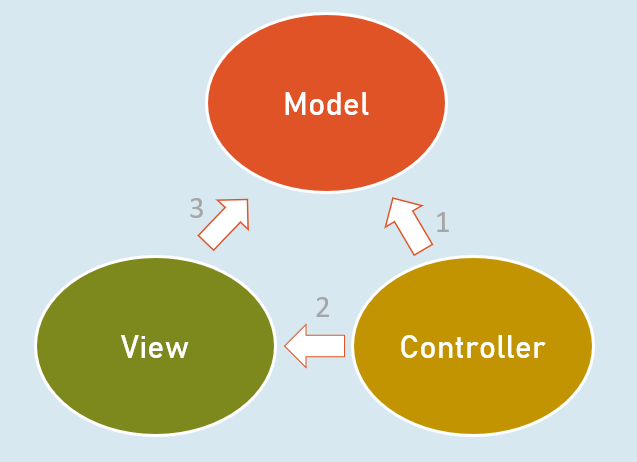
* **Mục đích:** Các thành phần chế độ xem có thể tái sử dụng được lưu trữ trong thư mục Xem/Chia sẻ.
* **Loại:**
  + **Trang bố cục:** Xác định cấu trúc trang tổng thể.
  + **Chế độ xem một phần:** Các phần tử giao diện người dùng có thể tái sử dụng.
* **Kết xuất:** Sử dụng PartialView() để hiển thị chế độ xem một phần, View() cho chế độ xem đầy đủ.

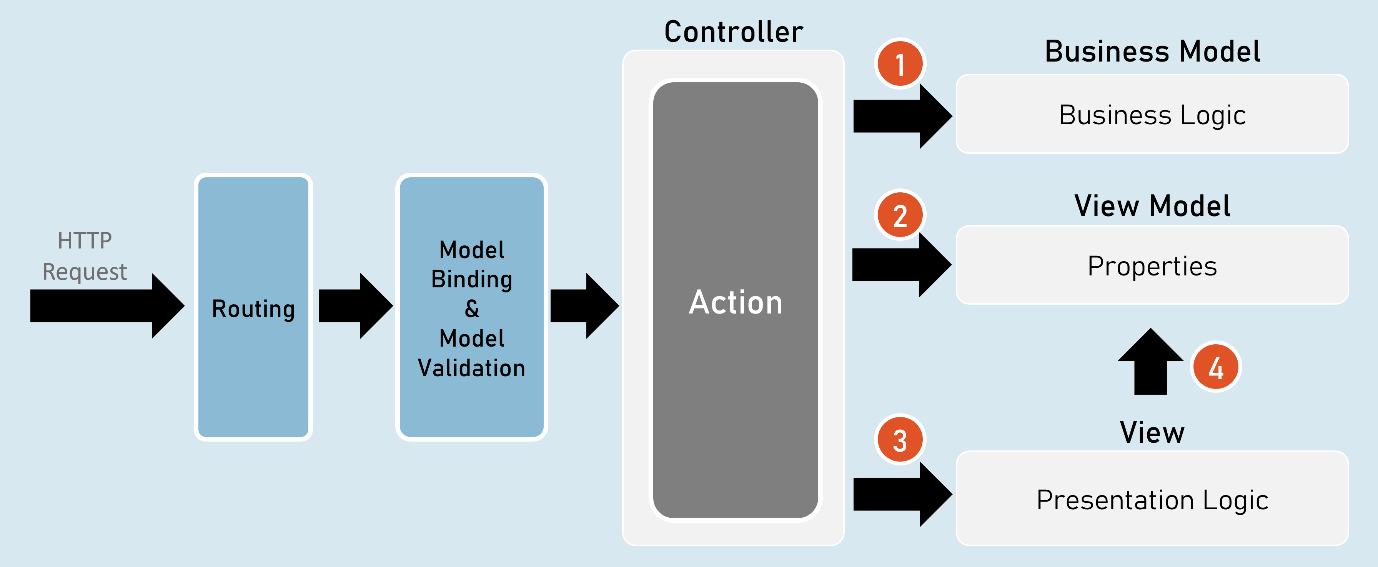
**Mẹo phỏng vấn:**

* **Cú pháp Razor:** Hãy sẵn sàng viết ví dụ về các yếu tố cú pháp Razor khác nhau và giải thích các trường hợp sử dụng của chúng.
* **View Logic:** Thảo luận về thời điểm thích hợp để sử dụng các hàm cục bộ hoặc view mô hình trong view của bạn.
* **Bảo mật:** Nhấn mạnh tầm quan trọng của bảo mật (xác thực đầu vào, tránh Html.Raw() cho nội dung không đáng tin cậy) khi làm việc với chế độ xem.
* **Các phương pháp hay nhất:** Thể hiện kiến thức của bạn về việc sử dụng mô hình chế độ xem, dạng xem được chia sẻ và \_ViewImports.cshtml để có mã rõ ràng hơn và dễ bảo trì hơn.

Mô hình - Chế độ xem - Bộ điều khiển (MVC)

"Model-View-Controller" (MVC) là một mẫu kiến trúc tách mã ứng dụng thành ba thành phần chính: Models, Views và Controllers.





1. Bộ điều khiển gọi Mô hình kinh doanh.
2. Bộ điều khiển tạo đối tượng của View Model.
3. Bộ điều khiển gọi View.
4. Chế độ xem truy cập vào Mô hình xem.

Trách nhiệm của Model-View-Controller

**Điều khiển**

* Nhận dữ liệu yêu cầu HTTP.
* Gọi mô hình kinh doanh để thực hiện logic nghiệp vụ.

**Mô hình kinh doanh**

* Nhận dữ liệu đầu vào từ bộ điều khiển.
* Thực hiện các hoạt động kinh doanh như truy xuất/chèn dữ liệu từ cơ sở dữ liệu.
* Gửi dữ liệu của cơ sở dữ liệu trở lại bộ điều khiển.

**Điều khiển**

* Tạo đối tượng của ViewModel và lưu dữ liệu vào các thuộc tính của nó.
* Chọn một view & gọi nó & cũng chuyển đối tượng của ViewModel đến view.

**Cảnh**

* Nhận đối tượng của ViewModel từ bộ điều khiển.
* Truy cập vào các thuộc tính của ViewModel để hiển thị dữ liệu trong mã html.
* Sau khi view hiển thị, kết quả view được hiển thị sẽ được gửi dưới dạng phản hồi.

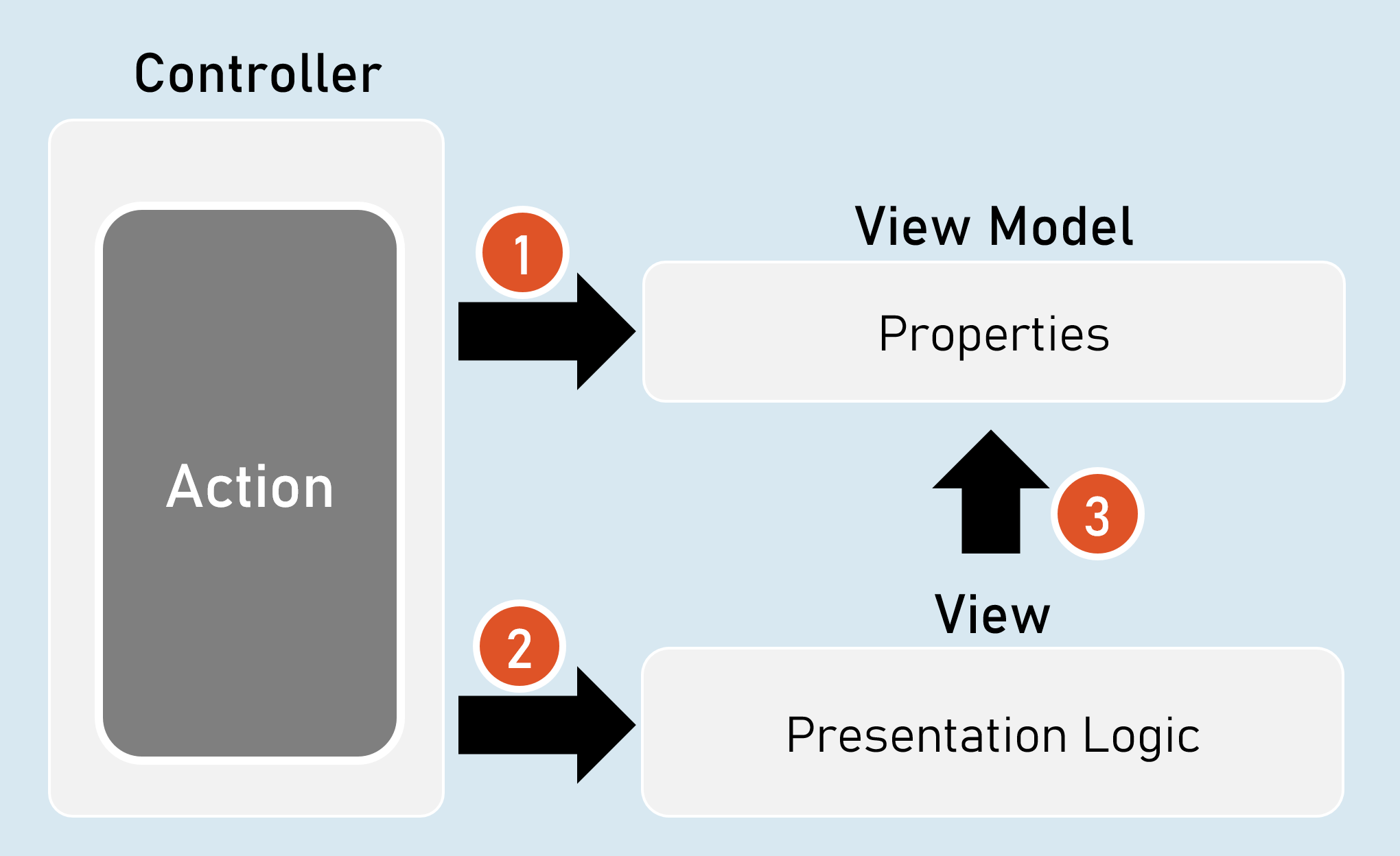
Lợi ích / Mục tiêu của mô hình kiến trúc MVC

* Tách rõ ràng các mối quan tâm
* Mỗi thành phần (mô hình, chế độ xem và bộ điều khiển) thực hiện một trách nhiệm duy nhất.
* Việc xác định và sửa lỗi sẽ dễ dàng.
* Mỗi thành phần (mô hình, chế độ xem và bộ điều khiển) có thể được phát triển độc lập.
* Trên thực tế, cả chế độ xem và bộ điều khiển đều phụ thuộc vào mô hình.
* Mô hình không phụ thuộc vào chế độ xem cũng như bộ điều khiển.
* Đây là một trong những lợi ích chính của 'tách sạch'.
* Sự tách biệt này cho phép mô hình được xây dựng và thử nghiệm độc lập.
* Kiểm tra đơn vị từng thành phần riêng lẻ dễ dàng hơn.

Views

Chế độ xem là một trang web (.cshtml) chịu trách nhiệm chứa logic trình bày hợp nhất dữ liệu cùng với mã thiết kế tĩnh (HTML).

* Bộ điều khiển tạo một đối tượng của ViewModel và điền dữ liệu vào các thuộc tính của nó.
* Controller chọn một view thích hợp và gọi cùng một view & cung cấp đối tượng của ViewModel cho View.
* Chế độ xem truy cập vào ViewModel.



* Chế độ xem chứa đánh dấu HTML với đánh dấu Razor (mã C# trong chế độ xem để hiển thị nội dung động).
* Razor là công cụ xem xác định cú pháp để viết mã C# trong chế độ xem. @ là cú pháp của cú pháp Razor.
* View KHÔNG được cho là có nhiều mã C#. Bất kỳ mã nào được viết trong chế độ xem phải liên quan đến việc trình bày nội dung (logic trình bày).
* View không nên gọi trực tiếp mô hình kinh doanh, cũng như không nên gọi các phương thức hành động của bộ điều khiển. Nhưng nó có thể gửi yêu cầu đến bộ điều khiển.

Dao cạo View Công cụ

**Khối mã dao cạo**

1. @{
3. Mã C# / html ở đây
5. }

Khối mã Razor là một khối mã C# chứa một hoặc nhiều dòng mã C# có thể chứa bất kỳ câu lệnh và hàm cục bộ nào.

**Biểu thức dao cạo**

1. @Expression
2. --hoặc--
3. @(Biểu thức)

Biểu thức dao cạo là một biểu thức C# (truy cập một trường hoặc lệnh gọi phương thức) trả về một giá trị.

**Dao cạo - Nếu**

1. @if (điều kiện) {
2. Mã C# / html ở đây
3. }

**Dao cạo - nếu... khác**

1. @if (điều kiện) {
2. Mã C# / html ở đây
3. }
4. khác {
5. Mã C# / html ở đây
6. }

Khác... if và nested-if cũng được hỗ trợ.

**Dao cạo - Công tắc**

1. @switch (biến) {
2. case value1: mã C# / html ở đây; phá vỡ;
3. case value2: mã C# / html ở đây; phá vỡ;
4. mặc định: mã C# / html ở đây; phá vỡ;
5. }

**Dao cạo - foreach**

1. @foreach (biến var trong bộ sưu tập) {
2. Mã C# / html ở đây
3. }

**Dao cạo - cho**

1. @for (khởi tạo; điều kiện; lặp lại) {
2. Mã C# / html ở đây
3. }

**Razor - Theo nghĩa đen**

1. @{
2. @: văn bản tĩnh
3. }

**Razor - Theo nghĩa đen**

<văn bản>văn bản tĩnh</văn bản>

**Razor - Chức năng cục bộ**

1. @{
2. return\_type method\_name(đối số) {
3. Mã C# / html ở đây
4. }
5. }

Các hàm cục bộ có thể gọi được trong cùng một chế độ xem.

Razor - Thành viên

**Razor - Phương pháp, Thuộc tính, Trường**

1. @functions {
2. return\_type method\_name(đối số) {
3. Mã C# / html ở đây
4. }
6. data\_type field\_name;
8. data\_type property\_name
9. {
10. đặt { ... }
11. nhận { ... }
12. }
13. }

Các thành viên của chế độ xem razor có thể được truy cập trong cùng một chế độ xem.

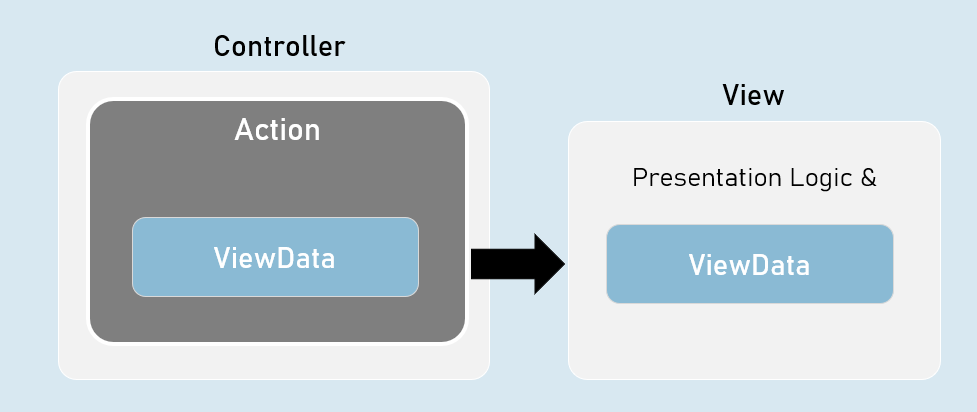
**Html.Raw( )**

1. @{
2. biến chuỗi = "mã html";
3. }
5. @Html.Raw(variable) // in đánh dấu html mà không cần mã hóa (chuyển đổi thẻ html thành văn bản thuần túy)

Dữ liệu xem

ViewData là một đối tượng từ điển được tạo tự động khi nhận được yêu cầu và sẽ tự động bị xóa trước khi gửi phản hồi đến máy khách.

Nó chủ yếu được sử dụng để gửi dữ liệu từ bộ điều khiển đến view.



ViewData là một thuộc tính của lớp Microsoft.AspNetCore.Mvc.Controller và lớp Microsoft.AspNetCore.Mvc.Razor.RazorPage.

Nó thuộc loại Microsoft.AspNet.Mvc.ViewFeatures.ViewDataDictionary.

1. namespace Microsoft.AspNetCore.Mvc
2. {
3. lớp trừu tượng công cộng Controller : ControllerBase
4. {
5. public ViewDataDictionary ViewData { get; set; }
6. }
7. }

* Nó có nguồn gốc từ IDictionary<KeyValuePair<string, object>> type.
* Điều đó có nghĩa là, nó hoạt động như một từ điển của các cặp khóa/giá trị.
* Khóa thuộc loại chuỗi.
* Giá trị thuộc loại đối tượng.

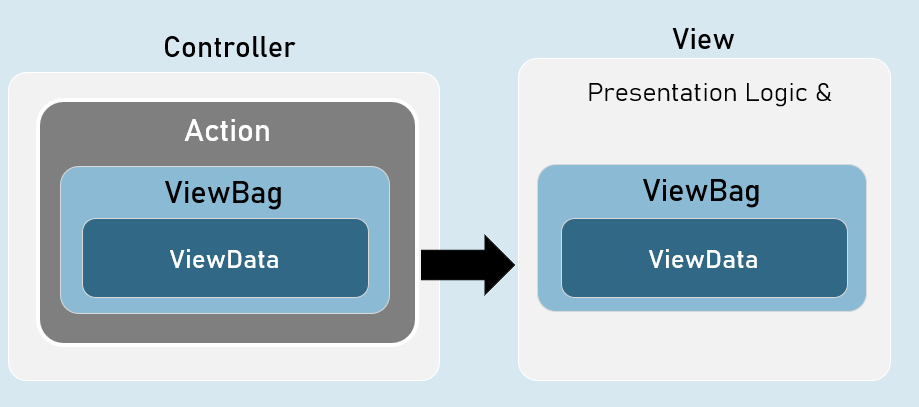
ViewData - Thuộc tính và phương thức

* int Count { get; set; } // lấy số phần tử.
* [string Key] // Nhận hoặc đặt một phần tử.
* Add(string key, object value) // Thêm một phần tử mới.
* ContainsKey(string key) // Xác định xem khóa được chỉ định có tồn tại hay không.
* Clear() // Xóa (xóa) tất cả các phần tử.

ViewTúi

ViewBag là một thuộc tính của Controller và View, được sử dụng để truy cập ViewData một cách dễ dàng.

ViewBag là loại 'động'.



ViewBag là một thuộc tính của lớp Microsoft.AspNetCore.Mvc.Controller và lớp Microsoft.AspNetCore.Mvc.Razor.RazorPageBase.

Nó thuộc loại động.

1. namespace Microsoft.AspNetCore.Mvc
2. {
3. lớp trừu tượng công cộng Controller : ControllerBase
4. {
5. ViewBag động công cộng { get; set; }
6. }
7. }

Loại 'động' tương tự như từ khóa 'var'.

Tuy nhiên, nó kiểm tra kiểu dữ liệu và tại thời gian chạy, thay vì tại thời điểm biên dịch.

Nếu bạn cố gắng truy cập vào một thuộc tính không tồn tại trong ViewBag, tài sản đó sẽ trả về null.

[string Key] // Nhận hoặc đặt một phần tử.

Lợi ích của 'ViewBag' so với ViewData

Cú pháp của ViewBag dễ truy cập các thuộc tính của nó hơn ViewData.

Ví dụ: ViewBag.property [vs] ViewData["key"]

Bạn KHÔNG cần phải loại các giá trị trong khi đọc nó.

Ví dụ: ViewBag.object\_name.property

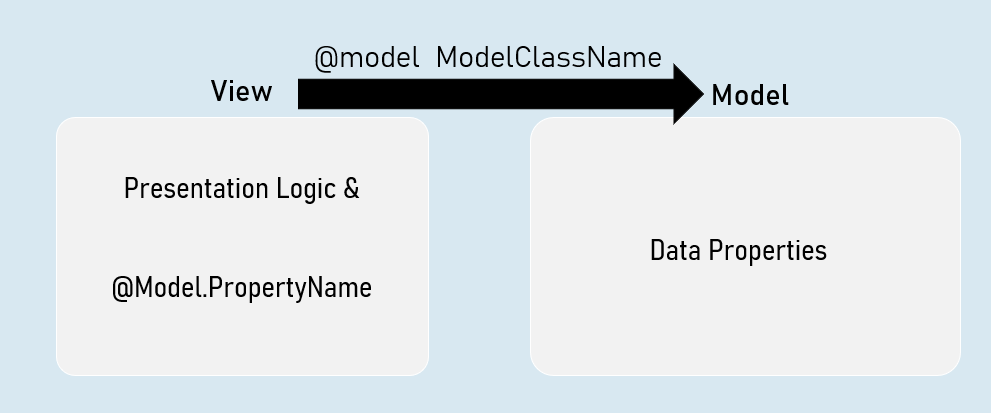
[so sánh]

(ViewData["key"] dưới dạng ClassName). Tài sản

Chế độ xem được gõ mạnh

Chế độ xem kiểu mạnh là một chế độ xem được liên kết với một lớp mô hình được chỉ định.

Nó chủ yếu được sử dụng để truy cập đối tượng mô hình / bộ sưu tập mô hình một cách dễ dàng trong chế độ xem.



Lợi ích của chế độ xem gõ mạnh

* Bạn sẽ nhận được Intellisense trong khi truy cập các thuộc tính mô hình trong các chế độ xem được gõ mạnh, vì loại của lớp mô hình đã được đề cập ở @model lệnh.
* Tên thuộc tính được kiểm tra theo thời gian biên dịch; và được hiển thị dưới dạng lỗi trong trường hợp thuộc tính sai chính tả / không tồn tại trong các chế độ xem được gõ mạnh.
* Bạn sẽ chỉ có một mô hình cho mỗi chế độ xem trong chế độ xem được gõ mạnh.
* Dễ dàng xác định mô hình nào đang được truy cập trong chế độ xem.

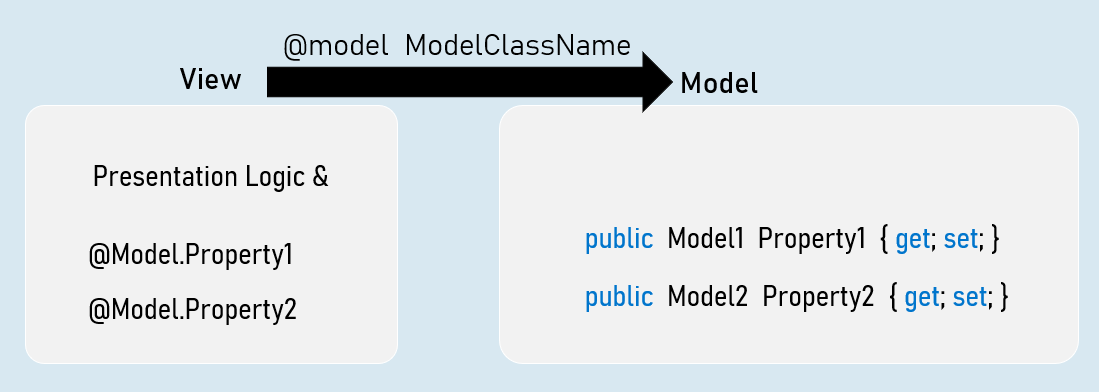
Các phương thức trợ giúp trong Controller để gọi View

* trả lại View( ); Tên dạng xem có cùng tên với phương thức hành động hiện tại.
* trả lại View(mô hình đối tượng); View name có cùng tên với phương thức hành động hiện tại và view có thể là một view được gõ mạnh để nhận đối tượng mô hình được cung cấp.
* trả về View(chuỗi ViewName); Tên chế độ xem được chỉ định rõ ràng.
* return View(chuỗi ViewName, đối tượng Model ); Tên chế độ xem được chỉ định rõ ràng và chế độ xem có thể là một chế độ xem được gõ mạnh để nhận đối tượng mô hình được cung cấp.

Chế độ xem được gõ mạnh

Chế độ xem gõ mạnh có thể được liên kết trực tiếp với một mô hình duy nhất.

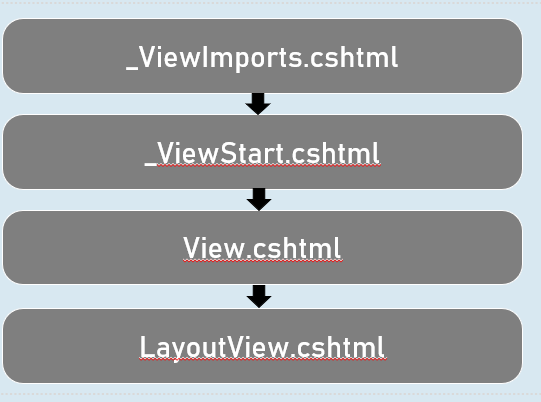
Nhưng lớp mô hình đó có thể có tham chiếu đến các đối tượng của các lớp mô hình khác.

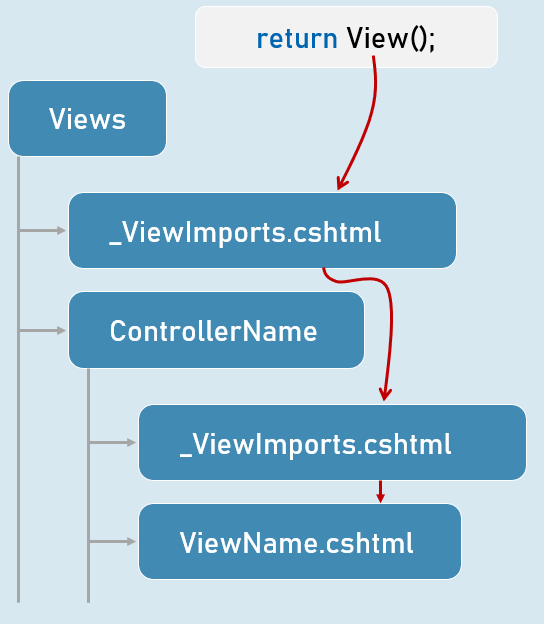


Xem Imports.cshtml

ViewImports.cshtml là một tệp đặc biệt trong thư mục "Views" hoặc thư mục con của nó, sẽ tự động thực thi trước khi thực thi chế độ xem.

Nó chủ yếu được sử dụng để nhập các không gian tên phổ biến được nhập trong một chế độ xem.





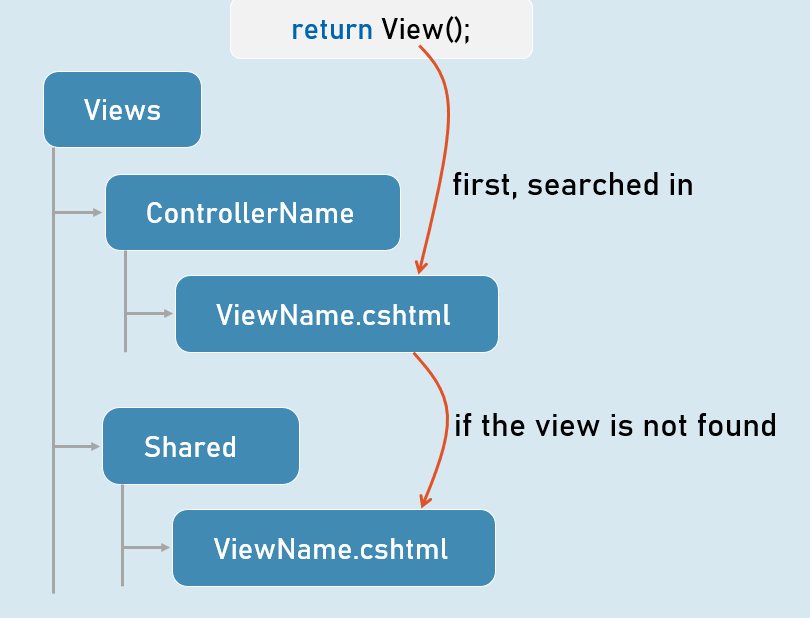
Chế độ xem được chia sẻ

Chế độ xem được chia sẻ được đặt trong thư mục "Chia sẻ" trong thư mục "Chế độ xem".

Chúng có thể truy cập từ bất kỳ bộ điều khiển nào, nếu chế độ xem KHÔNG có trong thư mục "Views\ControllerName".



Độ phân giải xem



# **7. Chế độ xem bố cục [MVC]**

Chế độ xem bố cục

Trong ASP.NET Core MVC, dạng xem bố cục là một mẫu chính xác định cấu trúc và các yếu tố chung mà bạn muốn chia sẻ trên nhiều trang trong ứng dụng của mình. Điều này có thể bao gồm tiêu đề, thanh điều hướng, chân trang, thanh bên và các thành phần giao diện người dùng phổ biến khác.

**Tại sao nên sử dụng chế độ xem bố cục?**

* **Khả năng tái sử dụng:** Tránh lặp lại mã HTML, CSS và JavaScript phổ biến trong mọi chế độ xem.
* **Tính nhất quán:** Duy trì giao diện đồng nhất trên toàn bộ ứng dụng của bạn.
* **Bảo trì đơn giản:** Thực hiện cập nhật cho chế độ xem bố cục một lần và các thay đổi sẽ tự động áp dụng cho tất cả các chế độ xem sử dụng nó.
* **Tổ chức được cải thiện:** Tách cấu trúc của các trang của bạn khỏi nội dung cụ thể của chúng.

**Cách chế độ xem bố cục hoạt động**

1. **Tệp \_Layout.cshtml:** Chế độ xem bố cục chính thường được đặt tên là \_Layout.cshtml và được đặt trong thư mục Xem/Chia sẻ. Bạn cũng có thể có nhiều tệp bố cục nếu cần.
2. **Phương thức RenderBody():** Chế độ xem bố cục chứa một phương thức Razor đặc biệt có tên là RenderBody(). Đây là một trình giữ chỗ nơi nội dung của chế độ xem cụ thể (ví dụ: Index.cshtml, About.cshtml) sẽ được chèn.
3. **Chỉ định bố cục:** Trong các chế độ xem riêng lẻ của mình, bạn sử dụng thuộc tính Bố cục để chỉ ra chế độ xem bố cục nào sẽ sử dụng. Điều này có thể được thực hiện theo hai cách:
   * **Ngầm (Dựa trên quy ước):** Nếu bạn không chỉ định bố cục một cách rõ ràng, ASP.NET Core MVC sẽ tự động sử dụng tệp \_Layout.cshtml trong thư mục Xem/Chia sẻ.
   * **Rõ ràng:** Sử dụng thuộc tính Layout ở đầu chế độ xem:
     1. @{
     2. Layout = "~/Views/Shared/\_Layout.cshtml";
     3. }

**Ví dụ về mã**

**\_Layout.cshtml (Chế độ xem bố cục)**

1. <! DOCTYPE html>
2. <html>
3. < đầu>
4. <title>@ViewData["Title"]</title>
5. <link href="~/StyleSheet.css" rel="stylesheet" />
6. < / đầu>
7. <thân>
8. <div class="container">
9. <div class="navbar">
10. <div class="navbar-brand">AspNet Core App</div>
11. <ul>
12. <li><a href="/">Home</a></li>
13. < / ul>
14. < / div>
16. <div class="page-content">
17. @RenderBody()
18. < / div>
19. < / div>
20. < / cơ thể>
21. </html>

* **@ViewData["Tiêu đề"]:** Tự động đặt tiêu đề của trang. Giá trị này sẽ được cung cấp bởi các chế độ xem riêng lẻ.
* **RenderBody():** Trình giữ chỗ này sẽ được thay thế bằng nội dung của chế độ xem cụ thể đang được hiển thị.

**Index.cshtml (Chế độ xem nội dung)**

1. @{
2. Layout = "~/Views/Shared/\_Layout.cshtml"; Chỉ định rõ ràng bố cục
3. ViewData["Tiêu đề"] = "Trang chủ";
4. }
6. <h1>Trang chủ</h1>
7. <p>Chào mừng bạn đến với trang chủ</tr>

* **Thuộc tính bố cục:** Đặt bố cục cho chế độ xem này.
* **ViewData["Title"]:** Đặt giá trị cho thuộc tính Tiêu đề, giá trị này sẽ được sử dụng trong chế độ xem bố cục.

**About.cshtml (Chế độ xem nội dung)**

1. @{
2. Layout = "~/Views/Shared/\_Layout.cshtml";
3. ViewData["Tiêu đề"] = "Giới thiệu";
4. }
6. <h1>Giới thiệu</h1>
7. <p>Về công ty</p>

Chế độ xem này tuân theo cấu trúc tương tự, đặt tiêu đề thành "Giới thiệu".

**Bộ điều khiển (HomeController.cs)**

Các hành động của bộ điều khiển chỉ đơn giản là trả về kết quả View(), điều này kích hoạt quá trình hiển thị chế độ xem. Vì bố cục không được xác định trong hành động, nên \_Layout.cshtml mặc định được sử dụng.

**Ghi chú**

* **Cấu trúc và tính nhất quán:** Chế độ xem bố cục giúp bạn thiết lập cấu trúc trang nhất quán trong toàn bộ ứng dụng của mình.
* **Nguyên tắc DRY:** Đừng lặp lại chính mình (DRY) bằng cách tránh trùng lặp các yếu tố phổ biến trong quan điểm của bạn.
* **RenderBody():** Phương thức này rất cần thiết để chèn nội dung của chế độ xem cụ thể vào bố cục.
* **Thuộc tính bố cục:** Sử dụng nó để chỉ định hoặc ghi đè bố cục cho một chế độ xem cụ thể.
* **Chế độ xem một phần:** Bạn có thể sử dụng chế độ xem một phần trong chế độ xem bố cục để chia nhỏ thêm giao diện người dùng của mình thành các thành phần có thể tái sử dụng.
* **Sections:** Phương thức RenderSection cho phép bạn xác định trình giữ chỗ trong chế độ xem bố cục và để các chế độ xem riêng lẻ cung cấp nội dung cho chúng.

\_ViewStart.cshtml

Tệp \_ViewStart.cshtml là một tệp đặc biệt trong ASP.NET Core MVC cho phép bạn định cấu hình các cài đặt chung áp dụng cho tất cả các chế độ xem trong một thư mục cụ thể và các thư mục con của nó. Tệp này thường nằm trong cùng một thư mục với chế độ xem của bạn, nhưng hiệu ứng của nó sẽ giảm xuống các thư mục con.

**Bạn có thể đặt gì vào \_ViewStart.cshtml?**

Mục đích chính của \_ViewStart.cshtml là chỉ định bố cục mặc định cho dạng xem của bạn. Bạn cũng có thể sử dụng nó để đặt các thuộc tính phổ biến khác cho chế độ xem của mình, nhưng sử dụng nó cho bố cục là phương pháp phổ biến nhất.

* **Thuộc tính bố cục:** Đặt bố cục mặc định cho tất cả các chế độ xem trong thư mục và thư mục con của nó.
  1. @{
  2. Bố cục = "\_Layout"; Đặt bố cục mặc định thành \_Layout.cshtml
  3. }

Bạn có thể chỉ định đường dẫn đầy đủ đến chế độ xem bố cục nếu cần.

* **Các thuộc tính khác:** Mặc dù ít phổ biến hơn, nhưng bạn có thể đặt các thuộc tính khác của đối tượng View trong \_ViewStart.cshtml. Tuy nhiên, bạn thường nên giữ mục đích chính của tệp này để xác định bố cục mặc định.

**Cách thức hoạt động**

Khi ASP.NET Core MVC xử lý yêu cầu chế độ xem, nó sẽ tìm kiếm tệp \_ViewStart.cshtml theo thứ tự sau:

1. **Cùng thư mục với chế độ xem:** Nếu tìm thấy, bố cục được chỉ định trong tệp này sẽ được áp dụng cho chế độ xem.
2. **Thư mục mẹ:** Sau đó, nó kiểm tra các thư mục cha của chế độ xem, đệ quy, cho đến khi nó đến thư mục Views gốc.
3. **Thư mục Chế độ xem gốc:** Cuối cùng, nó kiểm tra tệp \_ViewStart.cshtml trong thư mục Chế độ xem gốc. Nếu bố cục được chỉ định ở đó, bố cục đó sẽ được sử dụng làm mặc định cho bất kỳ chế độ xem nào chưa có bố cục được đặt rõ ràng.

**Ví dụ mã:**

Hãy tưởng tượng bạn có cấu trúc tệp sau:

1. Views/
2. \_ViewStart.cshtml (Bố cục = "\_Layout")
3. Nhà/
4. Chỉ mục.cshtml
5. Giới thiệu.cshtml
6. Sản phẩm/
7. \_ViewStart.cshtml (Bố cục = "\_ProductLayout")
8. Chỉ mục.cshtml
9. Chi tiết.cshtml

* Các dạng xem Home/Index.cshtml và Home/About.cshtml sẽ sử dụng \_Layout.cshtml làm bố cục của chúng (kế thừa từ \_ViewStart.cshtml của thư mục Views gốc).
* Các dạng xem Products/Index.cshtml và Products/Details.cshtml sẽ sử dụng \_ProductLayout.cshtml làm bố cục của chúng (được chỉ định trong \_ViewStart.cshtml của thư mục riêng của chúng).

**Lợi ích của việc sử dụng \_ViewStart.cshtml**

* **DRY (Don't Repeat Yourself):** Tránh lặp lại thông số kỹ thuật bố cục trong mọi view file.
* **Cấu hình tập trung:** Giúp bạn dễ dàng thay đổi bố cục mặc định cho toàn bộ phần của chế độ xem.
* **Khả năng bảo trì:** Đơn giản hóa việc cập nhật bố cục ứng dụng của bạn mà không cần sửa đổi các chế độ xem riêng lẻ.

**Những cân nhắc quan trọng**

* **Ghi đè bằng Thuộc tính bố cục:** Bạn vẫn có thể ghi đè bố cục mặc định trong các chế độ xem riêng lẻ bằng cách đặt thuộc tính Bố cục một cách rõ ràng.
* **Hành vi xếp tầng:** Hiểu hành vi xếp tầng khi bạn có nhiều tệp \_ViewStart.cshtml trong các thư mục khác nhau.
* **Phạm vi:** Lưu ý rằng \_ViewStart.cshtml ảnh hưởng đến các dạng xem trong cùng một thư mục và thư mục con của nó.

Chế độ xem bố cục động

Mặc dù tệp \_ViewStart.cshtml rất tuyệt vời để thiết lập bố cục mặc định, nhưng đôi khi bạn cần linh hoạt hơn để chọn các bố cục khác nhau dựa trên các điều kiện hoặc logic cụ thể trong chế độ xem của mình. Đây là lúc lựa chọn bố cục động phát huy tác dụng.

**Thuộc tính bố cục: Chìa khóa cho sự linh hoạt của bạn**

Thuộc tính Bố cục, mà bạn thường đặt trong tệp \_ViewStart.cshtml, cũng có thể được đặt trực tiếp trong các dạng xem riêng lẻ. Điều này cho phép bạn tự động thay đổi bố cục được sử dụng để hiển thị một chế độ xem cụ thể dựa trên dữ liệu hoặc ngữ cảnh của yêu cầu.

**Cách áp dụng bố cục động**

1. **Logic có điều kiện:** Sử dụng câu lệnh có điều kiện (if, else, switch) trong chế độ xem Razor của bạn để xác định bố cục nào sẽ áp dụng dựa trên các tiêu chí cụ thể.
2. **Đặt thuộc tính bố cục:** Trong các khối có điều kiện, gán đường dẫn bố cục thích hợp cho thuộc tính Bố cục.
3. **Render the View:** Framework MVC sẽ sử dụng bố cục được gán động để hiển thị trang cuối cùng.

**Ví dụ về mã: Bố cục động dựa trên ViewBag.ProductID**

1. Lượt xem/Sản phẩm/Tìm kiếm.cshtml
3. @{
4. ViewData["title"] = "Tìm kiếm sản phẩm";
5. nếu (ViewBag.ProductID != null)
6. {
7. Bố cục = "~/Views/Shared/\_ProductsLayout.cshtml";
8. }
9. }
11. <h1>Tìm kiếm sản phẩm</h1>
12. <P>tìm kiếm chi tiết tại đây</p>

**Lời giải thích**

1. **Kiểm tra điều kiện:** Trước tiên, mã sẽ kiểm tra xem ViewBag.ProductID có phải là rỗng hay không. Điều này cho thấy rằng chế độ xem đang được hiển thị trong ngữ cảnh của một sản phẩm cụ thể (vì ID sản phẩm có sẵn).
2. **Đặt bố cục:** Nếu ViewBag.ProductID không rỗng, thuộc tính Bố cục được đặt thành ~/Views/Shared/\_ProductsLayout.cshtml. Điều này có nghĩa là dạng xem sẽ sử dụng bố cục khác với bố cục mặc định có khả năng được chỉ định trong \_ViewStart.cshtml.
3. **Bố cục mặc định:** Nếu ViewBag.ProductID là rỗng, chế độ xem sẽ quay trở lại bố cục mặc định, được chỉ định trong tệp \_ViewStart.cshtml hoặc cấu hình chung của ứng dụng.

**Cân nhắc bổ sung**

* **Tính linh hoạt:** Bạn có thể sử dụng bất kỳ điều kiện nào bạn cần để xác định bố cục phù hợp. Điều này có thể liên quan đến việc kiểm tra thuộc tính mô hình, vai trò người dùng, cài đặt cấu hình hoặc các giá trị động khác.
* **Hiệu suất:** Nếu bạn có logic lựa chọn bố cục phức tạp, hãy cân nhắc di chuyển nó sang bộ điều khiển hoặc phương pháp trợ giúp để giữ cho chế độ xem của bạn sạch sẽ.
* **Khả năng bảo trì:** Sắp xếp bố cục của bạn rõ ràng và ghi lại các điều kiện mà mỗi bố cục được sử dụng.

**Những cân nhắc quan trọng**

* **Mô hình chế độ xem:** Cách ưu tiên để chuyển dữ liệu từ bộ điều khiển đến chế độ xem là sử dụng mô hình chế độ xem được gõ mạnh. ViewBag không an toàn về loại và có thể dẫn đến lỗi thời gian chạy nếu không được sử dụng cẩn thận.
* **Xử lý lỗi:** Xử lý trường hợp ViewBag.ProductID có thể không được đặt chính xác hoặc có thể là null khi bạn mong đợi nó có giá trị. Điều này có thể giúp tránh các lỗi thời gian chạy không mong muốn.

**Ví dụ cách sử dụng trong Bộ điều khiển (ProductsController.cs)**

1. Truy tìm kết quả IActioncông khai(int? productId)
2. {
3. // ... (truy xuất kết quả tìm kiếm và có khả năng đặt ViewBag.ProductID)...
5. ViewBag.ProductID = productId;
6. trả về View();
7. }

Bằng cách tự động áp dụng các bố cục khác nhau, bạn có thể tạo ra trải nghiệm người dùng phù hợp và hấp dẫn hơn, điều chỉnh cấu trúc và cách trình bày của chế độ xem để phù hợp với ngữ cảnh cụ thể của từng yêu cầu. Hãy cho tôi biết nếu bạn có bất kỳ câu hỏi nào khác!

Phần Chế độ xem bố cục

Trong ASP.NET Core MVC, các phần chế độ xem bố cục cho phép bạn xác định các phần giữ chỗ hoặc khu vực cụ thể trong chế độ xem bố cục của bạn, nơi nội dung từ các chế độ xem riêng lẻ có thể được chèn vào. Điều này cung cấp một cách mạnh mẽ để kiểm soát vị trí và tổ chức nội dung trong cấu trúc tổng thể của trang của bạn.

**Các khái niệm chính**

* **@RenderSection() trong bố cục:**
  + **Mục đích:** Xác định trình giữ chỗ (phần) trong chế độ xem bố cục nơi nội dung có thể được chèn từ các chế độ xem cụ thể.
  + **Cú pháp:**
    1. @RenderSection("sectionName", bắt buộc: true/false)
    2. sectionName: Tên duy nhất cho mục.
    3. required: Một cờ boolean cho biết phần đó có bắt buộc hay không (mặc định là true). Nếu đúng, các chế độ xem sử dụng bố cục này phải cung cấp nội dung cho phần này; nếu không, một ngoại lệ sẽ được đưa ra.
* **@section trong chế độ xem nội dung:**
  + **Mục đích:** Cung cấp nội dung được chèn vào phần tương ứng trong chế độ xem bố cục.
  + **Cú pháp:**
    1. @section sectionName {
    2. ... }

**Ví dụ về mã**

**\_Layout.cshtml (Chế độ xem bố cục)**

1. <div class="footer-content">
2. @RenderSection("footer\_section", false) // Phần chân trang tùy chọn
3. < / div>

**@RenderSection("footer\_section", false):** Dòng này xác định một phần có tên footer\_section. Đối số sai chỉ ra rằng phần này là *tùy chọn*. Chế độ xem sử dụng bố cục này có thể chọn có cung cấp nội dung cho bố cục hay không.

**Views/Home/Contact.cshtml (Chế độ xem nội dung)**

1. @section footer\_section
2. {
3. <p>Liên hệ hỗ trợ: 98348734873984734< / p>
4. }

**@section footer\_section { ... }:** Khối mã này xác định nội dung sẽ được hiển thị trong footer\_section của chế độ xem bố cục.

**Cách thức hoạt động**

1. Khi ASP.NET Core hiển thị một chế độ xem sử dụng bố cục \_Layout.cshtml, nó gặp phải dòng @RenderSection("footer\_section", false).
2. Nếu dạng xem nội dung (ví dụ: Contact.cshtml) cung cấp một khối @section footer\_section, nội dung đó sẽ được chèn vào bố cục tại vị trí @RenderSection.
3. Nếu chế độ xem nội dung không cung cấp khối @section footer\_section và phần được đánh dấu là tùy chọn (sai), thì không có nội dung nào được hiển thị trong phần đó. Nếu phần này là bắt buộc (true), một lỗi sẽ được đưa ra.

**Ghi chú**

* **Tính linh hoạt:** Các phần cho phép bạn tự động kiểm soát vị trí của nội dung trong bố cục.
* **Khả năng tái sử dụng:** Bạn có thể sử dụng lại cùng một bố cục với các chế độ xem nội dung khác nhau, mỗi chế độ xem cung cấp nội dung phần riêng.
* **Tổ chức:** Giữ cho mã bố cục của bạn sạch sẽ và tập trung bằng cách chia nhỏ trang thành các phần logic.
* **Tùy chọn so với Bắt buộc:** Sử dụng cờ bắt buộc trong @RenderSection một cách khôn ngoan để kiểm soát xem có phải cung cấp nội dung cho một phần hay không.
* **Nội dung mặc định:** Bạn có thể cung cấp nội dung mặc định trong khối @RenderSection trong chế độ xem bố cục, nội dung này sẽ được hiển thị nếu chế độ xem nội dung không ghi đè nội dung đó.

Chế độ xem bố cục lồng nhau

Trong ASP.NET Core MVC, các chế độ xem bố cục lồng nhau cho phép bạn tạo bố cục phân cấp trong đó một chế độ xem bố cục kế thừa từ một chế độ xem bố cục khác. Điều này tạo ra một cấu trúc trong đó bố cục con có thể xác định nội dung bổ sung hoặc ghi đè các phần cụ thể của bố cục mẹ của nó, trong khi vẫn kế thừa cấu trúc tổng thể và các yếu tố chung.

**Tại sao nên sử dụng bố cục lồng nhau?**

* **Khả năng tái sử dụng nâng cao:** Đạt được khả năng tái sử dụng cao hơn nữa bằng cách tạo nhiều lớp chế độ xem bố cục. Điều này đặc biệt hữu ích cho các ứng dụng lớn với các phần hoặc mô-đun riêng biệt chia sẻ một số yếu tố chung nhưng cũng có các yêu cầu bố cục riêng.
* **Điều khiển chi tiết:** Tùy chỉnh các khu vực cụ thể của bố cục mà không sao chép các đoạn mã lớn. Bạn có thể tạo bố cục cơ sở cho toàn bộ ứng dụng và sau đó có bố cục chuyên biệt cho các phần khác nhau, chẳng hạn như đầu trang, điều hướng hoặc chân trang.
* **Khả năng bảo trì:** Quản lý và cập nhật các yếu tố bố cục hiệu quả hơn bằng cách thực hiện các thay đổi ở cấp độ thích hợp trong hệ thống phân cấp bố cục.

**Triển khai bố cục lồng nhau**

1. **Bố cục gốc:** Tạo chế độ xem bố cục cha xác định cấu trúc chính và các yếu tố chung. Thông thường, đây là tệp \_Layout.cshtml của bạn trong thư mục Xem/Chia sẻ.
2. **Bố cục con:** Tạo chế độ xem bố cục con kế thừa từ bố cục mẹ. Trong bố cục con, bạn có thể:
   * Thêm các phần hoặc nội dung mới.
   * Ghi đè các phần được xác định trong bố cục cha bằng cách sử dụng @section.
   * Đặt thuộc tính Layout thành đường dẫn của bố cục mẹ.
3. **Chế độ xem nội dung:** Trong chế độ xem nội dung, bạn không cần phải chỉ định rõ ràng bố cục con. Nó sẽ tự động kế thừa từ bố cục cha nếu bạn không đặt thuộc tính Bố cục.

**Ví dụ mã: Chuyên sâu**

**\_MasterLayout.cshtml (Bố cục mẹ)**

1. <! DOCTYPE html>
2. <html>
3. < đầu>
4. < / đầu>
5. <thân>
6. <div class="container">
7. <div class="tiêu đề">
8. Ứng dụng Demo ASP.NET Core
9. < / div>
10. < / div>
11. <div>
12. @RenderBody()
13. < / div>
14. < / cơ thể>
15. </html>

**RenderBody():** Trình giữ chỗ chính nơi nội dung của bố cục con sẽ được chèn vào.

**\_Layout.cshtml (Bố cục con)**

1. @{
2. Bố cục = "~/Views/Shared/\_MasterLayout.cshtml"; Kế thừa từ \_MasterLayout.cshtml
3. }
5. <! DOCTYPE html>
6. <html>
7. < đầu>
8. < / đầu>
9. <thân>
10. <div class="container">
11. <div class="navbar">
12. < / div>
14. <div class="page-content">
15. @RenderBody()
16. < / div>
18. <div class="footer-content">
19. @RenderSection("footer\_section", sai)
20. < / div>
21. < / div>
22. < / cơ thể>
23. </html>

* **Layout = "~/Views/Shared/\_MasterLayout.cshtml":** Chỉ định bố cục mẹ.
* **Nội dung bổ sung:** Bố cục con thêm thanh điều hướng, khu vực nội dung trang và phần chân trang.
* **RenderBody():** Kế thừa trình giữ chỗ này từ bố cục mẹ, vì vậy nội dung của chế độ xem nội dung cuối cùng sẽ được hiển thị ở đây.

**\_ViewStart.cshtml**

1. @{
2. Layout = "~/Views/Shared/\_Layout.cshtml";
3. }

Thao tác này đặt bố cục mặc định cho tất cả các dạng xem trong ứng dụng thành \_Layout.cshtml.

**Cách thức hoạt động của lồng**

1. Khi một chế độ xem được hiển thị, trước tiên ASP.NET Core sẽ tìm kiếm tệp \_ViewStart.cshtml.
2. \_ViewStart.cshtml chỉ định \_Layout.cshtml làm bố cục mặc định.
3. Đến lượt mình, \_Layout.cshtml chỉ định \_MasterLayout.cshtml làm cha mẹ của nó.
4. Chế độ xem nội dung được hiển thị trong phần @RenderBody() của \_Layout.cshtml.
5. Nội dung kết hợp của \_Layout.cshtml sau đó được hiển thị trong phần @RenderBody() của \_MasterLayout.cshtml.

**Ghi chú**

* **Tính linh hoạt:** Bố cục lồng nhau mang lại sự linh hoạt để tạo cấu trúc giao diện người dùng phức tạp và mô-đun.
* **Kế thừa:** Bố cục con kế thừa cấu trúc và nội dung của bố cục mẹ của chúng.
* **Ghi đè:** Bố cục con có thể ghi đè hoặc mở rộng các phần được xác định trong bố cục mẹ.
* **Khả năng bảo trì:** Giúp cập nhật và quản lý bố cục tổng thể của ứng dụng của bạn dễ dàng hơn.

Những điểm chính cần nhớ

**Chế độ xem bố cục: Làm chủ cấu trúc trang**

* **Mục đích:** Xác định một mẫu chung cho cấu trúc và các yếu tố được chia sẻ của các trang web của bạn.
* **Lợi ích:**
  + **Khả năng tái sử dụng:** Tránh trùng lặp mã.
  + **Tính nhất quán:** Duy trì giao diện đồng nhất trên các trang.
  + **Khả năng bảo trì:** Cập nhật bố cục một lần và các thay đổi được áp dụng ở mọi nơi.
* **\_Layout.cshtml:** Tệp bố cục chính, thường được đặt trong thư mục Xem/Chia sẻ.
* **RenderBody():** Trình giữ chỗ trong chế độ xem bố cục nơi chèn nội dung của chế độ xem cụ thể.
* **Thuộc tính bố cục:**
  + Được sử dụng trong chế độ xem nội dung để chỉ định bố cục nào sẽ sử dụng.
  + Có thể được đặt động dựa trên các điều kiện trong chế độ xem.
  + Nếu không đặt, \_Layout.cshtml mặc định sẽ được sử dụng.

**Phần**

* **Mục đích:** Xác định trình giữ chỗ trong bố cục cho nội dung từ các chế độ xem cụ thể.
* **@RenderSection("sectionName", bắt buộc: true/false):** Khai báo một phần trong bố cục.
* **@section sectionName { ... }:** Cung cấp nội dung cho phần trong dạng xem nội dung.
* **Tùy chọn so với Bắt buộc:** Kiểm soát xem một phần là bắt buộc hay tùy chọn bằng cách sử dụng cờ bắt buộc.

**Bố cục lồng nhau**

* **Mục đích:** Tạo bố cục phân cấp trong đó một bố cục kế thừa từ bố cục khác.
* **Lợi ích:** Tăng khả năng tái sử dụng và kiểm soát chi tiết đối với tùy chỉnh bố cục.
* **Thực hiện:**
  + Đặt thuộc tính Layout trong bố cục con trỏ đến bố cục mẹ.
  + Sử dụng @RenderBody() trong cả hai bố cục.

**\_ViewStart.cshtml**

* **Mục đích:** Đặt bố cục mặc định cho tất cả các chế độ xem trong thư mục và thư mục con của nó.
* **Vị trí:** Thường được đặt trong thư mục Chế độ xem hoặc trong các thư mục con cụ thể.
* **Thuộc tính bố cục:** Được sử dụng trong \_ViewStart.cshtml để chỉ định tệp bố cục mặc định.

**Mẹo bổ sung**

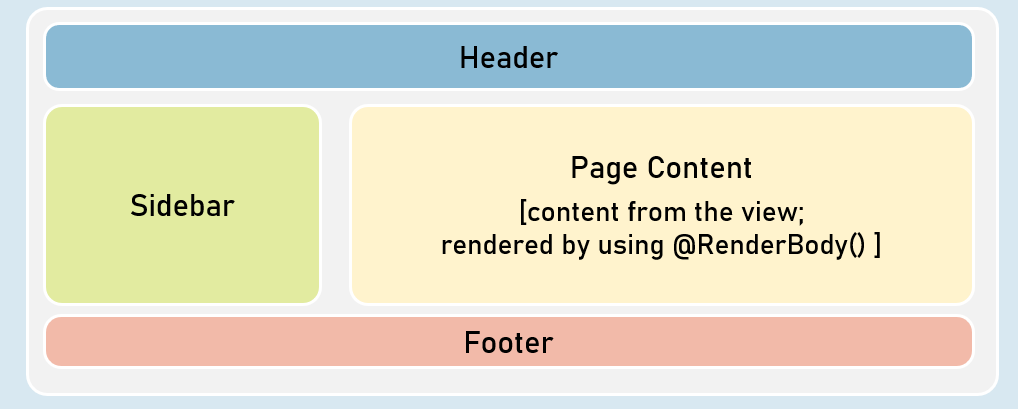
* **Mô hình chế độ xem:** Chuyển dữ liệu đến chế độ xem bố cục bằng cách sử dụng các mô hình chế độ xem được nhập mạnh.
* **Chế độ xem một phần:** Chia nhỏ các bố cục phức tạp thành các chế độ xem từng phần nhỏ hơn, có thể tái sử dụng.
* **Phần để linh hoạt:** Sử dụng các phần để làm cho bố cục của bạn có thể thích ứng với các chế độ xem nội dung khác nhau.
* **Xử lý lỗi:** Xử lý các lỗi tiềm ẩn (ví dụ: các phần bị thiếu) một cách duyên dáng.

**Mẹo phỏng vấn**

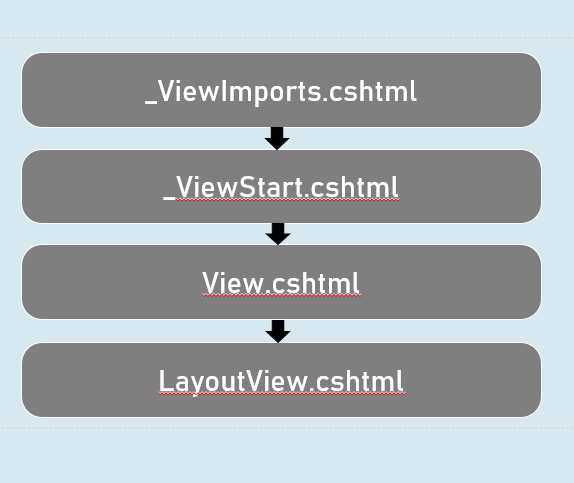
* **Giải thích hệ thống phân cấp:** Nêu rõ cách bố cục lồng nhau và \_ViewStart.cshtml hoạt động cùng nhau để xác định cấu trúc trang.
* **Ví dụ về mã:** Hãy chuẩn bị để viết các đoạn mã minh họa việc sử dụng chế độ xem bố cục, phần và lựa chọn bố cục động.
* **Các phương pháp hay nhất:** Thảo luận về cách sử dụng chế độ xem bố cục để tạo mã có thể bảo trì và tái sử dụng.
* **Khắc phục sự cố:** Giải thích cách bạn gỡ lỗi các vấn đề liên quan đến chế độ xem bố cục (ví dụ: hiển thị bố cục không chính xác).

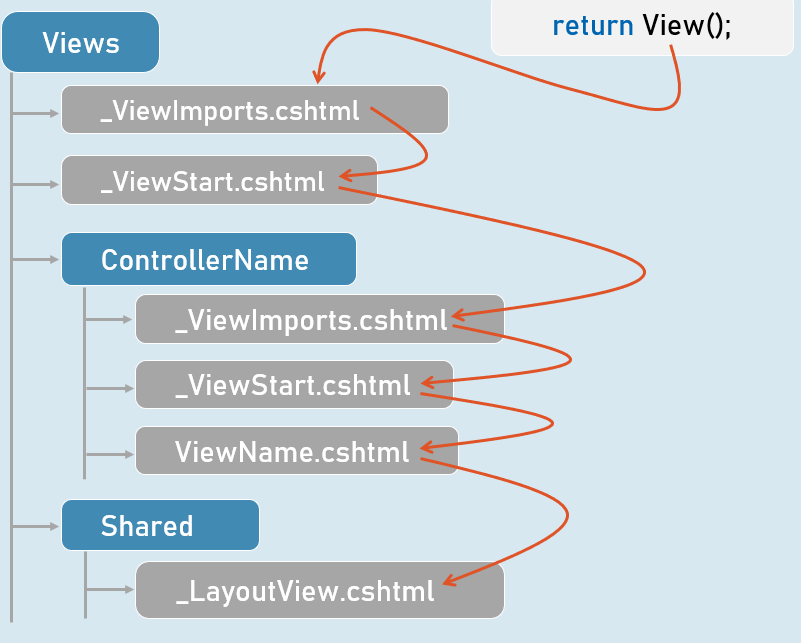
Chế độ xem bố cục

Layout View là một trang web (.cshtml) chịu trách nhiệm chứa mẫu logic trình bày (thường là mẫu html có đầu trang, thanh bên, chân trang, v.v.)



Thứ tự thực hiện Views





Chế độ xem bố cục

* Phương thức @RenderBody() chỉ hiển thị trong chế độ xem bố cục để biểu thị vị trí mà chính xác nội dung từ chế độ xem phải được hiển thị.
* Thuộc tính "Bố cục" của chế độ xem chỉ định đường dẫn của chế độ xem bố cục.
* Nó có thể được đặt động trong chế độ xem.
* Cả Chế độ xem và Chế độ xem bố cục đều chia sẻ cùng một đối tượng ViewData.
* Vì vậy, có thể gửi dữ liệu từ view sang layout, vì view thực thi trước.
* Các tệp css / tệp js được nhập trong chế độ xem bố cục cũng sẽ được áp dụng để xem, vì nội dung của chế độ xem sẽ được hợp nhất vào chế độ xem bố cục tại thời gian chạy.

# **8. Chế độ xem một phần [MVC]**

Chế độ xem một phần

Trong ASP.NET Core MVC, chế độ xem một phần là một phần đánh dấu Razor (.cshtml) có thể được nhúng trong các chế độ xem khác. Chúng được thiết kế để đóng gói các phần tử giao diện người dùng cụ thể, chẳng hạn như danh sách, biểu mẫu hoặc tiện ích, cho phép bạn tránh lặp lại mã và duy trì cấu trúc có tổ chức hơn.

**Tại sao nên sử dụng chế độ xem từng phần?**

* **Khả năng tái sử dụng:** Một chế độ xem một phần duy nhất có thể được sử dụng trong nhiều chế độ xem, tiết kiệm thời gian phát triển và giảm khả năng không nhất quán.
* **Tính mô-đun:** Chế độ xem một phần giúp chia nhỏ các chế độ xem phức tạp thành các thành phần nhỏ hơn, dễ quản lý hơn, giúp mã của bạn dễ đọc và bảo trì hơn.
* **Nội dung động:** Bạn có thể chuyển dữ liệu đến chế độ xem một phần, làm cho chúng động và có thể thích ứng với các ngữ cảnh khác nhau.

**Ghi chú**

* **Quy ước đặt tên:** Chế độ xem một phần thường được đặt tên bằng dấu gạch dưới hàng đầu (ví dụ: \_ListPartialView.cshtml). Quy ước này giúp phân biệt chúng với các quan điểm đầy đủ.
* **Vị trí:** Theo mặc định, dạng xem một phần nằm trong thư mục Dạng xem/Chia sẻ hoặc trong cùng thư mục với dạng xem sử dụng chúng. Bạn có thể ghi đè vị trí mặc định bằng cách chỉ định đường dẫn đầy đủ.
* **Kết xuất:** Bạn có thể hiển thị chế độ xem một phần trong chế độ xem chính của mình bằng cách:
  + **Trình trợ giúp thẻ:** <partial name="\_ListPartialView" /> hoặc <partial name="\_ListPartialView" model="yourModel" />
  + **Trình trợ giúp HTML:** @await Html.PartialAsync("\_ListPartialView") hoặc @await Html.PartialAsync("\_ListPartialView", yourModel)

**Ví dụ về mã**

**\_ListPartialView.cshtml (Chế độ xem một phần)**

1. <div class="list-container">
2. @{
3. ViewBag.ListTitle = "Đã cập nhật"; Không khuyến khích
4. }
6. <h3>@ViewBag.Danh sáchTiêu đề</h3>
7. <ul class="list">
8. @foreach (mục chuỗi trong ViewBag.ListItems)
9. {
10. <li>@item< / li>
11. }
12. < / ul>
13. < / div>

**ViewBag:** Ví dụ này sử dụng ViewBag để chuyển dữ liệu ListTitle và ListItems đến chế độ xem một phần. Tuy nhiên, thông thường bạn nên sử dụng mô hình được gõ mạnh để chuyển dữ liệu đến các dạng xem một phần để có tính an toàn và bảo trì kiểu tốt hơn.

**Views/Home/Index.cshtml (Chế độ xem chính)**

1. <h1>Trang chủ</h1>
3. <tên một phần ="\_ListPartialView" />
5. @{
6. var myViewData = mới ViewDataDictionary(ViewData); Tạo ViewDataDictionary mới
7. myViewData["ListTitle"] = "Quốc gia";
8. myViewData["ListItems"] = new List<string>() { "Hoa Kỳ", "Canada", "Nhật Bản", "Đức", "Ấn Độ" };
9. }
10. <div class="box">
11. <partial name="\_ListPartialView" view-data="myViewData" />
12. < / div>
14. <h3>ListTitle trong chế độ xem: @ViewData["ListTitle"]</h3>

Chế độ xem chính này hiển thị \_ListPartialView hai lần:

1. **Kết xuất đầu tiên:**
   * Giá trị @ViewBag.ListTitle trong chế độ xem một phần sẽ là null (vì nó không được đặt rõ ràng trước khi chế độ xem một phần được hiển thị).
   * @ViewBag.ListItems cũng sẽ là null và vòng lặp sẽ không thực thi.
   * Một ViewDataDictionary mới có tên myViewData được tạo.
2. **Kết xuất thứ hai:**
   * Từ điển myViewData được sử dụng để chuyển ListTitle ("Quốc gia") và ListItems (danh sách các quốc gia) đến dạng xem một phần.
   * Chế độ xem một phần sẽ hiển thị danh sách các quốc gia vì ViewBag.ListItems không phải là rỗng.
   * Sau khi chế độ xem một phần được hiển thị, ViewData["ListTitle"] vẫn tham chiếu đến giá trị ban đầu ("Asp.Net Core Demo App") được đặt trong bộ điều khiển. Điều này là do từ điển myViewData là một phiên bản riêng biệt của ViewDataDictionary.

**Views/Home/About.cshtml (Chế độ xem chính)**

1. <h1>Giới thiệu</h1>
3. <h2>Giới thiệu về công ty tại đây</h2>
4. <p>Lorem Ipsum chỉ đơn giản là văn bản giả... < / tr>
5. @{
6. chờ đợi Html.RenderPartialAsync("\_ListPartialView");
7. }

Chế độ xem này cũng hiển thị \_ListPartialView bằng phương thức Html.RenderPartialAsync. Kết quả giống như kết xuất đầu tiên trong Index.cshtml.

**Ghi chú**

* **Khả năng tái sử dụng:** Chế độ xem một phần là điều cần thiết cho sự phát triển DRY (Đừng lặp lại bản thân) trong chế độ xem của bạn.
* **Tính linh hoạt:** Bạn có thể chuyển dữ liệu đến chế độ xem một phần để làm cho chúng động.
* **Tùy chọn hiển thị:** Sử dụng trình trợ giúp thẻ hoặc trình trợ giúp HTML để hiển thị chế độ xem từng phần.
* **Chia sẻ dữ liệu:** Đảm bảo rằng chế độ xem một phần của bạn có quyền truy cập vào dữ liệu cần thiết bằng cách sử dụng các mô hình chế độ xem.
* **Tổ chức:** Chế độ xem một phần góp phần tạo nên một cơ sở mã có cấu trúc tốt và có thể bảo trì.
* **Khả năng tái sử dụng chế độ xem:** Cùng một chế độ xem một phần có thể được hiển thị nhiều lần trên cùng một trang, mỗi lần với một tập hợp dữ liệu khác nhau.

Chế độ xem một phần được gõ mạnh

Chế độ xem một phần được gõ mạnh, giống như chế độ xem được gõ mạnh, được liên kết với một lớp mô hình cụ thể bằng cách sử dụng chỉ thị @model. Điều này có nghĩa là dạng xem một phần có quyền truy cập trực tiếp vào các thuộc tính và phương thức của mô hình đó, cung cấp kiểm tra kiểu thời gian biên dịch và hỗ trợ IntelliSense trong dạng xem một phần.

**Lợi ích của chế độ xem từng phần được gõ mạnh**

* **An toàn loại:** Nắm bắt lỗi trong quá trình phát triển do kiểu dữ liệu hoặc tên thuộc tính không khớp.
* **IntelliSense:** Cung cấp tự động hoàn thành và đề xuất cho các thuộc tính mô hình, dẫn đến mã hóa nhanh hơn và chính xác hơn.
* **Tái cấu trúc:** Giúp cập nhật chế độ xem một phần dễ dàng hơn khi mô hình của bạn thay đổi.

**Cách sử dụng chế độ xem một phần được gõ mạnh**

1. **Tạo lớp mô hình:** Xác định một lớp đại diện cho cấu trúc dữ liệu bạn muốn chuyển đến chế độ xem một phần của mình.
2. **Sử dụng @model trong chế độ xem một phần:** Thêm lệnh @model ở đầu chế độ xem một phần, chỉ định lớp mô hình: @model YourNamespace.YourModel.
3. **Chuyển Mô hình:** Khi hiển thị chế độ xem một phần từ chế độ xem chính của bạn, hãy chuyển một thực thể của lớp mô hình của bạn bằng thuộc tính model của trình trợ giúp thẻ <partial> hoặc tham số mô hình của phương thức Html.PartialAsync.

**Ví dụ về mã**

**ListModel.cs (Mô hình)**

1. không gian tên PartialViewsExample.Models
2. {
3. lớp công cộng ListModel
4. {
5. chuỗi công cộng ListTitle { get; set; } = "";
6. public List<string> ListItems { get; set; } = new List<string>();
7. }
8. }

Mô hình này sẽ được sử dụng để biểu diễn dữ liệu cho danh sách được hiển thị trong chế độ xem một phần.

**\_ListPartialView.cshtml (Chế độ xem một phần được gõ mạnh)**

1. @model ListModel // Chỉ định loại mô hình
3. <div class="list-container">
4. <h3>@Model.Danh sáchTiêu đề</h3>
5. <ul class="list">
6. @foreach (mục chuỗi trong Model.ListItems)
7. {
8. <li>@item< / li>
9. }
10. < / ul>
11. < / div>

* **@model ListModel:** Chỉ thị này chỉ ra rằng chế độ xem một phần này mong đợi một mô hình thuộc loại ListModel.
* **Truy cập thuộc tính mô hình:** Bạn có thể truy cập trực tiếp các thuộc tính của đối tượng mô hình bằng cách sử dụng @Model.PropertyName.

**Index.cshtml (Chế độ xem chính)**

1. @using PartialViewsExample.Models
3. <h1>Trang chủ</h1>
5. @{
6. ListModel listModel = mới ListModel();
7. listModel.ListTitle = "Quốc gia";
8. listModel.ListItems = new List<string>() { "Hoa Kỳ", "Canada", "Nhật Bản", "Đức", "Ấn Độ" };
9. }
10. <một phần tên="\_ListPartialView" model="listModel" />

* **Tạo Model:** Một thực thể của ListModel được tạo và điền dữ liệu.
* **Chuyển Model:** Thuộc tính model trong trình trợ giúp thẻ <partial> được sử dụng để chuyển listModel đến chế độ xem một phần \_ListPartialView.

**About.cshtml (Chế độ xem chính)**

1. @{
2. ListModel listModel = mới ListModel();
3. listModel.ListTitle = "Ngôn ngữ lập trình";
4. listModel.ListItems = danh sách mới<chuỗi>()
5. {
6. "Java",
7. "C#",
8. "Trăn"
9. };
11. chờ đợi Html.RenderPartialAsync("\_ListPartialView", listModel);
12. }

Tương tự như Index.cshtml, nhưng sử dụng trình trợ giúp HTML Html.RenderPartialAsync.

**Ghi chú**

* **Nhập mạnh:** Sử dụng @model để chỉ định loại dữ liệu mà chế độ xem một phần mong đợi.
* **Chuyển mô hình:** Truyền một thực thể của lớp mô hình của bạn khi hiển thị chế độ xem một phần.
* **An toàn kiểu và IntelliSense:** Hưởng lợi từ việc kiểm tra thời gian biên dịch và hoàn thành mã khi làm việc với các thuộc tính mô hình.
* **Phương pháp hay nhất:** Chế độ xem một phần được nhập mạnh thúc đẩy mã sạch sẽ, dễ bảo trì và ít xảy ra lỗi hơn.

PartialViewResult

Trong ASP.NET Core MVC, PartialViewResult là một loại ActionResult cụ thể được thiết kế để trả về chế độ xem một phần từ các hành động của bộ điều khiển. Chế độ xem một phần, như bạn đã biết, là các phần đánh dấu Razor (.cshtml) có thể tái sử dụng có thể được nhúng trong các dạng xem khác. Tuy nhiên, thay vì được hiển thị dưới dạng một trang đầy đủ, chúng được dự định là một đoạn HTML có thể được chèn vào chế độ xem lớn hơn.

**Tại sao nên sử dụng PartialViewResult?**

* **Phân phối nội dung động:** Bạn có thể sử dụng PartialViewResult để tải nội dung động vào chế độ xem chính của mình bằng cách sử dụng AJAX hoặc các kỹ thuật phía máy khách khác.
* **Tách mối quan tâm:** Loại kết quả này giúp giữ cho các hành động của bộ điều khiển của bạn tập trung vào việc trả về dữ liệu và để lại việc hiển thị dữ liệu đó cho chế độ xem.
* **Thử nghiệm đơn giản:** Vì PartialViewResult không liên quan đến việc hiển thị một trang hoàn chỉnh, nên việc kiểm tra đơn vị các phương thức hành động trả về chế độ xem một phần thường dễ dàng hơn.

**Tạo PartialViewResult**

Trong các hành động của bộ điều khiển, bạn có thể trả về PartialViewResult theo một số cách:

1. **Khởi tạo trực tiếp:**
2. trả về PartialViewResult mới
3. {
4. ViewName = "\_ListPartialView", // Tên của chế độ xem một phần
5. ViewData = new ViewDataDictionary(ViewData, model) // Truyền dữ liệu mô hình
6. };
7. **Sử dụng phương thức trợ giúp PartialView():**
8. trả về PartialView("\_ListPartialView", model); Cách đơn giản hơn nhiều

Phương thức PartialView() là một cách viết tắt được cung cấp bởi lớp cơ sở Controller để tạo PartialViewResult một cách thuận tiện.

**Ví dụ về mã**

**Views/Home/Index.cshtml (Chế độ xem chính)**

1. <button class="button button-blue-back" type="button" id="button-load">Tải ngôn ngữ lập trình</button>
2. <div class="programming-languages-content">
4. < / div>
6. <kịch bản>
7. document.querySelector("#button-load").addEventListener("click", async function() {
8. var response = await fetch("ngôn ngữ lập trình");
9. var languages = chờ đợi response.text();
10. document.querySelector(".programming-languages-content").innerHTML = ngôn ngữ;
11. });
12. </kịch bản>

Chế độ xem này chứa một nút, một div và một số JavaScript để tải nội dung của chế độ xem một phần khi người dùng nhấp vào một nút bằng cách sử dụng chức năng tìm nạp.

**HomeController.cs (Bộ điều khiển)**

1. [Tuyến ("ngôn ngữ lập trình")]
2. IActionResult công khai ProgrammingLanguages()
3. {
4. ListModel listModel = mới ListModel() {
5. ListTitle = "Danh sách ngôn ngữ lập trình",
6. ListItems = new List<string>() { "Python", "C#", "Go" }
7. };
9. trả về PartialView("\_ListPartialView", listModel);
10. }

Trong phương pháp hành động này:

1. **Chuẩn bị dữ liệu:** Đối tượng ListModel được tạo và điền dữ liệu cho danh sách (tiêu đề và mục).
2. **Chế độ xem một phần được trả về:** Phương thức PartialView() được sử dụng để trả về PartialViewResult. Phương thức này có hai đối số:
   * "\_ListPartialView": Tên của tệp chế độ xem một phần (nằm trong Chế độ xem/Được chia sẻ theo mặc định).
   * listModel: Dữ liệu mô hình để chuyển đến chế độ xem một phần.
   * Khi phương thức hành động này được gọi thông qua URL /programming-languages bởi mã javascript trên trang, nó trả về chế độ xem một phần cùng với dữ liệu sau đó được đưa vào div với lớp programming-languages-content bởi javascript.

**Những điểm quan trọng:**

* **AJAX và Tải động:** PartialViewResult thường được sử dụng cùng với AJAX để tải nội dung linh hoạt mà không cần làm mới toàn bộ trang.
* **Mô hình chế độ xem (Phương pháp hay nhất):** Luôn cố gắng sử dụng mô hình chế độ xem để chuyển dữ liệu đến chế độ xem một phần của bạn để có tính bảo trì và an toàn kiểu tốt hơn.
* **Bộ nhớ đệm:** Cân nhắc sử dụng bộ nhớ đệm đầu ra trên chế độ xem một phần của bạn để cải thiện hiệu suất nếu chúng hiển thị dữ liệu không thay đổi thường xuyên.

Những điểm chính cần nhớ

**Chế độ xem một phần: Các thành phần chế độ xem có thể tái sử dụng**

* **Mục đích:** Đóng gói các phần tử giao diện người dùng có thể tái sử dụng hoặc các đoạn đánh dấu Razor (.cshtml) để tránh lặp lại.
* **Lợi ích:**
  + Tăng khả năng tái sử dụng mã
  + Cải thiện khả năng tổ chức và bảo trì mã
  + Tạo nội dung động
* **Quy ước đặt tên:** Tiền tố bằng dấu gạch dưới (ví dụ: \_PartialName.cshtml).
* **Vị trí:** Thường là trong Chế độ xem/Đã chia sẻ hoặc bên cạnh chế độ xem chính.
* **Tùy chọn kết xuất:**
  + **Trình trợ giúp thẻ:** <partial name="\_PartialName" /> hoặc <partial name="\_PartialName" model="yourModel" />
  + **Trình trợ giúp HTML:** @await Html.PartialAsync("\_PartialName", model)

**ViewData trong chế độ xem một phần**

* **Mục đích:** Chuyển dữ liệu đến dạng xem một phần từ chế độ xem hoặc bộ điều khiển chính.
* **Sử dụng:**
  + Trong chế độ xem hoặc bộ điều khiển gốc: ViewData["key"] = value
  + Trong chế độ xem một phần: @ViewData["key"]
* **Lưu ý:** ViewData không được nhập mạnh; hãy cẩn thận với việc tạo kiểu và kiểm tra rỗng.

**Chế độ xem một phần được gõ mạnh**

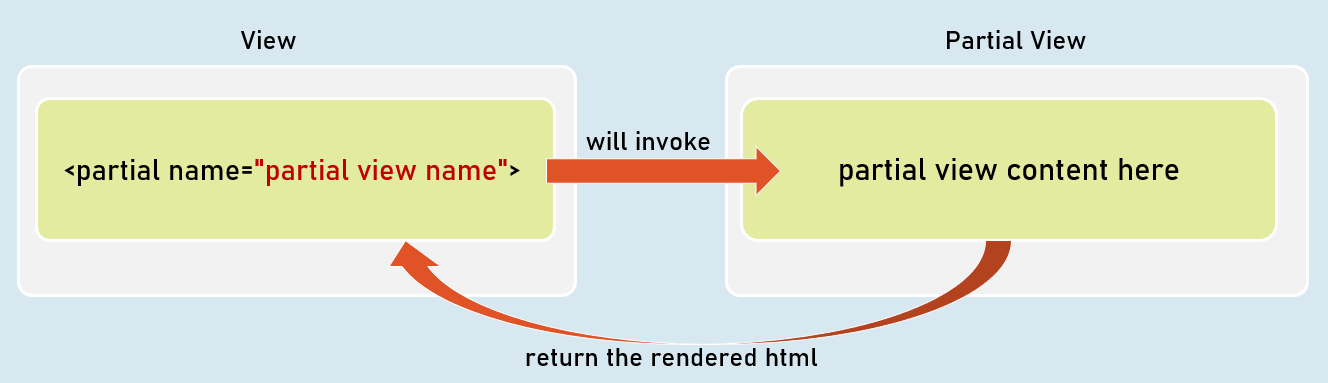
* **Mục đích:** Liên kết một chế độ xem một phần với một lớp mô hình cụ thể bằng cách sử dụng chỉ thị @model.
* **Lợi ích:**
  + An toàn loại: Bắt lỗi trong quá trình phát triển.
  + IntelliSense: Hoàn thành mã cho các thuộc tính mô hình.
  + Khả năng bảo trì: Cập nhật dễ dàng hơn khi mô hình thay đổi.
* **Cách sử dụng:**
  + Tạo một lớp mô hình.
  + Sử dụng @model YourModel trong chế độ xem một phần.
  + Chuyển một thực thể mô hình khi hiển thị: <partial model="yourModelInstance" /> hoặc @await Html.PartialAsync("\_PartialName", yourModelInstance)

**PartialViewResult**

* **Mục đích:** Trả về chế độ xem một phần từ hành động của bộ điều khiển (thường là đối với các yêu cầu AJAX).
* **Sáng tạo:**
  + trả về PartialView("\_PartialName", model); (ưu tiên)
  + return new PartialViewResult { ViewName = "\_PartialName", ViewData = ... };

Chế độ xem một phần

Chế độ xem một phần là một tệp đánh dấu dao cạo (.cshtml) không thể được gọi riêng lẻ từ bộ điều khiển; nhưng có thể được gọi từ bất kỳ chế độ xem nào trong cùng một ứng dụng web.



Gọi chế độ xem một phần

<partial name="tên một phần view" />

Trả lại nội dung cho chế độ xem mẹ.

@await Html.PartialAsync("tên dạng xem một phần")

Trả lại nội dung cho chế độ xem mẹ.

@{ await Html.RenderPartialAsync("tên chế độ xem một phần"); }

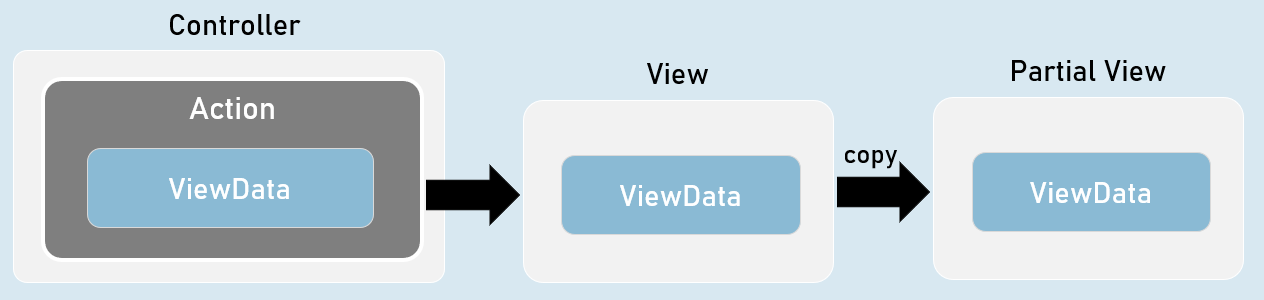
Truyền nội dung đến trình duyệt.

Chế độ xem một phần với ViewData

Khi chế độ xem một phần được gọi, nó sẽ nhận được một bản *sao* của đối tượng ViewData của chế độ xem gốc.

Vì vậy, bất kỳ thay đổi nào được thực hiện trong ViewData trong chế độ xem một phần, KHÔNG ảnh hưởng đến ViewData của chế độ xem gốc.

Theo tùy chọn, bạn có thể cung cấp đối tượng ViewData tùy chỉnh cho chế độ xem một phần, nếu bạn không muốn chế độ xem một phần truy cập vào toàn bộ ViewData của chế độ xem gốc.



**Gọi chế độ xem một phần với dữ liệu chế độ xem**

@{ await Html.RenderPartialAsync("tên chế độ xem một phần", ViewData); }

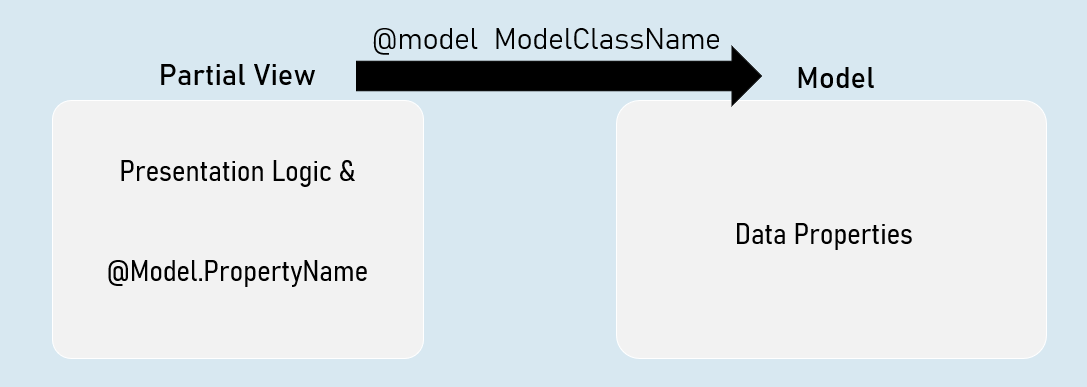
--hoặc--

<partial name="một phần view name" view-data="ViewData" />

Chế độ xem một phần được gõ mạnh

Chế độ xem một phần được gõ mạnh là chế độ xem một phần được liên kết với một lớp mô hình được chỉ định.

Vì vậy, nó nhận được tất cả các lợi ích của một chế độ xem được gõ mạnh.



**Gọi chế độ xem một phần gõ mạnh**

@{ await Html.RenderPartialAsync("tên chế độ xem một phần", Model); }

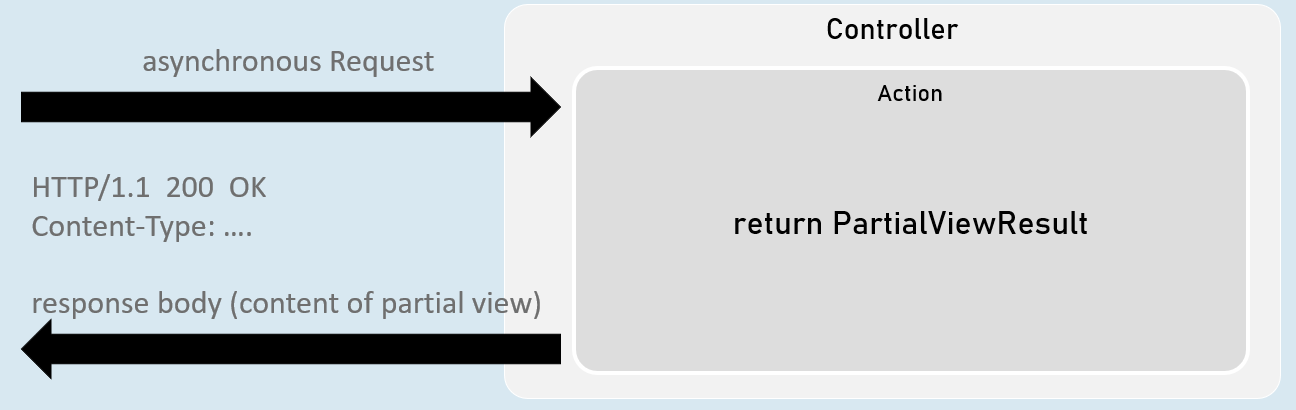
--hoặc--

<partial name="partial view name" model="Model" />

PartialViewResult

PartialViewResult có thể đại diện cho nội dung của một phần.

Nói chung hữu ích để tìm nạp nội dung của một phần chế độ xem vào trình duyệt, bằng cách thực hiện một yêu cầu không đồng bộ (XMLHttpRequest / yêu cầu tìm nạp) từ trình duyệt.



return new PartialViewResult() { ViewName = "tên chế độ xem một phần", Model = model };

[hoặc]

return PartialView("partial view name", model);

# **9. Xem các thành phần [MVC]**

Xem các thành phần

View Components là các khối xây dựng giao diện người dùng khép kín, có thể tái sử dụng trong ASP.NET Core MVC. Chúng được thiết kế để đóng gói logic kết xuất phức tạp hơn những gì bạn thường đặt trong chế độ xem một phần nhưng không đảm bảo độ phức tạp của bộ điều khiển và chế độ xem đầy đủ.

**Mục đích**

* **Đóng gói độ phức tạp:** Nhóm logic hiển thị giao diện người dùng liên quan thành một đơn vị gắn kết.
* **Khả năng tái sử dụng:** Sử dụng Thành phần chế độ xem trên nhiều chế độ xem để tránh trùng lặp mã.
* **Khả năng kiểm tra:** Dễ dàng kiểm tra đơn vị View Components do tính chất khép kín của chúng.
* **Logic kết xuất:** Lý tưởng cho các tiện ích động, menu điều hướng, biểu mẫu đăng nhập, tóm tắt giỏ hàng hoặc bất kỳ yếu tố giao diện người dùng nào liên quan đến việc tìm nạp dữ liệu hoặc thực hiện logic trước khi kết xuất.

**Các phương pháp hay nhất**

* **Đặt tên:** Các lớp Thành phần chế độ xem phải kết thúc bằng hậu tố ViewComponent (ví dụ: ProductListViewComponent).
* **Cấu trúc:** Đặt View Components trong thư mục ViewComponents. Tệp dạng xem được liên kết phải được đặt trong Views/Shared/Components/{ViewComponent Name}/{View Name}.cshtml.
* **Dữ liệu:** Chuyển dữ liệu đến Thành phần chế độ xem bằng cách sử dụng mô hình hoặc mô hình dạng xem được nhập mạnh.
* **Không đồng bộ:** Sử dụng các phương thức không đồng bộ (InvokeAsync) để tránh chặn luồng khi thực hiện truy cập dữ liệu hoặc các hoạt động I/O khác.
* **Đơn giản:** Giữ cho logic của View Component tập trung vào việc hiển thị phần tử giao diện người dùng. Tránh logic nghiệp vụ phức tạp trong thành phần.

**Những điều cần tránh**

* **Sử dụng quá mức:** Không sử dụng Thành phần chế độ xem cho các thành phần giao diện người dùng đơn giản có thể được xử lý bởi chế độ xem một phần.
* **Khớp nối chặt chẽ:** Tránh khớp nối chặt chẽ Xem các thành phần với bộ điều khiển hoặc hành động cụ thể. Làm cho chúng có thể tái sử dụng trên các phần khác nhau của ứng dụng của bạn.
* **Logic phức tạp:** View Components không nên chịu trách nhiệm xử lý logic nghiệp vụ. Thay vào đó, hãy ủy quyền điều đó cho các mô hình hoặc dịch vụ của bạn.
* **Truy cập cơ sở dữ liệu trực tiếp:** Tránh truy cập trực tiếp vào cơ sở dữ liệu của bạn từ bên trong Thành phần Xem. Sử dụng dịch vụ để truy cập dữ liệu.

**Khi nào nên sử dụng các thành phần chế độ xem**

* **Các thành phần giao diện người dùng phức tạp:** Khi một phần tử giao diện người dùng yêu cầu logic kết xuất phức tạp hơn so với chế độ xem một phần đơn giản.
* **Các phần tử theo hướng dữ liệu:** Khi bạn cần tìm nạp dữ liệu hoặc thực hiện một số tính toán trước khi hiển thị phần tử giao diện người dùng.
* **Widget có thể tái sử dụng:** Khi bạn muốn tạo một widget có thể tái sử dụng có thể được sử dụng trong các phần khác nhau của ứng dụng của mình.

**Cách triển khai các thành phần chế độ xem**

1. **Tạo lớp thành phần View:** Lấy từ ViewComponent và triển khai phương thức Invoke hoặc InvokeAsync.
2. **Tạo dạng xem:** Tạo tệp dạng xem Razor (.cshtml) trong thư mục Views/Shared/Components/{ViewComponent Name}.
3. **Gọi trong chế độ xem của bạn:** Sử dụng trình trợ giúp @await Component.InvokeAsync("ViewComponent Name", arguments) trong chế độ xem chính của bạn để hiển thị View Component.

Mã

1. GridViewComponent.cs (Xem thành phần)
2. lớp công khai GridViewComponent : ViewComponent
3. {
4. public async Task<IViewComponentResult> InvokeAsync()
5. {
6. Dữ liệu cho thành phần chế độ xem (có thể đến từ cơ sở dữ liệu, dịch vụ, v.v.)
7. Mô hình PersonGridModel = PersonGridModel() mới
8. {
9. GridTitle = "Danh sách người",
10. Người = danh sách mới<Người>()
11. {
12. new Person() { PersonName = "John", JobTitle = "Người quản lý" },
13. Nhiều người hơn
14. }
15. };
17. ViewBag không lý tưởng; thích sử dụng mô hình được gõ mạnh
18. ViewData["Grid"] = mô hình;
20. return View("Mẫu"); Thao tác này sẽ tìm kiếm dạng xem trong Views/Shared/Components/Grid/Sample.cshtml
21. }
22. }
23. Views/Home/Index.cshtml (Chế độ xem chính)
24. @await Component.InvokeAsync("Grid") // Gọi thành phần chế độ xem "Grid"
25. Lượt xem/Trang chủ/Giới thiệu.cshtml
26. @await Component.InvokeAsync("Grid")
28. <vc:grid></vc:grid> // cú pháp thay thế (không được ưu tiên, vì nó không được gõ mạnh)

**Lời giải thích:**

* Lớp GridViewComponent là một thành phần chế độ xem tìm nạp danh sách mọi người và chuyển nó đến chế độ xem một phần "Mẫu" cùng với tiêu đề lưới bằng ViewBag.
* Views/Home/Index.cshtml và Views/Home/About.cshtml gọi thành phần dạng xem Lưới bằng cách sử dụng @await Component.InvokeAsync("Grid") (hoặc ít lý tưởng hơn là trình trợ giúp thẻ vc:grid), hiển thị danh sách người ở định dạng lưới.

**Ghi chú**

* **Mục đích:** Đóng gói logic hiển thị giao diện người dùng phức tạp, có thể tái sử dụng.
* **Đặt tên:** Các lớp ViewComponent kết thúc bằng ViewComponent.
* **Vị trí:** Thư mục ViewComponents cho các lớp, Views/Shared/Components/{ViewComponent Name} cho chế độ xem.
* **Truyền dữ liệu:** Sử dụng các mô hình được nhập mạnh hoặc mô hình xem (ưu tiên hơn ViewBag).
* **Kết xuất:** @await Component.InvokeAsync("ViewComponent Name", arguments)

Các thành phần chế độ xem được gõ mạnh

Cũng giống như các chế độ xem được gõ mạnh, các thành phần chế độ xem được gõ mạnh được liên kết với một lớp mô hình cụ thể bằng cách sử dụng chỉ thị @model. Lớp mô hình này, thường được gọi là "mô hình chế độ xem", cung cấp một cách có cấu trúc để chuyển dữ liệu từ thành phần chế độ xem đến chế độ xem tương ứng của nó. Lợi ích chính là tăng cường an toàn kiểu và cải thiện khả năng bảo trì mã.

**Lợi ích của các thành phần chế độ xem được gõ mạnh**

* **An toàn loại:** Nắm bắt lỗi trong quá trình phát triển do kiểu dữ liệu hoặc tên thuộc tính không khớp, vì công cụ Razor thực thi kiểm tra kiểu dựa trên lớp mô hình của bạn.
* **IntelliSense:** Nâng cao năng suất bằng cách cung cấp tính năng hoàn thành mã và đề xuất cho các thuộc tính mô hình trực tiếp trong dạng xem Razor. Điều này dẫn đến ít lỗi chính tả hơn và phát triển nhanh hơn.
* **Tái cấu trúc:** Đơn giản hóa quá trình cập nhật chế độ xem khi mô hình của bạn thay đổi. Vì chế độ xem được gõ mạnh, việc đổi tên hoặc sửa đổi thuộc tính trong mô hình sẽ tự động cập nhật các tham chiếu trong chế độ xem.
* **Rõ ràng và dễ đọc:** Làm cho mã của bạn tự lập tài liệu hơn bằng cách xác định rõ ràng cấu trúc dữ liệu mà thành phần chế độ xem mong đợi.

**Cách triển khai các thành phần chế độ xem gõ mạnh**

1. **Tạo mô hình chế độ xem:** Xác định một lớp đại diện cho dữ liệu bạn muốn chuyển đến thành phần chế độ xem của mình. Lớp này phải chứa tất cả các thuộc tính cần thiết mà thành phần view sẽ sử dụng để hiển thị đầu ra của nó.
2. **Sử dụng @model trong Chế độ xem:** Trong tệp dạng xem Razor của thành phần dạng xem (ví dụ: Default.cshtml), hãy sử dụng lệnh @model để chỉ định lớp mô hình chế độ xem.
3. **Trả về Mô hình từ InvokeAsync:** Trong phương thức InvokeAsync của thành phần chế độ xem, hãy tạo một thực thể của mô hình chế độ xem, điền dữ liệu cần thiết và trả về nó bằng cách sử dụng View(model).

Mã

**PersonGridModel.cs (Xem mô hình)**

1. lớp công cộng PersonGridModel
2. {
3. chuỗi công cộng GridTitle { get; set; }
4. public List<Person> Persons { get; set; }
5. }

Mô hình này sẽ được sử dụng để biểu diễn dữ liệu được truyền từ thành phần view đến view.

**GridViewComponent.cs**

1. ViewComponent
2. lớp công khai GridViewComponent : ViewComponent
3. {
4. public async Task<IViewComponentResult> InvokeAsync()
5. {
6. PersonGridModel personGridModel = PersonGridModel() mới
7. {
8. GridTitle = "Danh sách người",
9. Người = danh sách mới<Người>() {
10. new Person() { PersonName = "John", JobTitle = "Người quản lý" },
11. new Person() { PersonName = "Jones", JobTitle = "Trợ lý Quản lý" },
12. new Person() { PersonName = "William", JobTitle = "Thư ký" },
13. }
14. };
15. trả về View("Sample", personGridModel);
16. }
17. }

Phương thức này bây giờ tạo một đối tượng PersonGridModel, điền nó và truyền nó làm đối số thứ hai của phương thức View().

**Views/Shared/Components/Grid/Sample.cshtml**

1. @model PersonGridModel
3. <div class="box">
4. <h3>@Model.GridTitle</h3>
5. <table class="bảng w-100">
6. <đầu>
7. <tr>
8. <>Sl. Số</th>
9. <>tên</th>
10. < / tr>
11. < / thead>
12. <tbody>
13. @foreach (Người trong Model.Persons)
14. {
15. <tr>
16. <td>@person. Tên người< / td>
17. <td>@person. Chức danh</td>
18. < / tr>
19. }
20. < / tbody>
21. </bảng>
22. < / div>

* **@model PersonGridModel:** Điều này chỉ định rằng chế độ xem được nhập mạnh và mong đợi một đối tượng thuộc loại PersonGridModel.
* **Truy cập thuộc tính:** Chế độ xem truy cập trực tiếp vào các thuộc tính của đối tượng Model như Model.GridTitle và Model.Persons.

**Ghi chú**

* **Nhập mạnh:** Sử dụng @model trong tệp chế độ xem của thành phần chế độ xem để chỉ định loại mô hình chế độ xem.
* **Chuyển Model:** Chuyển một phiên bản của lớp mô hình view của bạn khi trả về view từ phương thức InvokeAsync: return View("ViewName", model);
* **Đặt tên:** Các thành phần chế độ xem phải được đặt tên bằng hậu tố "ViewComponent" (ví dụ: ProductListViewComponent).
* **Vị trí:** Các lớp thành phần dạng xem nằm trong thư mục ViewComponents và các dạng xem được liên kết của chúng nằm trong Views/Shared/Components/{ViewComponent Name}/.

Xem các thành phần với các tham số

Mặc dù các thành phần chế độ xem có thể hoạt động mà không cần tham số, nhưng việc truyền các tham số cho chúng giúp tăng cường đáng kể tính linh hoạt và khả năng tái sử dụng của chúng. Các tham số cho phép bạn tùy chỉnh hành vi và đầu ra của thành phần chế độ xem dựa trên dữ liệu được cung cấp bởi chế độ xem hoặc bộ điều khiển gọi. Điều này làm cho các thành phần chế độ xem năng động hơn và có thể thích ứng với các tình huống khác nhau trong ứng dụng của bạn.

**Cách truyền các tham số để xem các thành phần**

1. **Xác định tham số trong InvokeAsync:** Trong lớp thành phần chế độ xem của bạn, hãy xác định các tham số mà bạn muốn nhận trong phương thức InvokeAsync (hoặc Invoke). Các tham số này sẽ trở thành một phần của chữ ký của phương thức.
2. **Truyền đối số khi gọi:** Khi gọi thành phần chế độ xem bằng cách sử dụng @await Component.InvokeAsync("ViewComponent Name", arguments) hoặc trình trợ giúp thẻ <vc:grid>, hãy chuyển một đối tượng ẩn danh chứa các giá trị tham số. Tên thuộc tính trong đối tượng ẩn danh phải khớp với tên tham số trong phương thức InvokeAsync.

Mã

**GridViewComponent.cs**

1. lớp công khai GridViewComponent : ViewComponent
2. {
3. public async Task<IViewComponentResult> InvokeAsync(PersonGridModel grid)
4. {
5. trả lại View("Sample", lưới);
6. }
7. }

**Tham số:** Phương thức InvokeAsync hiện nhận một tham số thuộc loại PersonGridModel. Điều này có nghĩa là thành phần view mong đợi nhận được một mô hình chứa dữ liệu lưới khi nó được gọi.

**Lượt xem/Trang chủ/Index.cshtml**

1. @{
2. PersonGridModel personGridModel = new PersonGridModel() { /\* ... (khởi tạo dữ liệu) ... \*/ };
3. }
5. @await Component.InvokeAsync("Grid", new { grid = personGridModel })
7. @{
8. PersonGridModel personGridModel2 = new PersonGridModel() { /\* ... (khởi tạo dữ liệu khác nhau) ... \*/ };
9. }
10. @await Component.InvokeAsync("Grid", new { grid = personGridModel2 })

Hai thực thể của PersonGridModel được tạo và chuyển dưới dạng đối số cho thành phần view khi gọi nó.

**Lượt xem/Trang chủ/Giới thiệu.cshtml**

1. @{
2. PersonGridModel personGridModel = new PersonGridModel() { /\* ... (khởi tạo dữ liệu) ... \*/ };
3. }
5. <vc:grid grid="personGridModel"></vc:grid>

* **Sử dụng Tag Helper:** Trình trợ giúp thẻ <vc:grid> được sử dụng ở đây để gọi thành phần view. Thuộc tính grid được đặt thành personGridModel, chuyển dữ liệu mô hình đến thành phần view.

**Lợi ích của các thành phần chế độ xem được gõ mạnh với các tham số**

* **An toàn kiểu:** Đảm bảo rằng bạn chuyển đúng kiểu dữ liệu đến thành phần chế độ xem, ngăn ngừa lỗi thời gian chạy do không khớp kiểu.
* **IntelliSense:** Bạn nhận được đề xuất hoàn thành mã cho tên tham số và thuộc tính mô hình, giúp phát triển nhanh hơn và hiệu quả hơn.
* **Tính linh hoạt:** Tùy chỉnh hành vi của thành phần chế độ xem dựa trên các thông số được cung cấp, giúp nó dễ thích ứng hơn với các tình huống khác nhau.
* **Khả năng bảo trì:** Dễ dàng tái cấu trúc và sửa đổi thành phần chế độ xem và cách sử dụng nó trong chế độ xem của bạn.

**Ghi chú**

* **Xác định tham số:** Xác định rõ ràng các tham số mà thành phần chế độ xem của bạn mong đợi trong phương thức InvokeAsync.
* **Truyền tham số:** Khi gọi thành phần chế độ xem, hãy chuyển một đối tượng ẩn danh với các thuộc tính khớp với tên và loại tham số.
* **Nhập mạnh:** Sử dụng mô hình chế độ xem (như PersonGridModel) để chuyển dữ liệu có cấu trúc đến thành phần chế độ xem của bạn và hưởng lợi từ an toàn kiểu và IntelliSense.

ViewComponentResult

Mặc dù các thành phần chế độ xem chủ yếu được gọi từ bên trong chế độ xem, nhưng lớp ViewComponentResult cho phép bạn trực tiếp trả về một thành phần chế độ xem được hiển thị từ một hành động của bộ điều khiển. Điều này cung cấp một cách mạnh mẽ để tích hợp các thành phần chế độ xem với logic bộ điều khiển MVC của bạn và tận dụng khả năng hiển thị có thể tái sử dụng của chúng.

**Mục đích**

* **Tải thành phần chế độ xem động:** Sử dụng ViewComponentResult để tải các thành phần chế độ xem theo yêu cầu dựa trên logic hoặc dữ liệu của bộ điều khiển. Điều này đặc biệt hữu ích để hiển thị các thành phần giao diện người dùng phức tạp phụ thuộc vào dữ liệu hoặc điều kiện cụ thể.
* **Phân tách các mối quan tâm:** Giữ cho các hành động của bộ điều khiển của bạn tập trung vào việc xử lý các yêu cầu và truy xuất dữ liệu, đồng thời ủy quyền hiển thị các thành phần giao diện người dùng cụ thể cho các thành phần xem.
* **Điểm cuối giống API:** Tạo điểm cuối giống API trả về các đoạn HTML (thành phần xem) thay vì dữ liệu JSON hoặc XML. Điều này có thể hữu ích cho việc xây dựng các ứng dụng kết hợp, nơi bạn cần kết hợp kết xuất phía máy chủ với JavaScript phía máy khách.

**Tạo ViewComponentResult**

Trong các hành động của bộ điều khiển, bạn có thể trả về ViewComponentResult bằng cách sử dụng phương thức trợ giúp ViewComponent():

1. return ViewComponent("ViewComponent Name", arguments);

* **ViewComponent Name:** Tên của view component bạn muốn gọi. Điều này phải khớp với tên lớp của thành phần chế độ xem của bạn (không có hậu tố "ViewComponent").
* **arguments (Tùy chọn):** Một đối tượng ẩn danh chứa các tham số bạn muốn chuyển đến phương thức InvokeAsync của thành phần view.

Mã

1. HomeController.cs (Bộ điều khiển)
2. [Lộ trình ("danh sách bạn bè")]
3. IActionResult công khai LoadFriendsList()
4. {
5. PersonGridModel personGridModel = PersonGridModel() mới
6. {
7. GridTitle = "Bạn bè",
8. Người = danh sách mới<Người>()
9. {
10. // ... (danh sách bạn bè)...
11. }
12. };
14. return ViewComponent("Grid", new { grid = personGridModel });
15. }

Trong mã này:

1. **Chuẩn bị dữ liệu:** Đối tượng personGridModel được tạo và điền dữ liệu cho danh sách bạn bè.
2. **ViewComponentResult:** Phương thức ViewComponent() được sử dụng để trả về ViewComponentResult. Nó có hai đối số:
   * "Grid": Điều này cho biết rằng chúng ta muốn gọi GridViewComponent.
   * new { grid = personGridModel }: Đối tượng ẩn danh này chuyển personGridModel làm tham số lưới cho phương thức InvokeAsync của GridViewComponent.

**Cách thức hoạt động**

1. **Yêu cầu:** Yêu cầu đến /friends-list kích hoạt hành động LoadFriendsList.
2. **Lệnh gọi thành phần xem:** ViewComponentResult được trả về từ hành động khiến MVC cốt lõi ASP.NET định vị và gọi GridViewComponent.
3. **Thực thi thành phần xem:** Phương thức InvokeAsync trong GridViewComponent nhận personGridModel và sử dụng nó để hiển thị chế độ xem một phần "Mẫu".
4. **Phản hồi:** Đầu ra được hiển thị của GridViewComponent (HTML cho lưới bạn bè) được trả về dưới dạng phản hồi HTTP.

**Ghi chú**

* **Mục đích:** Hiển thị các thành phần chế độ xem trực tiếp từ các hành động của bộ điều khiển.
* **Tính linh hoạt:** Tải các thành phần chế độ xem linh hoạt dựa trên logic của bộ điều khiển.
* **Điểm cuối kiểu API:** Hữu ích để tạo điểm cuối trả về các đoạn HTML.
* **Cú pháp:** Sử dụng ViewComponent("ViewComponent Name", arguments) để tạo ViewComponentResult.
* **Tham số:** Chuyển các tham số đến thành phần chế độ xem bằng cách sử dụng đối tượng ẩn danh.

Những điểm chính cần nhớ

**1. Xem các thành phần: Các thành phần giao diện người dùng mô-đun**

* **Mục đích: Đóng**  gói logic hiển thị giao diện người dùng có thể tái sử dụng phức tạp hơn chế độ xem một phần.
* **Lợi ích:**
  + Mô-đun hóa và tái sử dụng
  + Tách các mối quan tâm (logic giao diện người dùng từ bộ điều khiển)
  + Cải thiện khả năng kiểm tra
* **Cấu trúc:**
  + Class: Kế thừa từ ViewComponent (ví dụ: ProductListViewComponent).
  + Xem: Tệp dao cạo trong Views/Shared/Components/{ViewComponent Name}/{View Name}.cshtml.

**2. Phương pháp**

* **Invoke hoặc InvokeAsync:** Phương thức chính cho logic của thành phần của bạn.
  + Trả về IViewComponentResult.
  + Có thể lấy các thông số để tùy chỉnh.
  + Sử dụng async cho các hoạt động không đồng bộ.

**3. Gọi các thành phần View**

* **Trong Lượt xem:**
  + @await Component.InvokeAsync("ViewComponent Name", đối số)
  + <vc:view-component-name></vc:view-component-name> (cú pháp Tag Helper, ít ưu tiên hơn)
* **Trong Bộ điều khiển:**
  + return ViewComponent("ViewComponent Name", arguments);

**4. Các thành phần chế độ xem được gõ mạnh**

* **Mục đích:** Chuyển dữ liệu đến chế độ xem bằng cách sử dụng mô hình được nhập mạnh.
* **Lợi ích:** An toàn kiểu loại, IntelliSense, khả năng bảo trì tốt hơn.
* **Cách sử dụng:**
  1. Tạo lớp mô hình chế độ xem.
  2. Sử dụng @model YourViewModel trong tệp chế độ xem của thành phần chế độ xem.
  3. Trả về mô hình chế độ xem từ InvokeAsync: return View(model);

**5. Truyền thông số**

* **Tham số InvokeAsync:** Xác định các tham số trong chữ ký phương thức InvokeAsync.
* **Đối số gọi:** Truyền đối số dưới dạng đối tượng ẩn danh khi gọi thành phần.

**Ví dụ:**

1. ViewComponent
2. public async Task<IViewComponentResult> InvokeAsync(int categoryId)
3. {
4. var products = \_productService.GetProductsByCategory(categoryId);
5. trả lại Chế độ xem (sản phẩm);
6. }
8. Cảnh
9. @await Component.InvokeAsync("ProductList", new { categoryId = 5 })

**7. Các phương pháp hay nhất**

* **Đặt tên:** Sử dụng hậu tố "ViewComponent" cho tên lớp.
* **Cấu trúc thư mục:**
  + Xem các thành phần trong thư mục ViewComponents.
  + Chế độ xem trong Views/shared/Components/{ViewComponent Name}/.
* **Nhập mạnh:** Sử dụng mô hình chế độ xem được nhập mạnh.
* **Hoạt động đồng bộ:** Sử dụng InvokeAsync cho các tác vụ không đồng bộ.
* **Limit Logic:** Giữ logic tập trung vào kết xuất giao diện người dùng, không phải hoạt động kinh doanh.
* **Tránh truy cập cơ sở dữ liệu trực tiếp:** Sử dụng các dịch vụ để truy cập dữ liệu.

**8. Những điều cần tránh**

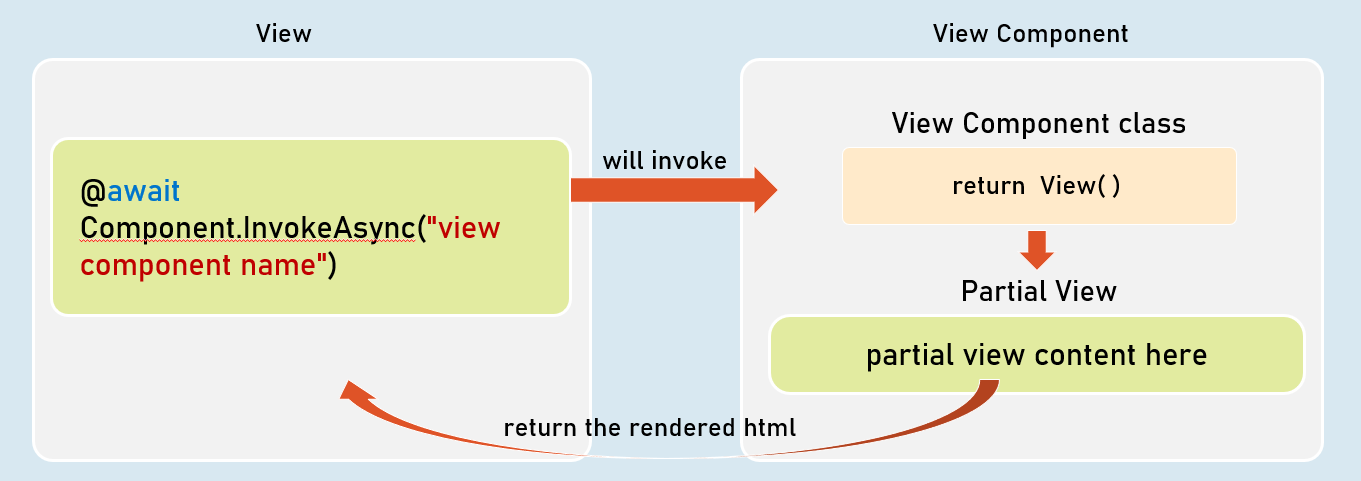
* **Lạm dụng:** Không sử dụng cho các yếu tố giao diện người dùng đơn giản.
* **Khớp nối chặt chẽ:** Giữ chúng độc lập với các bộ điều khiển cụ thể.
* **Logic phức tạp:** Tránh logic nghiệp vụ mở rộng trong các thành phần.

**9. Khi nào sử dụng**

* **Các yếu tố giao diện người dùng phức tạp:** Khi chế độ xem một phần là không đủ.
* **Các yếu tố theo hướng dữ liệu:** Nội dung động dựa trên dữ liệu / logic.
* **Tiện ích có thể tái sử dụng:** Để tạo các thành phần giao diện người dùng có thể tái sử dụng.

Xem các thành phần

Thành phần xem là sự kết hợp của một lớp (bắt nguồn từ Microsoft.AspNetCore.ViewComponent) cung cấp dữ liệu và một dạng xem một phần để hiển thị dữ liệu đó.



**Gọi View Component**

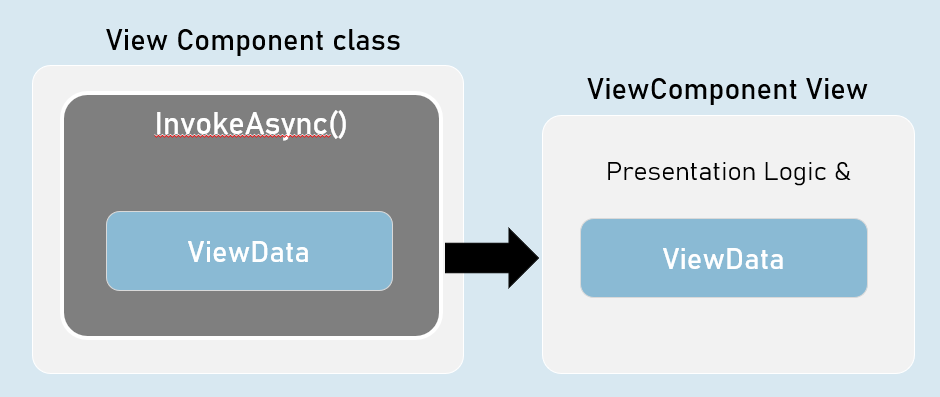
1. @await Component.InvokeAsync("view component name");
2. --hoặc--
3. <vc:view-component-name />

**Xem các thành phần**

* View hiển thị một chunk chứ không phải toàn bộ phản hồi.
* Bao gồm các lợi ích tách biệt mối quan tâm và khả năng kiểm tra tương tự được tìm thấy với bộ điều khiển và chế độ xem.
* Nên có hậu tố bằng từ "ViewComponent" hoặc phải có thuộc tính [ViewComponent].
* Tùy chọn, nó có thể kế thừa từ System.AspNetCore.Mvc.ViewComponent.

Xem các thành phần bằng ViewData

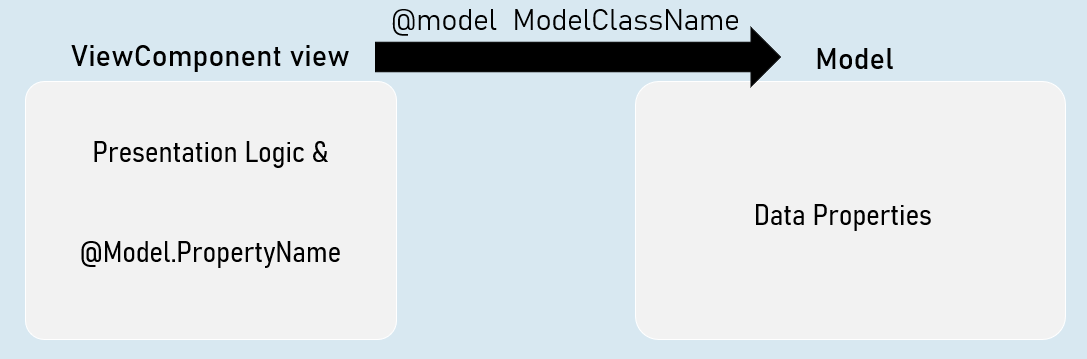
Lớp ViewComponent có thể chia sẻ đối tượng ViewData với chế độ xem ViewComponent.



ViewComponent được gõ mạnh

Chế độ xem của ViewComponent được gõ mạnh được ràng buộc chặt chẽ với một lớp mô hình được chỉ định.

Vì vậy, nó nhận được tất cả các lợi ích của một chế độ xem được gõ mạnh.

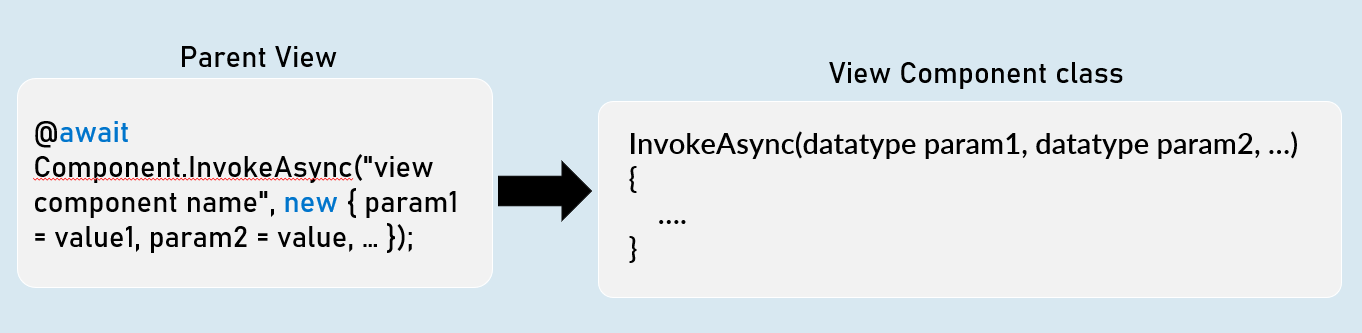


ViewComponents với các tham số

Bạn có thể cung cấp một hoặc nhiều tham số cho lớp thành phần chế độ xem.

Các tham số được nhận bởi phương thức InvokeAsync của lớp thành phần view.

Tất cả các thông số của thành phần chế độ xem là bắt buộc (phải cung cấp một giá trị).



Gọi ViewComponent bằng các tham số

@await Component.InvokeAsync("view component name", new { param = value });

--hoặc--

<vc:view-component-name param="value" />

ViewComponentResult

ViewComponent có thể đại diện cho nội dung của một thành phần chế độ xem.

Nói chung hữu ích để tìm nạp nội dung của thành phần view vào trình duyệt, bằng cách thực hiện một yêu cầu không đồng bộ (XMLHttpRequest / fetch request) từ trình duyệt.



1. return new ViewComponentResult() { ViewName = "view component name", Arguments = new { param1 = value, param2 = value } };
2. [hoặc]
3. return ViewComponent("view component name", new { param1 = value, param2 = value } });

# **10. DI [MVC so với API]**

Dịch vụ

Trong ASP.NET Core MVC, các dịch vụ là các lớp chịu trách nhiệm triển khai logic nghiệp vụ cốt lõi của ứng dụng của bạn. Chúng được thiết kế để có thể tái sử dụng, khép kín và độc lập với các bộ điều khiển hoặc chế độ xem cụ thể. Các dịch vụ là xương sống của ứng dụng của bạn, xử lý các tác vụ như truy cập dữ liệu, tính toán, giao tiếp với các hệ thống bên ngoài và bất kỳ hoạt động nào khác liên quan đến "cách thức" chức năng của ứng dụng của bạn.

**Mục đích chính của dịch vụ**

1. **Đóng gói logic nghiệp vụ:** Dịch vụ cung cấp một cách rõ ràng để đóng gói các hoạt động phức tạp và giữ chúng tách biệt với lớp trình bày của bạn (bộ điều khiển và chế độ xem).
2. **Khả năng tái sử dụng:** Một dịch vụ duy nhất có thể được sử dụng bởi nhiều bộ điều khiển, thúc đẩy các nguyên tắc DRY (Đừng lặp lại bản thân) và làm cho mã của bạn dễ bảo trì hơn.
3. **Khả năng kiểm tra:** Các dịch vụ có thể dễ dàng được kiểm tra đơn vị một cách riêng biệt, cho phép bạn xác minh tính đúng đắn của logic nghiệp vụ của mình mà không phải trả chi phí cho việc chạy toàn bộ ứng dụng.
4. **Dependency Injection (DI):** Các dịch vụ thường được đăng ký trong vùng chứa DI, giúp các bộ điều khiển và các thành phần khác trong ứng dụng của bạn dễ dàng truy cập chúng.

**Trách nhiệm điển hình của dịch vụ**

* **Truy cập dữ liệu:** Giao tiếp với cơ sở dữ liệu hoặc các nguồn dữ liệu khác để tìm nạp, chèn, cập nhật hoặc xóa dữ liệu.
* **Quy tắc kinh doanh:** Thực hiện các quy tắc chi phối cách ứng dụng của bạn hoạt động (ví dụ: xác thực, tính toán, chuyển đổi).
* **Tích hợp:** Tương tác với các hệ thống hoặc API bên ngoài.
* **Thông báo:** Gửi email, tin nhắn SMS hoặc các thông báo khác.
* **Ghi nhật ký:** Ghi lại các sự kiện và lỗi để khắc phục sự cố và phân tích.

Mã

1. CitiesService.cs (Dịch vụ)
2. không gian tên Dịch vụ
3. {
4. Thành phố lớp công cộngDịch vụ
5. {
6. Danh sách riêng <chuỗi> \_cities;
8. Constructor
9. Dịch vụ thành phố công cộng()
10. {
11. \_cities = new List<string>() { "London", "Paris", "New York", "Tokyo", "Rome" };
12. }
14. Public List<string> GetCities()
15. {
16. trở lại \_cities;
17. }
18. }
19. }
21. HomeController.cs (Bộ điều khiển)
22. lớp công cộng HomeController: Bộ điều khiển
23. {
24. thành phố chỉ đọc riêng \_citiesService;
26. Constructor (chèn dịch vụ)
27. HomeController công cộng()
28. {
29. \_citiesService = CitiesService() mới;
30. }
32. [Tuyến ("/")]
33. Chỉ số IActionResult công khai()
34. {
35. Danh sách<chuỗi> thành phố = \_citiesService.GetCities();
36. trở lại View (thành phố); Chuyển dữ liệu vào chế độ xem
37. }
38. }

Lưu ý rằng trong ví dụ mã này, không có tính năng chèn phần phụ thuộc nào được sử dụng. Trong các dự án trong thế giới thực, bạn nên đăng ký các dịch vụ của bạn với vùng chứa chèn phụ thuộc tích hợp sẵn của ASP.NET Core và sau đó đưa chúng vào bộ điều khiển của bạn (hoặc các thành phần khác) thông qua constructor.

**Lời giải thích**

1. **Lớp CitiesService:** Lớp này đại diện cho một dịch vụ đơn giản chứa danh sách tên thành phố và cung cấp phương thức GetCities để truy xuất chúng.
2. **Lớp HomeController:**
   * **Phụ thuộc:** Nó phụ thuộc vào lớp CitiesService.
   * **Instantiation:** Trong ví dụ đơn giản này, đối tượng CitiesService được tạo trực tiếp trong constructor của controller.
   * **Phương thức hành động:** Phương thức hành động Index gọi phương thức GetCities của \_citiesService để truy xuất danh sách các thành phố và sau đó chuyển dữ liệu này đến chế độ xem.

**Các phương pháp hay nhất**

* **Nguyên tắc trách nhiệm duy nhất (SRP):** Thiết kế các dịch vụ của bạn để có một trách nhiệm duy nhất để giữ cho chúng tập trung và có thể duy trì.
* **Dependency Injection:** Sử dụng dependency injection để quản lý thời gian sử dụng dịch vụ và các phần phụ thuộc, giúp code của bạn được ghép nối lỏng lẻo và dễ kiểm tra hơn.
* **Lập trình dựa trên giao diện:** Xác định giao diện cho các dịch vụ của bạn để tạo các lớp trừu tượng và tạo điều kiện kiểm tra với các mô phỏng.
* **Đặt tên rõ ràng:** Sử dụng tên mô tả cho các dịch vụ của bạn và phương pháp của chúng để làm cho mã của bạn tự lập hồ sơ.
* **Thử nghiệm:** Viết kiểm thử đơn vị cho các dịch vụ của bạn để đảm bảo rằng logic nghiệp vụ của bạn hoạt động chính xác một cách riêng biệt.

**Những điểm chính cần nhớ**

* **Đóng gói:** Dịch vụ đóng gói "cách thức" logic ứng dụng của bạn, tách nó khỏi lớp trình bày.
* **Khả năng tái sử dụng:** Dịch vụ có thể được sử dụng lại trên nhiều bộ điều khiển.
* **Khả năng kiểm tra:** Các dịch vụ được thiết kế để dễ dàng kiểm tra đơn vị.
* **Dependency Injection:** Các dịch vụ thường được quản lý bởi container DI và được đưa vào controller.

Nguyên tắc đảo ngược phụ thuộc (DIP)

DIP là một nguyên tắc thiết kế thúc đẩy kiến trúc phần mềm được ghép nối lỏng lẻo. Nó nói rằng:

1. **Các mô-đun cấp cao không nên phụ thuộc vào các mô-đun cấp thấp.** Cả hai nên phụ thuộc vào trừu tượng.
2. **Trừu tượng hóa không nên phụ thuộc vào chi tiết.** Chi tiết nên phụ thuộc vào các khái niệm trừu tượng.

Nói một cách đơn giản hơn:

* Thay vì kết hợp chặt chẽ các lớp của bạn bằng cách để chúng phụ thuộc vào các triển khai cụ thể, chúng nên phụ thuộc vào các trừu tượng (giao diện hoặc các lớp trừu tượng).
* Điều này cho phép bạn dễ dàng hoán đổi các triển khai mà không cần thay đổi mã cấp cao hơn.

**Đảo ngược kiểm soát (IoC): Chuyển trách nhiệm**

IoC là một nguyên tắc rộng liên quan đến việc chuyển quyền kiểm soát việc tạo và quản lý đối tượng từ mã ứng dụng của bạn sang khung hoặc vùng chứa. Thay vì các lớp của bạn tạo ra các phần phụ thuộc của chúng một cách rõ ràng, chúng nhận chúng từ một nguồn bên ngoài. Nguồn bên ngoài này thường là một vùng chứa DI.

**Dependency Injection (DI): Công cụ thực tế**

DI là một thực hiện cụ thể của nguyên tắc IoC. Nó liên quan đến việc cung cấp các phụ thuộc cho một lớp từ bên ngoài chính lớp. Cách phổ biến nhất để thực hiện điều này trong ASP.NET Core là thông qua chèn hàm tạo, nhưng cũng có các kỹ thuật khác như chèn thuộc tính và chèn phương thức.

**Lợi ích của DIP, IoC và DI**

* **Loose Coupling:** Giảm sự phụ thuộc trực tiếp giữa các lớp, giúp chúng dễ dàng thay đổi và kiểm tra độc lập hơn.
* **Tính linh hoạt:** Bạn có thể dễ dàng hoán đổi các triển khai phụ thuộc khác nhau mà không ảnh hưởng đến lớp tiêu thụ.
* **Khả năng kiểm tra:** Kiểm thử đơn vị trở nên dễ dàng hơn nhiều, vì bạn có thể cung cấp các phần phụ thuộc giả để cô lập mã đang được kiểm tra.
* **Khả năng bảo trì:** Mã trở nên mô-đun hơn, dễ hiểu hơn và ít bị ảnh hưởng đến các thay đổi.

Mã

1. ServiceContracts (Giao diện)
2. không gian tên ServiceContracts
3. {
4. giao diện công cộng ICitiesService // Trừu tượng hóa CitiesService
5. {
6. Danh sách<chuỗi> GetCities();
7. }
8. }
10. Dịch vụ (Triển khai)
11. không gian tên Dịch vụ
12. {
13. lớp công cộng CitiesService : ICitiesService // CitiesService phụ thuộc vào trừu tượng hóa ICitiesService
14. {
15. // ... (Triển khai GetCities)...
16. }
17. }

Giao diện ICitiesService xác định trừu tượng hóa cho một dịch vụ có thể truy xuất danh sách các thành phố. Lớp CitiesService cung cấp việc triển khai cụ thể, nhưng nó phụ thuộc vào giao diện ICitiesService, không phải vào một lớp cụ thể.

Mã

1. Program.cs (hoặc Startup.cs)
2. chủ thầu. Services.Add(ServiceDescriptor mới(
3. typeof(ICitiesService), // Giao diện để đăng ký
4. typeof(CitiesService), // Triển khai cụ thể
5. ServiceLifetime.Transient // Tuổi thọ của dịch vụ (sẽ nói thêm về điều này sau)
6. ));
8. HomeController.cs (Bộ điều khiển)
9. lớp công cộng HomeController: Bộ điều khiển
10. {
11. ICitiesService chỉ đọc riêng \_citiesService; Phụ thuộc vào giao diện
13. Chèn hàm tạo
14. HomeController công cộng(ICitiesService citiesServiceService)
15. {
16. \_citiesService = citiesService;
17. }
19. // ... (Phương pháp hành động)...
20. }

Trong mã này:

1. **Đăng ký dịch vụ:** CitiesService được đăng ký trong vùng chứa DI bằng phương thức Add. ServiceDescriptor chỉ định:
   * Loại giao diện (ICitiesService) mà các thành phần khác sẽ yêu cầu.
   * Loại triển khai cụ thể (CitiesService) mà vùng chứa sẽ tạo.
   * Tuổi thọ của dịch vụ (ServiceLifetime.Transient có nghĩa là một phiên bản mới được tạo cho mỗi yêu cầu).
2. **Constructor Injection:** Constructor HomeController có một tham số thuộc loại ICitiesService. Điều này có nghĩa là vùng chứa DI sẽ tự động cung cấp một phiên bản của CitiesService khi bộ điều khiển được tạo.

**Ghi chú**

* **Trừu tượng:** Tập trung vào việc thiết kế các giao diện hoặc các lớp trừu tượng để đại diện cho các phần phụ thuộc của bạn.
* **Loose Coupling:** Các lớp của bạn nên phụ thuộc vào các khái niệm trừu tượng, không phải các triển khai cụ thể.
* **Dependency Injection:** Sử dụng vùng chứa DI (như vùng chứa được tích hợp trong ASP.NET Core) để quản lý và giải quyết các phần phụ thuộc.
* **Thời gian sử dụng dịch vụ:** Hiểu các tuổi thọ dịch vụ khác nhau (Tạm thời, Phạm vi, Singleton) và chọn tuổi thọ phù hợp cho từng dịch vụ.

Kịch bản - Công tắc đèn và bóng đèn

* **Không có DIP / IoC / DI:**
  + Hãy tưởng tượng một công tắc đèn truyền thống được nối trực tiếp với một bóng đèn cụ thể. Nếu bạn muốn thay bóng đèn sang một loại khác, bạn có thể cần phải nối lại công tắc vì nó được kết hợp chặt chẽ với bóng đèn ban đầu. Điều này tương tự như mã được ghép nối chặt chẽ, trong đó các lớp phụ thuộc trực tiếp vào việc triển khai cụ thể của các lớp khác.
* **Với DIP / IoC / DI:**
  + Bây giờ, hãy tưởng tượng một ổ cắm điện tiêu chuẩn và một phích cắm. Ổ cắm đại diện cho một giao diện (một trừu tượng), trong khi phích cắm đại diện cho một lớp thực hiện giao diện này. Bạn có thể cắm bất kỳ thiết bị tương thích nào (bóng đèn, quạt, v.v.) vào ổ cắm và nó sẽ hoạt động. Đây là bản chất của DIP - phụ thuộc vào các khái niệm trừu tượng, không phải thực hiện cụ thể.
  + "Sự đảo ngược của điều khiển" xuất hiện bởi vì ổ cắm (giao diện) quyết định hình dạng của phích cắm (triển khai), không phải ngược lại.
  + "Dependency injection" xảy ra khi bạn cắm thiết bị vào ổ cắm. Ổ cắm không tạo ra thiết bị; nó chỉ đơn giản là nhận nó và cho phép nó hoạt động.

**Biểu diễn trực quan**

1. Không có DIP / IoC / DI:
3. Công tắc đèn -----> bóng đèn cụ thể
4. (Ghép nối chặt chẽ)
6. Với DIP / IoC / DI:
8. Ổ cắm (Giao diện) <--- phích cắm (Triển khai)
9. |
10. |
11. Bóng đèn, Quạt, v.v.

**Tương tự mã**

1. Không có DIP
2. lớp LightSwitch
3. {
4. riêng tư SpecificLightBulb \_bulb = SpecificLightBulb() mới;
6. trống công cộng TurnOn()
7. {
8. \_củ. Chiếu sáng ();
9. }
10. }
12. Với DIP
13. giao diện ILight
14. {
15. void Illuminate();
16. }
18. lớp Bóng đèn : ILight { /\* ... \*/ }
20. lớp LightSwitch
21. {
22. ILight tư nhân \_light;
24. public LightSwitch (ILight light) // Chèn phụ thuộc
25. {
26. \_light = ánh sáng;
27. }
29. trống công cộng TurnOn()
30. {
31. \_ánh sáng. Chiếu sáng ();
32. }
33. }

Trong phiên bản DIP:

* LightSwitch phụ thuộc vào giao diện ILight, không phải một bóng đèn cụ thể.
* Lớp LightBulb triển khai ILight.
* Hàm khởi tạo LightSwitch nhận tham số ILight (chèn phụ thuộc). Bây giờ, bạn có thể truyền bất kỳ đối tượng nào triển khai ILight (một loại bóng đèn khác, quạt, v.v.) và LightSwitch sẽ hoạt động với nó.

**Những điểm chính**

* **DIP:** Phụ thuộc vào các khái niệm trừu tượng (giao diện) để làm cho mã của bạn linh hoạt và dễ bảo trì hơn.
* **IoC:** Cho phép một framework hoặc container quản lý việc tạo đối tượng và phụ thuộc.
* **DI:** Triển khai IoC bằng cách cung cấp các phụ thuộc cho các lớp của bạn (thường thông qua các hàm tạo).
* **Lợi ích:** Đạt được khớp nối lỏng lẻo, tính linh hoạt, khả năng kiểm tra và khả năng bảo trì trong mã của bạn.

Thời gian sử dụng dịch vụ

Khi bạn đăng ký một dịch vụ trong vùng chứa DI, bạn chỉ định thời gian tồn tại của dịch vụ đó. Điều này xác định cách vùng chứa DI tạo và quản lý các phiên bản của dịch vụ đó trong suốt quá trình thực thi ứng dụng của bạn.

**Ba tùy chọn trọn đời chính**

1. **Nhất thời:**
   * **Tạo:** Một phiên bản mới được tạo mỗi khi dịch vụ được yêu cầu (chèn).
   * **Trọn đời:** Phiên bản chỉ tồn tại miễn là cần thiết để thực hiện yêu cầu hiện tại.
   * **Cách sử dụng:** Lý tưởng cho các dịch vụ nhẹ, không trạng thái, trong đó mỗi yêu cầu yêu cầu một phiên bản mới.
   * **Ví dụ:** Ngữ cảnh cơ sở dữ liệu, trình ghi nhật ký, lớp trợ giúp.
2. **Phạm vi:**
   * **Tạo:** Một phiên bản duy nhất được tạo cho mỗi yêu cầu HTTP (hoặc phạm vi) trong ứng dụng của bạn.
   * **Trọn đời:** Phiên bản được chia sẻ trong suốt yêu cầu và được xử lý khi yêu cầu kết thúc.
   * **Cách sử dụng:** Tuổi thọ phổ biến nhất cho các ứng dụng web. Đảm bảo tính nhất quán trong một yêu cầu đồng thời tránh các đối tượng tồn tại lâu dài.
   * **Ví dụ:** Dữ liệu dành riêng cho người dùng, xử lý giao dịch, giỏ hàng.
3. **Singleton:**
   * **Tạo:** Một phiên bản duy nhất được tạo trong toàn bộ vòng đời của ứng dụng của bạn.
   * **Trọn đời:** Phiên bản được chia sẻ trên tất cả các yêu cầu và thành phần.
   * **Cách sử dụng:** Thích hợp cho các dịch vụ không trạng thái, bộ nhớ đệm, tác vụ nền hoặc cấu hình mà bạn muốn tải một lần và chia sẻ trên toàn cầu.
   * **Ví dụ:** Cài đặt cấu hình trên toàn ứng dụng, bộ nhớ đệm được chia sẻ, triển khai mẫu thiết kế singleton.

**Chọn trọn đời phù hợp**

Tuổi thọ bạn chọn cho một dịch vụ phụ thuộc vào mục đích và cách bạn định sử dụng dịch vụ đó:

* **Trạng thái:** Nếu dịch vụ của bạn giữ trạng thái cần duy nhất cho mỗi yêu cầu, hãy sử dụng Scoped. Nếu trạng thái cần được chia sẻ trên toàn cầu, hãy sử dụng Singleton. Nếu trạng thái không liên quan, Thoáng qua thường là đủ.
* **Sử dụng tài nguyên:** Các dịch vụ Singleton tiêu thụ bộ nhớ trong toàn bộ vòng đời ứng dụng, vì vậy hãy sử dụng chúng một cách thận trọng.
* **Đồng thời:** Lưu ý các vấn đề đồng thời khi sử dụng dịch vụ đơn lẻ trong môi trường đa luồng.

**Ví dụ về đăng ký**

1. Startup.cs (hoặc Program.cs)
2. chủ thầu. Services.AddTransient<ITransientService, TransientService>();
3. chủ thầu. Services.AddScoped<IScopedService, ScopedService>();
4. chủ thầu. Services.AddSingleton<ISingletonService, SingletonService>();

**Các phương pháp hay nhất trọn đời**

* **Ưu tiên Scoped cho ứng dụng web:** Trong hầu hết các trường hợp, Scoped là thời gian tồn tại được khuyến nghị cho các dịch vụ trong ứng dụng web.
* **Tránh phụ thuộc cố định:** Không chèn một dịch vụ có tuổi thọ ngắn hơn (ví dụ: Tạm thời) vào một dịch vụ tồn tại lâu hơn (ví dụ: Singleton). Điều này có thể dẫn đến hành vi không mong muốn và rò rỉ bộ nhớ.
* **Xem xét sự an toàn của luồng:** Nếu bạn sử dụng dịch vụ singleton, hãy đảm bảo dịch vụ đó an toàn với luồng nếu dịch vụ đó được truy cập đồng thời.

Kỹ thuật Dependency Injection trong ASP.NET Core

ASP.NET Core cung cấp một vùng chứa chèn phụ thuộc (DI) tích hợp mạnh mẽ cho phép bạn đưa các dịch vụ vào các thành phần của ứng dụng (bộ điều khiển, phần mềm trung gian, thành phần chế độ xem, v.v.). Dưới đây là những cách chính để bạn có thể chèn các phần phụ thuộc:

1. **Constructor Injection (Phổ biến nhất):**
   * **Cơ chế:** Các phụ thuộc được chuyển dưới dạng tham số đến hàm khởi tạo của lớp.
   * **Lợi ích:**
     + Dễ hiểu và dễ sử dụng.
     + Khuyến khích khớp nối lỏng lẻo và kiểm tra.
     + Đảm bảo rằng các phần phụ thuộc bắt buộc có sẵn trước khi lớp được sử dụng.
   * **Ví dụ:**
   * sản phẩm lớp công cộngNgười điều khiển : Người điều khiển
   * {
   * IProductService chỉ đọc riêng \_productService;
   * Sản phẩm công cộngBộ điều khiển (IProductService productServiceService)
   * {
   * \_productService = productService;
   * }
   * }
2. **Tiêm thuộc tính (ít phổ biến hơn):**
   * **Cơ chế:** Các phần phụ thuộc được gán cho các thuộc tính công khai với thuộc tính [FromServices].
   * **Lợi ích:**
     + Có thể hữu ích khi bạn có các phần phụ thuộc tùy chọn hoặc muốn tránh sự lộn xộn của hàm tạo.
     + Cho phép tải các phụ thuộc lười biếng.
   * **Ví dụ:**
   * lớp công khai MyMiddleware
   * {
   * [TừDịch vụ]
   * public ILogger<MyMiddleware> Logger { get; set; }
   * }
3. **Phương pháp tiêm (ít phổ biến nhất):**
   * **Cơ chế:** Các phụ thuộc được truyền dưới dạng tham số cho các phương thức riêng lẻ.
   * **Lợi ích:**
     + Cung cấp khả năng kiểm soát chi tiết về thời điểm các phần phụ thuộc được giải quyết.
     + Có thể hữu ích trong trường hợp bạn chỉ cần một phần phụ thuộc trong một phương thức cụ thể.
   * **Ví dụ:**
   * chỉ mục IActionResult công khai([FromServices] IUserService userService)
   * {
   * // ... sử dụng userService trong phương pháp này
   * }
   * Sử dụng kỹ thuật tiêm này nếu bạn yêu cầu một dịch vụ trong một hoặc một vài hành động.
4. **Phương pháp hành động tiêm:**

* **Cơ chế:** Đưa các dịch vụ trực tiếp vào các phương thức hành động dưới dạng tham số.
* **Lợi ích:**
  + Đơn giản hóa việc quản lý phụ thuộc trong các hành động cụ thể.
  + Hữu ích cho các tình huống chỉ cần phụ thuộc trong một phương thức hành động cụ thể.
* **Ví dụ:**

1. công khai IActionResult MyAction([FromServices] IMyService service)
2. {
3. // ... Sử dụng dịch vụ trong hành động này
4. }

**Lựa chọn kỹ thuật tiêm phù hợp**

* **Constructor Injection:** Cách tiếp cận được đề xuất và phổ biến nhất cho các phần phụ thuộc bắt buộc.
* **Chèn thuộc tính:** Sử dụng cho các phần phụ thuộc tùy chọn hoặc khi chèn hàm khởi tạo cồng kềnh.
* **Chèn phương thức:** Hãy xem xét điều này đối với các phần phụ thuộc chỉ cần thiết trong các phương thức cụ thể hoặc để kiểm soát tốt hơn đối với việc giải quyết phần phụ thuộc.
* **Action Method Injection:** Lý tưởng cho các tình huống chỉ cần phụ thuộc trong một phương thức hành động cụ thể.

**Những điểm chính cần nhớ**

* **Khớp nối lỏng lẻo:** Bất kể loại phun, nguyên tắc cốt lõi của DI là đạt được khớp nối lỏng lẻo giữa các thành phần.
* **Nguyên tắc đảo ngược phụ thuộc (DIP):** Đảm bảo rằng các lớp của bạn phụ thuộc vào các khái niệm trừu tượng (giao diện) hơn là các triển khai cụ thể.
* **Dependency Injection Container:** Container DI tích hợp của ASP.NET Core xử lý việc đăng ký và giải quyết các dịch vụ.
* **Thời gian sử dụng dịch vụ:** Hiểu các thời gian sử dụng dịch vụ khác nhau (Tạm thời, Phạm vi, Singleton) và chọn thời gian thích hợp cho từng phụ thuộc.

Các phương pháp hay nhất của DI

**1. Constructor Injection làm mặc định**

* **Lý do:** Chèn hàm khởi tạo là cách đơn giản và đáng tin cậy nhất để chèn các phần phụ thuộc. Nó đảm bảo rằng một lớp có tất cả các phụ thuộc cần thiết trước khi nó có thể được sử dụng, thúc đẩy tính hợp lệ của đối tượng.
* **Cách thực hiện:** Khai báo tất cả các phụ thuộc cần thiết dưới dạng tham số hàm khởi tạo.

1. Dịch vụ sản phẩm lớp công cộng: IProductService
2. {
3. IProductRepository chỉ đọc riêng \_productRepository;
4. private readonly ILogger<ProductService> \_logger;
6. public ProductService (IProductRepository productRepository, ILogger<ProductService> logger)
7. {
8. \_productRepository = productRepository;
9. \_logger = gỗ;
10. }
11. }

**2. Sử dụng giao diện cho các phụ thuộc**

* **Lý do:** Giao diện thúc đẩy khớp nối lỏng lẻo, cho phép bạn dễ dàng hoán đổi triển khai trong quá trình thử nghiệm hoặc khi sử dụng các môi trường khác nhau.
* **Cách thực hiện:** Xác định giao diện cho các dịch vụ của bạn và để các lớp của bạn phụ thuộc vào các giao diện, không phải triển khai cụ thể.

C#

1. giao diện công cộng IProductRepository { /\* ... \*/ }
2. lớp công cộng ProductRepository : IProductRepository { /\* ... \*/ }

**3. Tránh chống mẫu định vị dịch vụ**

* **Lý do:** Mẫu Service Locator liên quan đến việc truy cập trực tiếp vào vùng chứa DI từ bên trong các lớp của bạn (ví dụ: sử dụng IServiceProvider.GetService()). Điều này kết hợp chặt chẽ mã của bạn với vùng chứa DI và làm cho việc kiểm tra trở nên khó khăn hơn.
* **Cách thực hiện:** Thay vào đó, hãy yêu cầu vùng chứa DI chèn các phần phụ thuộc trực tiếp vào các lớp của bạn.

**4. Đăng ký các phần phụ thuộc ở gốc thành phần**

* **Lý do:** Gốc thành phần (thường là tệp Program.cs hoặc tệp Startup.cs) là nơi bạn nên định cấu hình vùng chứa DI của mình. Điều này tập trung đăng ký phần phụ thuộc và giúp quản lý và hiểu cấu trúc ứng dụng của bạn dễ dàng hơn.
* **Cách thực hiện:** Sử dụng các phương thức IServiceCollection.Add\* (ví dụ: AddTransient, AddScoped, AddSingleton) để đăng ký các dịch vụ của bạn và thời gian tồn tại của chúng.

**5. Chọn tuổi thọ dịch vụ phù hợp**

* **Tạm thời:** Một phiên bản mới được tạo mỗi khi dịch vụ được yêu cầu.
* **Phạm vi:** Một phiên bản duy nhất được tạo cho mỗi yêu cầu.
* **Singleton:** Một phiên bản duy nhất được tạo trong toàn bộ vòng đời của ứng dụng.
* **Cách thực hiện:** Xem xét cẩn thận bản chất của dịch vụ của bạn (stateful so với stateless) và các mẫu sử dụng của dịch vụ để chọn thời gian tồn tại phù hợp.

**6. Tránh sự phụ thuộc bị giam giữ**

* **Lý do:** Phần phụ thuộc cố định xảy ra khi bạn chèn một dịch vụ có thời gian tồn tại ngắn hơn (ví dụ: Tạm thời) vào một dịch vụ có tuổi thọ cao hơn (ví dụ: Singleton). Điều này có thể dẫn đến hành vi không mong muốn và rò rỉ bộ nhớ.
* **Cách thực hiện:** Đảm bảo rằng thời gian sử dụng dịch vụ của bạn tương thích và bạn không vô tình nắm bắt một phiên bản tạm thời trong một đơn thử.

**7. Sử dụng các công cụ trang trí để thêm các mối quan tâm xuyên suốt**

* **Lý do:** Trình trang trí bao bọc các dịch vụ hiện có và cho phép bạn thêm hành vi bổ sung (ví dụ: ghi nhật ký, bộ nhớ đệm) mà không cần sửa đổi dịch vụ gốc.
* **Cách thực hiện:** Triển khai cùng một giao diện với dịch vụ bạn muốn trang trí và đưa dịch vụ gốc vào trình trang trí.

**8. Tận dụng mẫu tùy chọn để cấu hình**

* **Lý do:** Mẫu Tùy chọn cung cấp một cách nhập mạnh để truy cập cài đặt cấu hình trong dịch vụ của bạn.
* **Cách thực hiện:** Tạo các lớp đại diện cho các phần cấu hình của bạn và sử dụng giao diện IOptions để chèn chúng.

**9. Xem xét DI thuần túy để có khả năng kiểm tra**

* **Lý do:** DI thuần túy (tránh hoàn toàn IServiceProvider) làm cho các lớp của bạn dễ kiểm tra hơn, vì bạn có thể dễ dàng cung cấp các phụ thuộc giả trong quá trình kiểm thử đơn vị.
* **Cách thực hiện:** Thiết kế các lớp của bạn sao cho tất cả các phần phụ thuộc của chúng được truyền qua constructor hoặc các điểm tiêm khác.

**10. Đừng lạm dụng DI**

* **Lý do:** Dependency injection là một công cụ mạnh mẽ, nhưng nó nên được sử dụng một cách thận trọng. Việc lạm dụng nó có thể dẫn đến các biểu đồ đối tượng phức tạp và khiến mã khó suy luận hơn.
* **Cách thực hiện:** Đừng chèn mọi lớp vào ứng dụng của bạn. Sử dụng DI cho các dịch vụ và thành phần có sự phụ thuộc rõ ràng và nơi bạn cần tính linh hoạt và khả năng kiểm tra.

Giao diện tự động

Trong khi ASP.NET Core có vùng chứa chèn phụ thuộc (DI) tích hợp, Autofac là vùng chứa IoC phổ biến của bên thứ ba được biết đến với tính linh hoạt, các tính năng nâng cao và các tùy chọn tùy chỉnh. Nó tích hợp liền mạch với ASP.NET Core, cung cấp cho bạn các công cụ mạnh mẽ hơn để quản lý các phụ thuộc của bạn.

**Ưu điểm chính của Autofac**

* **Tính linh hoạt:** Cung cấp phạm vi tuổi thọ của thành phần và các tùy chọn đăng ký rộng hơn so với container tích hợp.
* **Tùy chỉnh:** Cung cấp khả năng kiểm soát chi tiết hơn về cách giải quyết và quản lý các phần phụ thuộc.
* **Các tính năng nâng cao:** Hỗ trợ các tính năng như đăng ký dựa trên mô-đun, chèn tài sản, quét lắp ráp và chặn (đối với các mối quan tâm cắt ngang).
* **Hiệu suất:** Thường được coi là có hồ sơ hiệu suất tốt.

**Tích hợp Autofac với ASP.NET Core**

1. **Cài đặt gói:** Thêm gói Autofac.Extensions.DependencyInjection NuGet vào dự án của bạn.
2. **Định cấu hình vùng chứa:** Trong Program.cs của bạn (hoặc Startup.cs trong các phiên bản cũ hơn), hãy thay thế nhà cung cấp dịch vụ mặc định bằng Autofac:
   1. chủ thầu. Host.UseServiceProviderFactory(AutofacServiceProviderFactory());
3. **Đăng ký dịch vụ:** Sử dụng trình tạo. Host.ConfigureContainer<ContainerBuilder> để truy cập ContainerBuilder của Autofac và đăng ký dịch vụ của bạn: C#
   1. chủ thầu. Host.ConfigureContainer<ContainerBuilder>(containerBuilder =>
   2. {
   3. Logic đăng ký Autofac của bạn tại đây
   4. });

Mã

1. Program.cs
2. // ... hàng nhập khẩu khác...
4. chủ thầu. Host.UseServiceProviderFactory(AutofacServiceProviderFactory()); Sử dụng Autofac
5. chủ thầu. Services.AddControllersWithViews(); Thêm dịch vụ MVC
7. chủ thầu. Host.ConfigureContainer<ContainerBuilder>(containerBuilder =>
8. {
9. containerBuilder.RegisterType<CitiesService>(). Như<ICitiesService>(). InstancePerLifetimeScope (); Đăng ký CitiesService theo phạm vi
10. });
12. var app = trình tạo. Xây dựng();
13. // ... phần còn lại của mã ...

Trong mã này:

1. **UseServiceProviderFactory:** Dòng này yêu cầu ASP.NET Core sử dụng Autofac làm nhà cung cấp dịch vụ.
2. **AddControllersWithViews:** Điều này đăng ký các dịch vụ cần thiết cho MVC (mô hình, chế độ xem, bộ điều khiển).
3. **ConfigureContainer:** Biểu thức lambda này cung cấp cho bạn quyền truy cập vào ContainerBuilder của Autofac.
4. **RegisterType<CitiesService>(). Như<ICitiesService>(). InstancePerLifetimeScope();:** Thao tác này đăng ký lớp CitiesService như một triển khai của giao diện ICitiesService với thời gian tồn tại có phạm vi.

**Phương thức đăng ký Autofac**

* **RegisterType<T>():** Đăng ký một loại cụ thể.
* **As<T>():** Chỉ định giao diện hoặc kiểu cơ sở mà loại đã đăng ký sẽ được phân giải thành.
* **Phạm vi trọn đời:**
  + InstancePerDependency() (tương đương với Transient)
  + InstancePerLifetimeScope() (tương đương với Scoped)
  + SingleInstance() (tương đương với Singleton)

**Ghi chú**

* **Tại sao nên chọn Autofac?**
  + Linh hoạt hơn và kiểm soát giải quyết phần phụ thuộc.
  + Các tính năng bổ sung (mô-đun, tiêm thuộc tính, v.v.).
* **Tích hợp:** Thay thế nhà máy nhà cung cấp dịch vụ mặc định bằng Autofac.
* **Đăng ký:** Sử dụng cú pháp của Autofac (RegisterType, As, phạm vi trọn đời) trong lambda ConfigureContainer.
* **Các khái niệm quen thuộc:** Các khái niệm cơ bản của DI (trừu tượng, tuổi thọ) vẫn giữ nguyên, chỉ với cú pháp khác nhau.

Phạm vi dịch vụ

Trong ASP.NET Core DI, phạm vi dịch vụ là ranh giới logic xác định tuổi thọ của các dịch vụ được đăng ký là *Phạm vi*. Khi một phạm vi được tạo, vùng chứa DI sẽ khởi tạo bất kỳ dịch vụ có phạm vi nào được yêu cầu trong phạm vi đó. Các phiên bản dịch vụ có phạm vi này sau đó được chia sẻ trên tất cả các thành phần trong phạm vi đó, đảm bảo tính nhất quán và tránh tạo đối tượng không cần thiết.

**Phạm vi dịch vụ hoạt động như thế nào trong ASP.NET Core**

1. **Phạm vi yêu cầu (Mặc định):** Trong các ứng dụng web ASP.NET Core, phạm vi phổ biến nhất là *phạm vi yêu cầu*. Một phạm vi mới được tự động tạo ở đầu mỗi yêu cầu HTTP. Tất cả các dịch vụ có phạm vi được giải quyết từ phạm vi yêu cầu này và vẫn tồn tại trong toàn bộ chu kỳ yêu cầu-phản hồi. Sau khi yêu cầu được xử lý, phạm vi sẽ được xử lý và tất cả các dịch vụ có phạm vi trong đó cũng sẽ được xử lý.
2. **Tạo phạm vi rõ ràng:** Bạn cũng có thể tạo phạm vi tùy chỉnh theo cách thủ công. Điều này hữu ích trong các trường hợp bạn cần thời gian tồn tại có phạm vi cho các hoạt động không tương ứng trực tiếp với yêu cầu HTTP (ví dụ: tác vụ nền, kiểm thử đơn vị). Bạn có thể tạo phạm vi bằng phương thức IServiceProvider.CreateScope().

Mã

1. sử dụng (var scope = provider. CreateScope())
2. {
3. var scopedService = phạm vi. ServiceProvider.GetRequiredService<IScopedService>();
4. Sử dụng scopedService trong phạm vi này
5. }

**Tuổi thọ của các dịch vụ có phạm vi**

* **Tạo:** Một phiên bản mới của dịch vụ có phạm vi được tạo vào lần đầu tiên được yêu cầu trong phạm vi.
* **Chia sẻ:** Các yêu cầu tiếp theo cho cùng một dịch vụ có phạm vi trong cùng một phạm vi sẽ nhận được cùng một phiên bản.
* **Xử lý:** Khi phạm vi được xử lý (ví dụ: khi kết thúc yêu cầu HTTP), tất cả các dịch vụ có phạm vi trong phạm vi đó cũng sẽ được xử lý.

**Lợi ích của phạm vi dịch vụ**

* **Quản lý trạng thái:** Các dịch vụ có phạm vi hoàn hảo để quản lý trạng thái cần tồn tại trong suốt một yêu cầu nhưng không được rò rỉ trên các yêu cầu khác nhau.
* **Sử dụng tài nguyên hiệu quả:** Phạm vi đảm bảo rằng bạn không tạo ra các phiên bản dịch vụ không cần thiết, dẫn đến quản lý bộ nhớ tốt hơn.
* **Tính nhất quán:** Các dịch vụ có phạm vi cung cấp chế độ xem nhất quán về dữ liệu và trạng thái trong một yêu cầu duy nhất.

**Các tình huống phổ biến cho các dịch vụ có phạm vi**

* **Ngữ cảnh cơ sở dữ liệu (EF Core):** Một phiên bản ngữ cảnh cơ sở dữ liệu mới thường được tạo cho mỗi yêu cầu để đảm bảo cách ly dữ liệu và tránh các vấn đề đồng thời.
* **Dữ liệu dành riêng cho người dùng:** Các dịch vụ chứa dữ liệu cụ thể cho người dùng hiện tại (ví dụ: giỏ hàng) thường được giới hạn trong phạm vi yêu cầu.
* **Ghi nhật ký bằng ngữ cảnh:** Nếu bạn cần ghi nhật ký thông tin với ngữ cảnh yêu cầu cụ thể, thì trình ghi nhật ký có phạm vi sẽ có lợi.
* **Giao dịch:** Nếu bạn cần duy trì tính toàn vẹn của giao dịch trong một yêu cầu, bạn có thể sử dụng dịch vụ có phạm vi để quản lý giao dịch.

**Những cân nhắc quan trọng**

* **Tránh phụ thuộc cố định:** Hãy thận trọng khi đưa một dịch vụ có phạm vi vào một dịch vụ đơn lẻ. Điều này có thể dẫn đến hành vi không mong muốn và rò rỉ bộ nhớ vì dịch vụ có phạm vi sẽ được duy trì trong toàn bộ vòng đời của ứng dụng.
* **Xử lý rõ ràng:** Khi bạn tạo phạm vi tùy chỉnh, hãy nhớ xử lý chúng đúng cách bằng cách sử dụng câu lệnh using hoặc bằng cách gọi Dispose() theo cách thủ công trên đối tượng IServiceScope.

Những điểm chính cần nhớ

**Nguyên tắc đảo ngược phụ thuộc (DIP)**

* **Ý tưởng cốt lõi:** Các mô-đun cấp cao không nên phụ thuộc vào các mô-đun cấp thấp; cả hai đều nên phụ thuộc vào các trừu tượng (giao diện/lớp trừu tượng).
* **Mục tiêu:** Khớp nối lỏng lẻo, linh hoạt, khả năng kiểm tra.

**Đảo ngược kiểm soát (IoC)**

* **Ý tưởng cốt lõi:** Chuyển quyền kiểm soát việc tạo và quản lý đối tượng từ mã của bạn sang khung hoặc vùng chứa (ví dụ: vùng chứa DI).
* **Mục tiêu: Tách** rời, cấu hình đơn giản hóa, cải thiện khả năng kiểm tra.

**Chèn phụ thuộc (DI)**

* **Ý tưởng cốt lõi:** Các phụ thuộc được cung cấp (đưa vào) vào một lớp từ một nguồn bên ngoài (thường là một vùng chứa DI).
* **Các loại trong ASP.NET Core:**
  + **Chèn hàm khởi tạo:** Các phần phụ thuộc được truyền dưới dạng tham số hàm khởi tạo (phổ biến nhất).
  + **Chèn thuộc tính:** Các phần phụ thuộc được gán cho các thuộc tính có thuộc tính [FromServices].
  + **Chèn phương thức:** Các phần phụ thuộc được chuyển dưới dạng tham số phương thức.
* **Lợi ích:**
  + Khớp nối lỏng lẻo
  + Linh hoạt
  + Khả năng kiểm tra
  + Khả năng bảo trì

**Tuổi thọ sử dụng trong ASP.NET Core DI**

* **Tạm thời:** Một phiên bản mới được tạo mỗi khi yêu cầu dịch vụ.
* **Phạm vi:** Một phiên bản duy nhất được tạo cho mỗi yêu cầu/phạm vi.
* **Singleton:** Một phiên bản duy nhất được tạo một lần và được chia sẻ trong suốt vòng đời của ứng dụng.

**Autofac: Vùng chứa IoC mạnh mẽ**

* **Mục đích:** Thay thế cho vùng chứa ASP.NET Core DI tích hợp.
* **Lợi ích:**
  + Đăng ký linh hoạt hơn và các tùy chọn tuổi thọ
  + Các tính năng nâng cao (mô-đun, tiêm thuộc tính, đánh chặn)
  + Hiệu suất tốt
* **Tích hợp:**
  + Cài đặt Autofac.Extensions.DependencyInjection.
  + Sử dụng trình tạo. Host.UseServiceProviderFactory(new AutofacServiceProviderFactory()) trong Program.cs.
  + Đăng ký dịch vụ trong trình xây dựng. Host.ConfigureContainer<ContainerBuilder>.

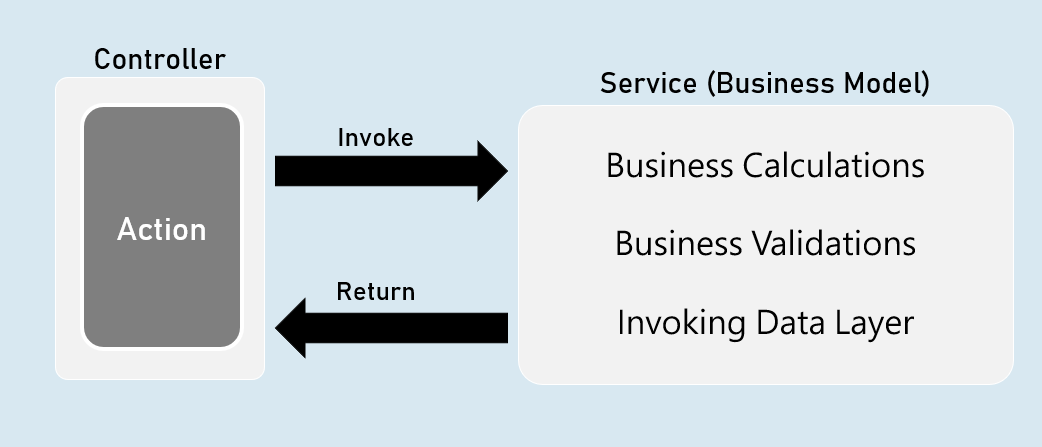
**Đăng ký Autofac**

* containerBuilder.RegisterType<T>(). Như<TInterface>(). InstancePerLifetimeScope ();
  + Tương đương với dịch vụ. AddScoped<TInterface, T>().
* InstancePerDependency() (Tạm thời), SingleInstance() (Singleton)

**Mẹo phỏng vấn**

* **Hiểu biết khái niệm:** Có thể giải thích các nguyên tắc của DIP, IoC và DI cũng như cách chúng liên quan đến nhau.
* **Ứng dụng thực tế:** Thể hiện khả năng của bạn để chọn tuổi thọ phù hợp cho một dịch vụ nhất định và giải thích ý nghĩa của từng lựa chọn.
* **Các phương pháp hay nhất:** Thảo luận về những lợi thế của việc sử dụng giao diện và chèn hàm tạo.
* **Autofac:** Làm nổi bật những lợi ích của việc sử dụng Autofac so với vùng chứa tích hợp và thể hiện kiến thức của bạn về cú pháp đăng ký của nó.
* **Khắc phục sự cố:** Giải thích cách bạn chẩn đoán các vấn đề DI phổ biến (ví dụ: phụ thuộc tuần hoàn, thời gian tồn tại không chính xác).

Dịch vụ



'Dịch vụ' là một lớp chứa logic nghiệp vụ như tính toán kinh doanh, xác thực kinh doanh cụ thể cho miền doanh nghiệp của khách hàng.

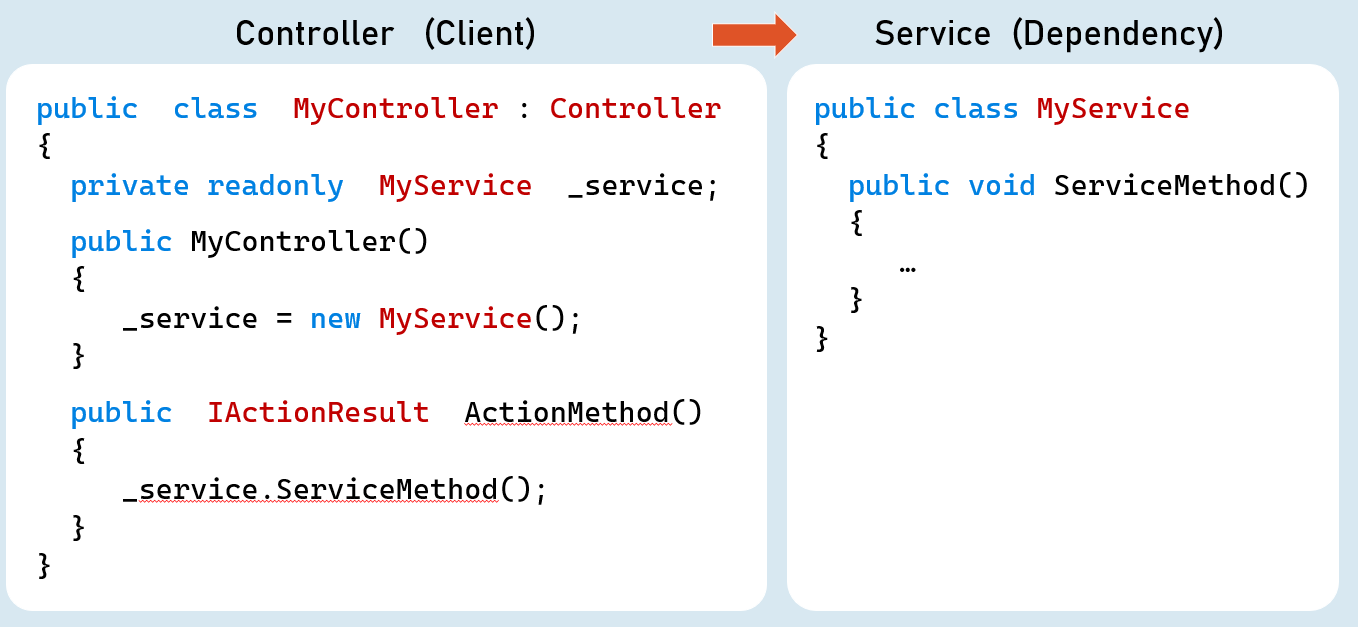
Service là một lớp trừu tượng (lớp giữa) giữa lớp trình bày (hoặc lớp ứng dụng) và lớp dữ liệu.

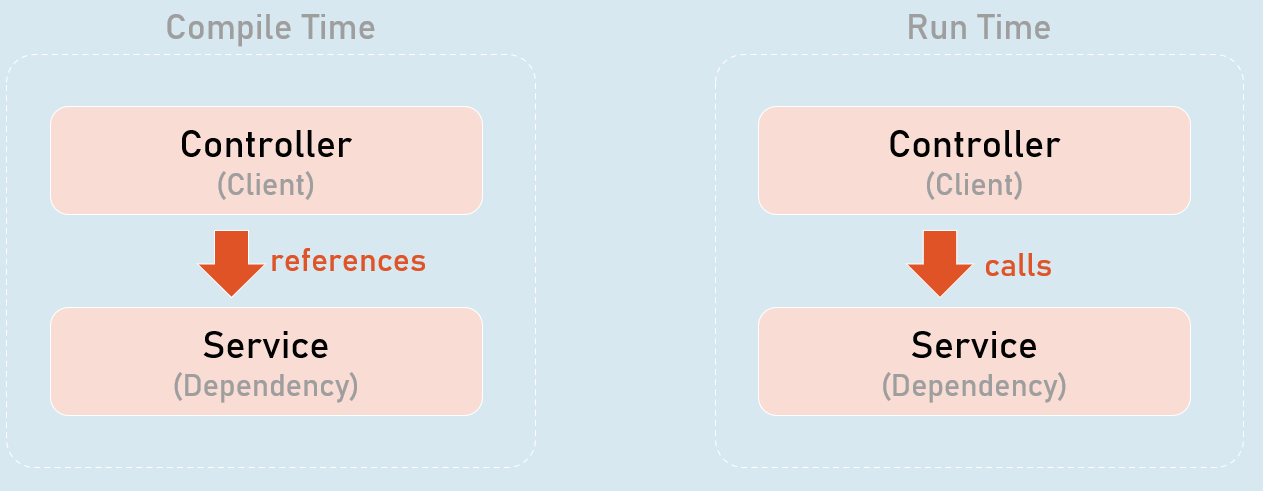
Nó làm cho logic nghiệp vụ tách biệt khỏi lớp trình bày và lớp dữ liệu.

Nó làm cho logic nghiệp vụ có thể kiểm tra đơn vị một cách dễ dàng.

Sẽ được gọi bởi bộ điều khiển.

Phụ thuộc trực tiếp





Các mô-đun cấp cao hơn phụ thuộc vào các mô-đun cấp thấp hơn.

Vấn đề phụ thuộc

Các mô-đun cấp cao hơn phụ thuộc vào các mô-đun cấp thấp hơn.

* Có nghĩa là, cả hai đều được kết hợp chặt chẽ.
* Nhà phát triển mô-đun cấp cao hơn NÊN ĐỢI cho đến khi hoàn thành việc phát triển mô-đun cấp thấp hơn.
* Yêu cầu nhiều thay đổi mã để hoán đổi một mô-đun cấp thấp hơn thay thế.
* Bất kỳ thay đổi nào được thực hiện trong mô-đun cấp thấp hơn sẽ ảnh hưởng đến thay đổi trong mô-đun cấp cao hơn.
* Khó kiểm tra một mô-đun mà không ảnh hưởng / kiểm tra mô-đun khác.

Nguyên tắc đảo ngược phụ thuộc

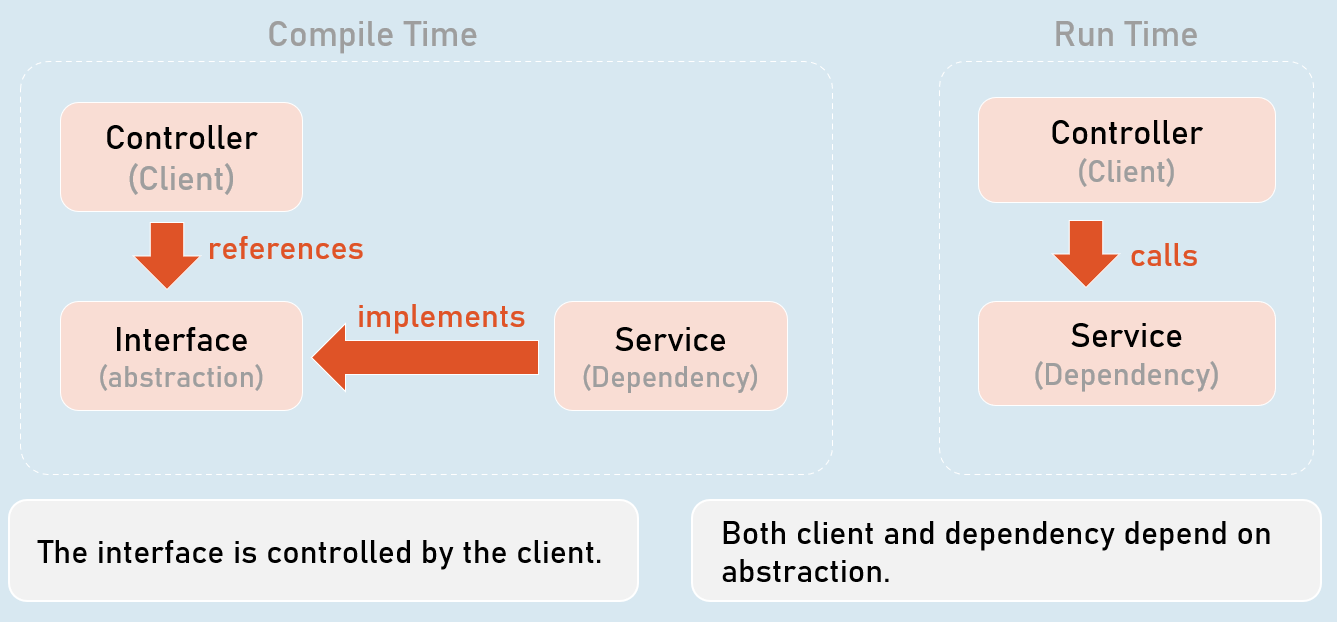
Nguyên tắc đảo ngược phụ thuộc (DIP) là một nguyên tắc thiết kế (hướng dẫn), là một giải pháp cho vấn đề phụ thuộc.

"Các mô-đun cấp cao hơn (máy khách) KHÔNG NÊN phụ thuộc vào các mô-đun cấp thấp (phụ thuộc).

Cả hai nên phụ thuộc vào các trừu tượng (giao diện hoặc lớp trừu tượng)."

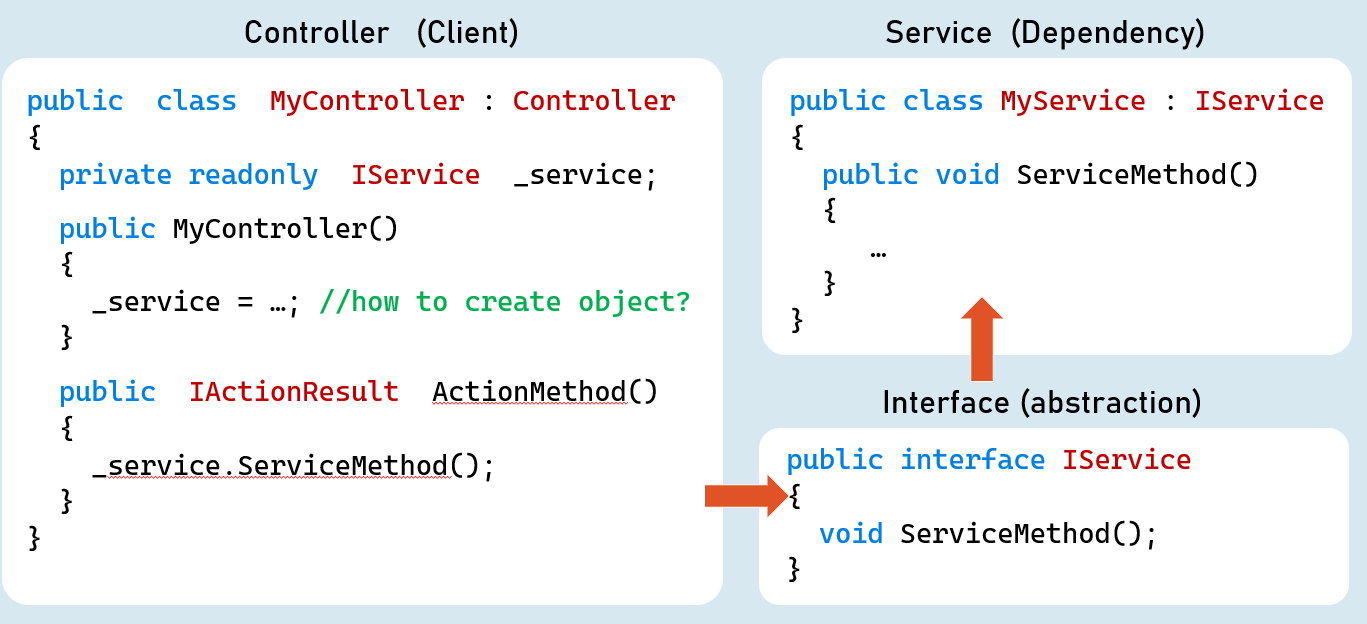
"Trừu tượng hóa không nên phụ thuộc vào chi tiết (cả khách hàng và phụ thuộc).

Các chi tiết (cả khách hàng và phụ thuộc) nên phụ thuộc vào các khái niệm trừu tượng."



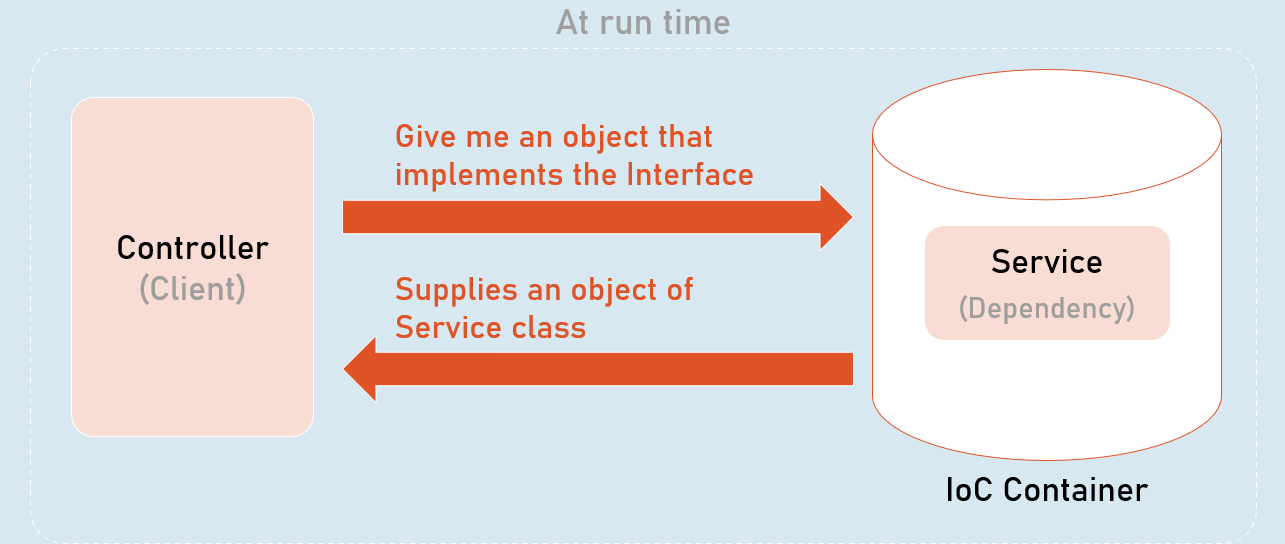
Giao diện được điều khiển bởi máy khách.

Cả máy khách và phụ thuộc đều phụ thuộc vào trừu tượng.



Đảo ngược kiểm soát (IoC)

* Đảo ngược kiểm soát (IoC) là một mẫu thiết kế (giải pháp có thể tái sử dụng cho một vấn đề phổ biến), đề xuất "vùng chứa IoC" để thực hiện Nguyên tắc đảo ngược phụ thuộc (DIP).
* Nó đảo ngược điều khiển bằng cách chuyển điều khiển sang vùng chứa IoC.
* "Đừng gọi cho chúng tôi, chúng tôi sẽ gọi cho bạn".
* Nó có thể được thực hiện bởi các mẫu thiết kế khác như sự kiện, công cụ định vị dịch vụ, chèn phụ thuộc, v.v.

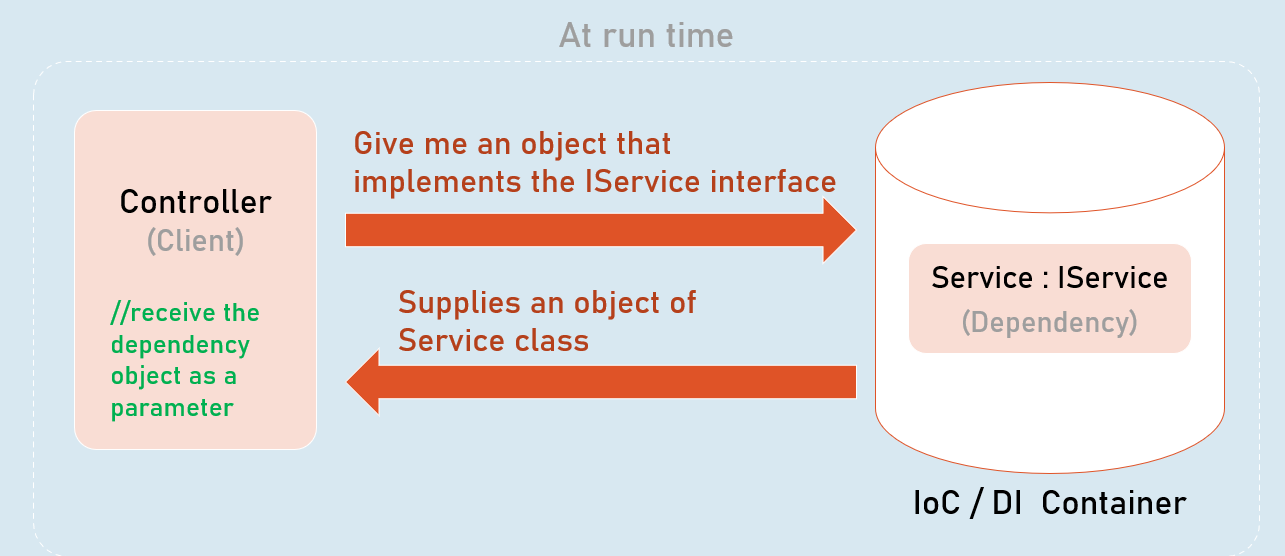


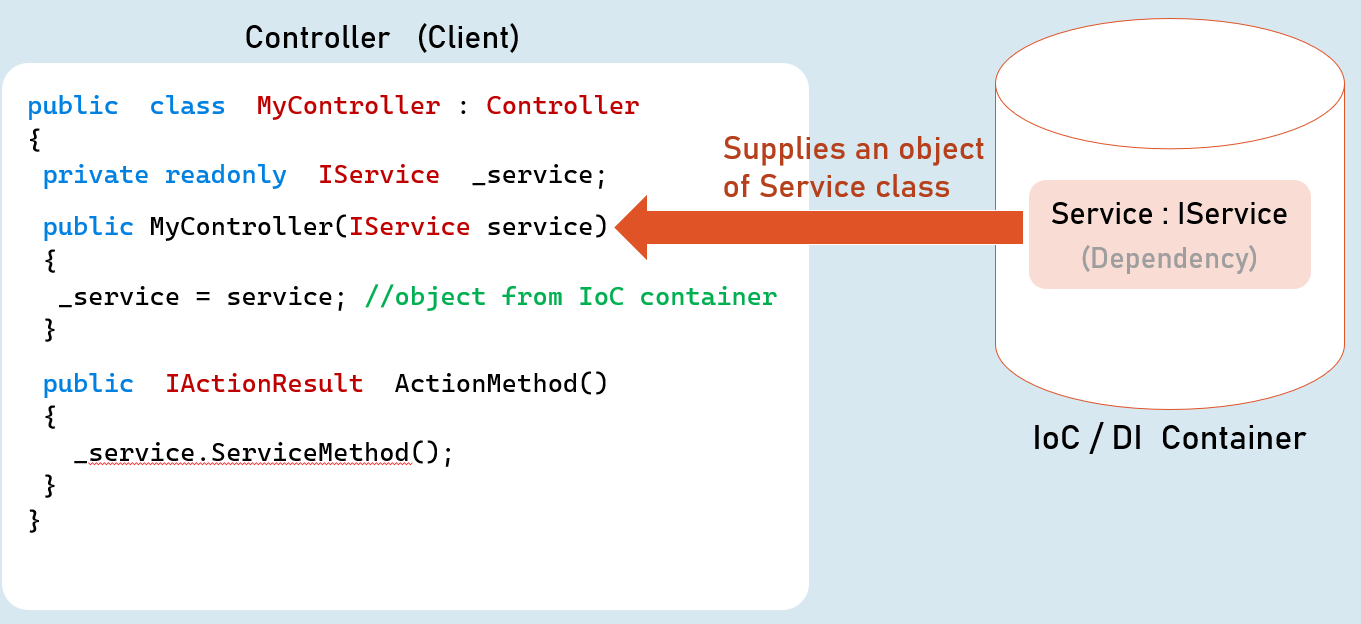
Tất cả các phần phụ thuộc nên được thêm vào IServiceCollection (hoạt động như vùng chứa IoC).

1. chủ thầu. Services.Add(
2. ServiceDescriptor mới(
3. typeof (Giao diện),
4. typeof (Dịch vụ)
5. ServiceLifetime.LifeTime // Thoáng qua, Phạm vi, Singleton
6. )
7. );

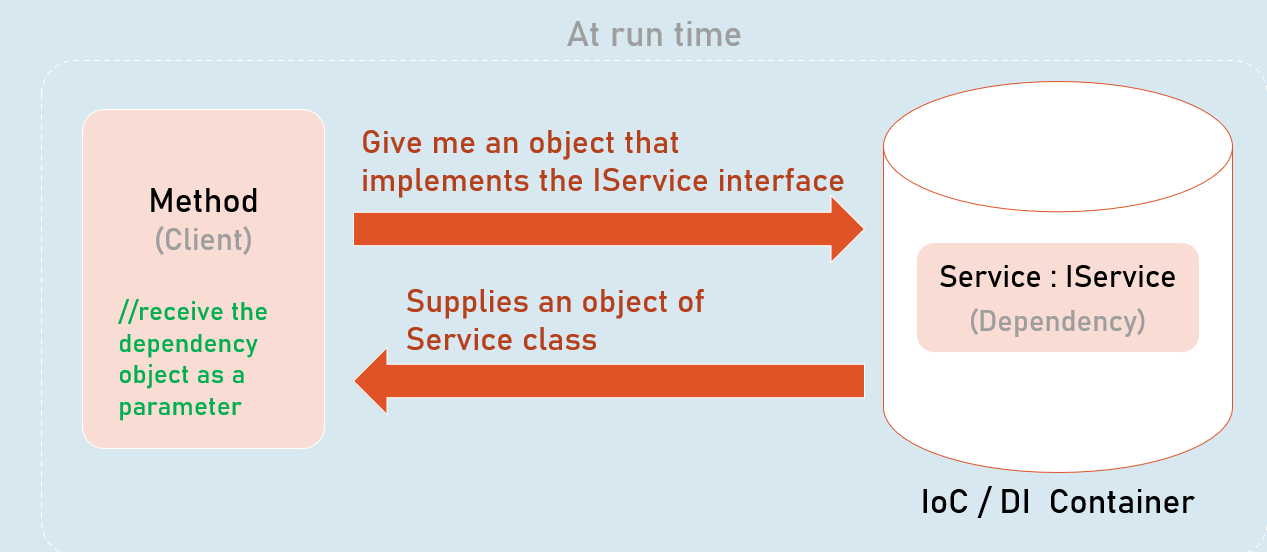
Chèn phụ thuộc (DI)

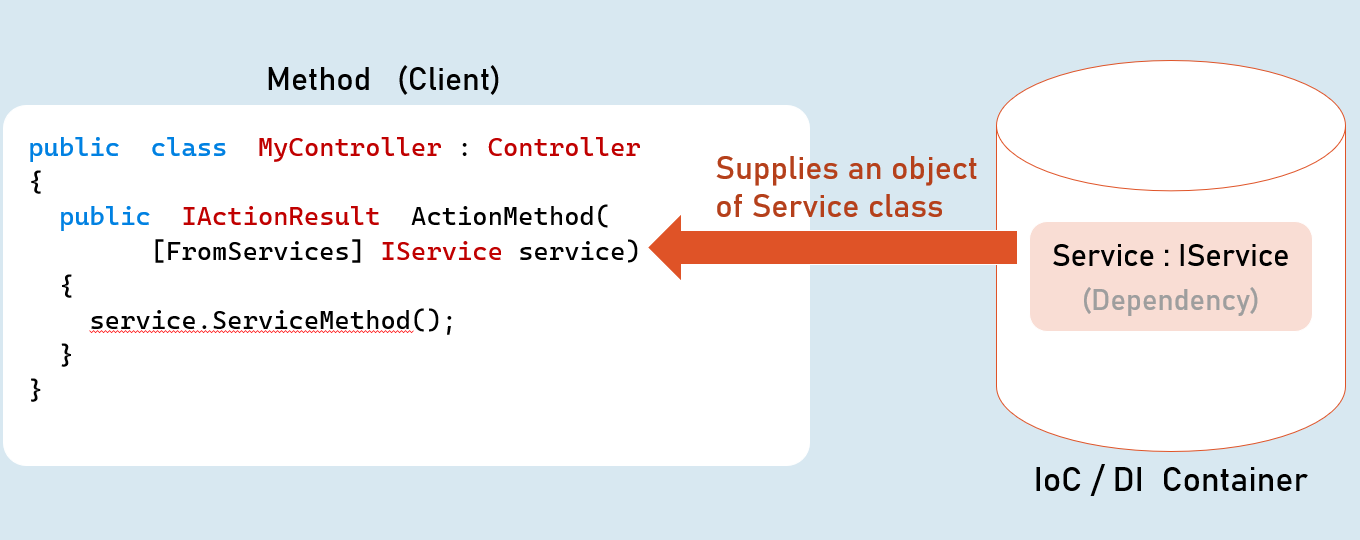
* Dependency injection (DI) là một mẫu thiết kế, là một kỹ thuật để đạt được "Inversion of Control (IoC)" giữa máy khách và các phụ thuộc của họ.
* Nó cho phép bạn đưa (cung cấp) một đối tượng triển khai cụ thể của một thành phần cấp thấp vào một thành phần cấp cao.
* Lớp máy khách nhận đối tượng phụ thuộc dưới dạng tham số trong hàm khởi tạo hoặc trong một phương thức.





Phương pháp tiêm



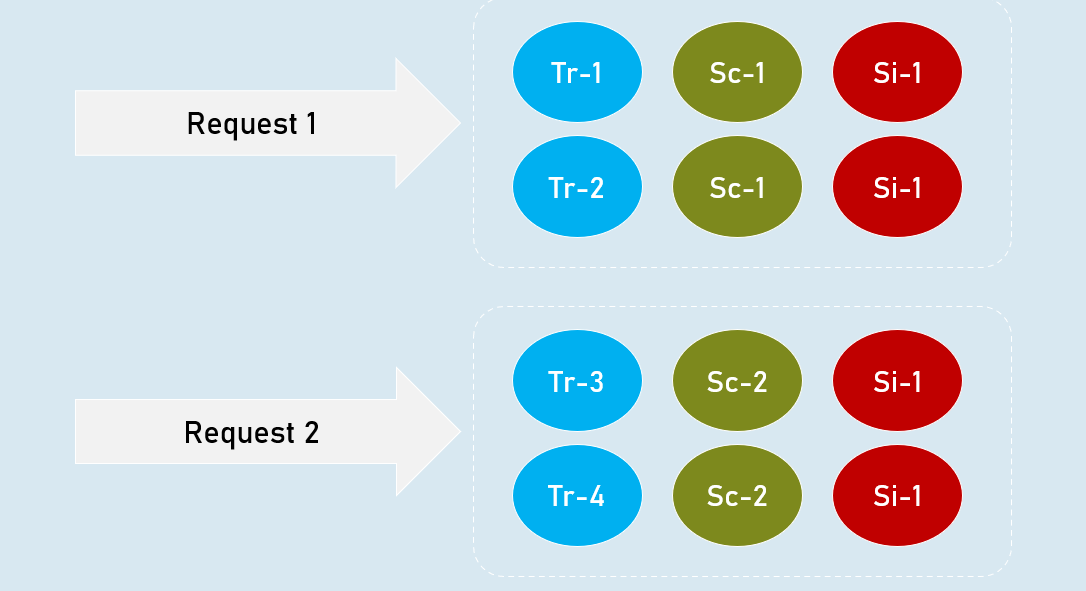


Tuổi thọ dịch vụ

(Quá độ, Phạm vi, Singleton)

Tuổi thọ dịch vụ cho biết khi nào một đối tượng mới của dịch vụ phải được tạo bởi vùng chứa IoC / DI.

1. **Quá độ:** Mỗi lần tiêm
2. **Phạm vi:** Mỗi phạm vi (yêu cầu trình duyệt)
3. **Singleton:** Trong toàn bộ vòng đời ứng dụng.



**Nhất thời**

Các đối tượng dịch vụ trọn đời tạm thời được tạo mỗi khi chúng được tiêm.

Các phiên bản dịch vụ được xử lý ở cuối phạm vi (thường là yêu cầu trình duyệt)

**Phạm vi**

Các đối tượng dịch vụ trọn đời có phạm vi được tạo một lần cho mỗi phạm vi (thường là một yêu cầu trình duyệt).

Các phiên bản dịch vụ được xử lý ở cuối phạm vi (thường là yêu cầu trình duyệt).

**Singleton**

Các đối tượng dịch vụ trọn đời Singleton được tạo lần đầu tiên khi được yêu cầu.

Các phiên bản dịch vụ được xử lý khi tắt ứng dụng.

**Nhất thời**

chủ thầu. Services.AddTransient<IService, Dịch vụ>(); Dịch vụ tạm thời

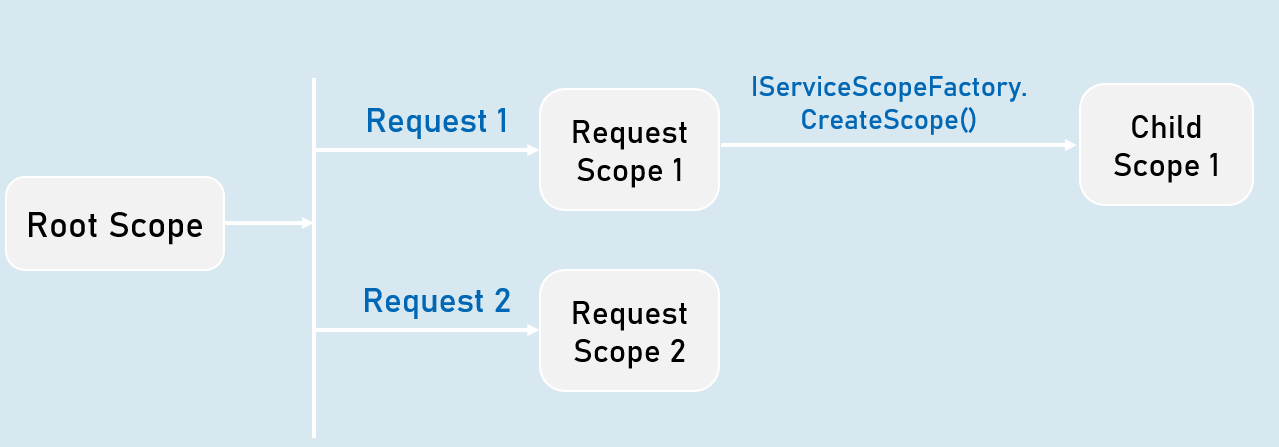
**Phạm vi**

chủ thầu. Services.AddScoped<IService, Service>(); Dịch vụ có phạm vi

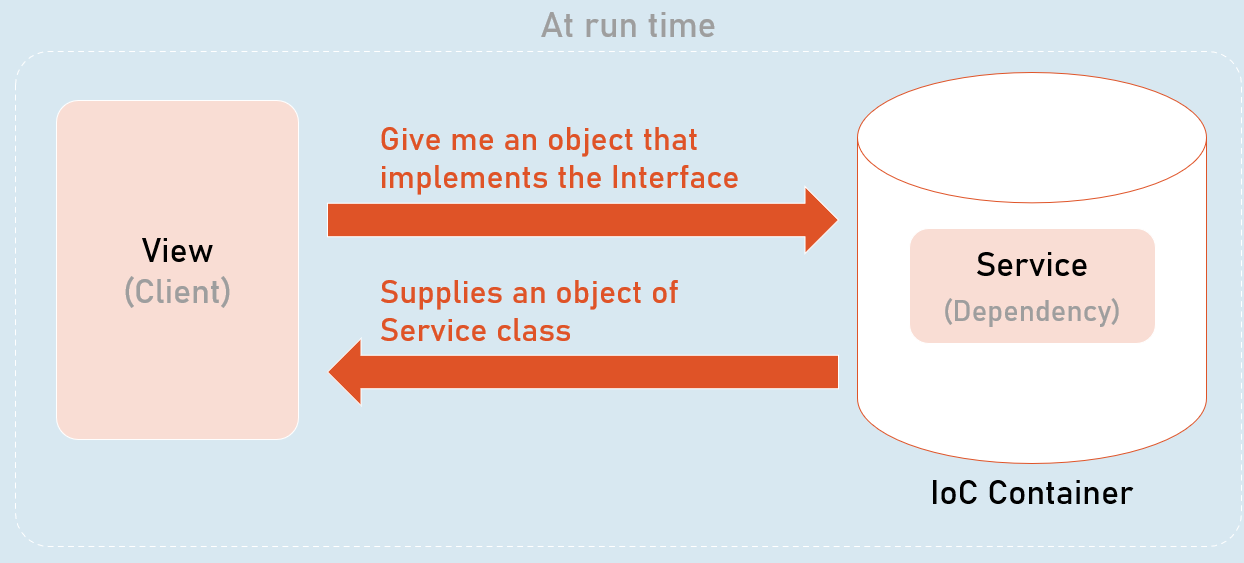
**Singleton**

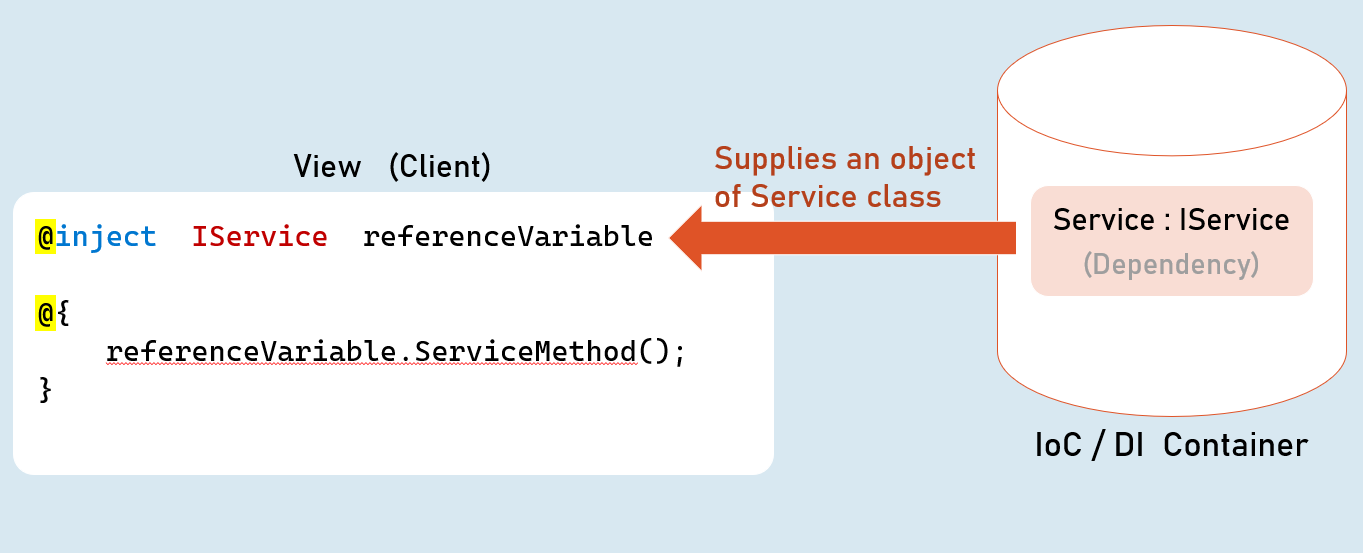
chủ thầu. Services.AddSingleton<IService, Dịch vụ>(); Dịch vụ Singleton

Phạm vi dịch vụ



Xem tiêm





Các phương pháp hay nhất trong DI

**Trạng thái toàn cầu trong dịch vụ**

Tránh sử dụng các **lớp tĩnh** để lưu trữ một số dữ liệu trên toàn cầu cho tất cả người dùng / tất cả các yêu cầu.

Bạn có thể sử dụng  **dịch vụ Singleton** cho các tình huống đơn giản / lượng dữ liệu đơn giản. Trong trường hợp này, hãy ưu tiên ConcurrentDictionary thay vì Dictionary, xử lý tốt hơn quyền truy cập đồng thời thông qua nhiều luồng.

Ngoài ra, ưu tiên sử dụng **Bộ nhớ đệm / Redis phân tán** cho bất kỳ lượng dữ liệu đáng kể hoặc các tình huống phức tạp nào.

**Yêu cầu trạng thái trong dịch vụ**

Không sử dụng các dịch vụ có phạm vi để chia sẻ dữ liệu giữa các dịch vụ trong cùng một yêu cầu, vì chúng KHÔNG an toàn cho luồng.

Thay vào đó, **hãy sử dụng** HttpContext.Items.

**Mẫu định vị dịch vụ**

Tránh sử dụng mẫu định vị dịch vụ, mà không tạo phạm vi con, vì sẽ khó biết về các phụ thuộc của một lớp.

Ví dụ: không gọi **GetService()** trong phạm vi mặc định được tạo khi nhận được yêu cầu mới.

Nhưng bạn có thể sử dụng **IServiceScopeFactory.ServiceProvider. GetService()** trong phạm vi con.

**Gọi phương thức Dispose()**

Không gọi phương thức Dispose() theo cách thủ công cho các dịch vụ được chèn qua DI.

Vùng chứa IoC tự động gọi Dispose(), ở cuối phạm vi của nó.

**Phụ thuộc cố định**

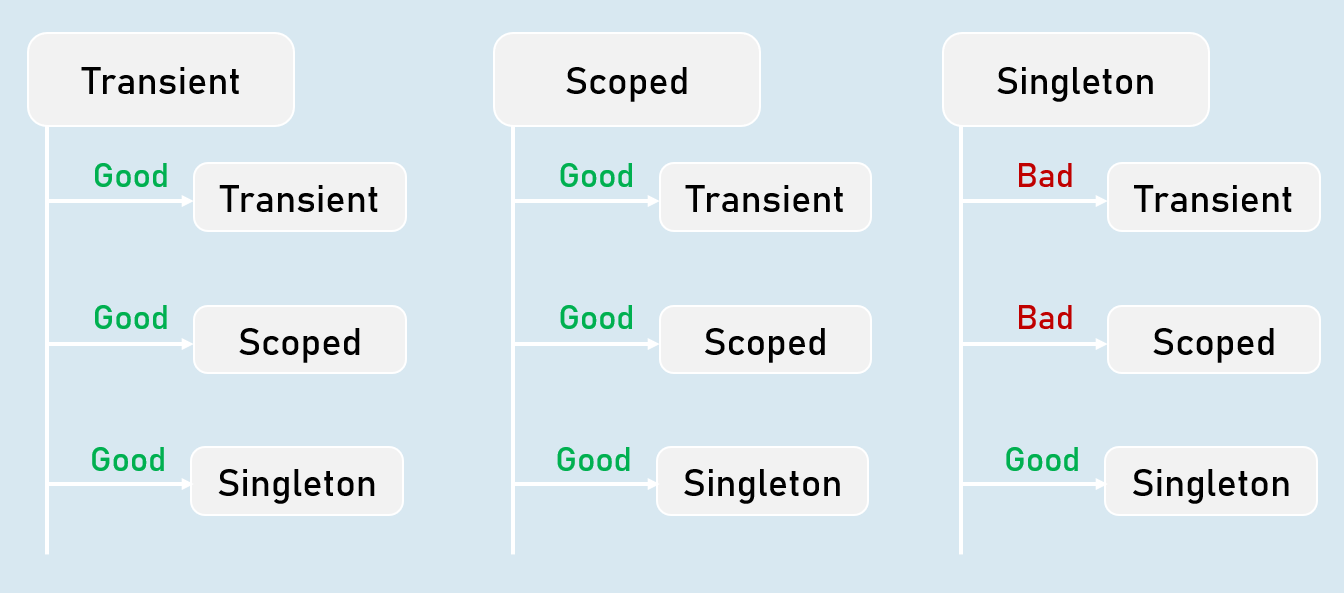
Không chèn các dịch vụ có phạm vi hoặc tạm thời vào các dịch vụ singleton.

Bởi vì, trong trường hợp này, các dịch vụ tạm thời hoặc có phạm vi hoạt động như các dịch vụ đơn lẻ, bên trong dịch vụ đơn lẻ.

**Lưu trữ tham chiếu của phiên bản dịch vụ**

Không giữ tham chiếu của một đối tượng dịch vụ đã giải quyết.

Nó có thể gây rò rỉ bộ nhớ và bạn có thể có quyền truy cập vào đối tượng dịch vụ đã vứt bỏ.



Giao diện tự động

* Autofac là một thư viện vùng chứa IoC khác cho .Net Core.
* Có nghĩa là, cả hai đều được kết hợp chặt chẽ.
* Microsoft.Extensions.DependencyInjection [với] Autofac
* https://autofac.readthedocs.io/en/latest/getting-started/index.html

**Microsoft.Extensions.DependencyInjection**

* Vùng chứa IoC tích hợp trong lõi asp.net
* Tuổi thọ: Thoáng qua, Phạm vi, Singleton
* Siêu dữ liệu cho các dịch vụ: Không được hỗ trợ
* Trình trang trí: Không được hỗ trợ

**Giao diện tự động**

* Thay thế cho Microsoft.Extensions
* Thời gian trọn đời: InstancePerDependency, InstancePerLifetimeScope, SingleInstance, InstancePerOwned, InstancePerMatchingLifetimeScope
* Siêu dữ liệu cho các dịch vụ: Được hỗ trợ
* Người trang trí: Được hỗ trợ

# **11. Môi trường [MVC vs API]**

Môi trường cốt lõi ASP.NET

Trong ASP.NET Core, môi trường là các cấu hình được đặt tên cho phép bạn điều chỉnh hành vi của ứng dụng cho các tình huống triển khai khác nhau. Điều này giúp bạn quản lý cài đặt, cấu hình và quy trình phần mềm trung gian dành riêng cho môi trường phát triển, thử nghiệm, dàn dựng hoặc sản xuất.

**Môi trường chung**

* **Phát triển:** Môi trường phát triển địa phương của bạn. Đó là nơi bạn xây dựng và kiểm tra ứng dụng của mình.
* **Giai đoạn:** Môi trường tiền sản xuất phản ánh chặt chẽ thiết lập sản xuất của bạn. Bạn sử dụng nó để kiểm tra và xác nhận cuối cùng.
* **Sản xuất:** Môi trường trực tiếp nơi người dùng tương tác với ứng dụng của bạn.

**Thiết lập môi trường**

ASP.NET Core đọc môi trường từ biến môi trường ASPNETCORE\_ENVIRONMENT khi ứng dụng của bạn khởi động. Giá trị của biến này xác định môi trường hoạt động.

**Làm thế nào để thiết lập môi trường**

* **launchSettings.json:** Đối với Visual Studio, bạn có thể đặt biến ASPNETCORE\_ENVIRONMENT trong tệp launchSettings.json trong thư mục Thuộc tính của dự án.
* **Biến môi trường:** Đặt biến ASPNETCORE\_ENVIRONMENT trực tiếp trong các biến môi trường của hệ thống của bạn.
* **Dòng lệnh:** Khi chạy ứng dụng của bạn từ dòng lệnh, bạn có thể đặt biến môi trường bằng cách sử dụng cờ --environment hoặc -e:Bash
  1. dotnet run --environment Staging

**Sử dụng môi trường trong Program.cs**

1. **Truy xuất môi trường:**
   1. var builder = WebApplication.CreateBuilder(args);
   2. var environment = trình tạo. Môi trường;

Đối tượng môi trường cung cấp cho bạn quyền truy cập vào tên của môi trường hiện tại và các thuộc tính khác.

1. **Cấu hình có điều kiện:** Bạn có thể sử dụng logic có điều kiện dựa trên tên môi trường để định cấu hình các cài đặt hoặc phần mềm trung gian khác nhau.

C#

* 1. nếu (ứng dụng. Môi trường.IsDevelopment())
  2. {
  3. App. UseDeveloperExceptionPage(); Sử dụng trang lỗi chi tiết trong quá trình phát triển
  4. }
  5. khác
  6. {
  7. App. UseExceptionHandler("/lỗi"); Sử dụng trang lỗi chung trong sản xuất
  8. }

1. **Các tệp cấu hình dành riêng cho môi trường:**
   1. Bạn có thể tạo các tệp cấu hình dành riêng cho môi trường như cài đặt ứng dụng. Development.json, cài đặt ứng dụng. Staging.json và cài đặt ứng dụng. Production.json.
   2. ASP.NET Core tự động tải tệp cấu hình thích hợp dựa trên môi trường hiện tại.
   3. Sử dụng các tệp này để lưu trữ các cài đặt khác nhau giữa các môi trường, chẳng hạn như chuỗi kết nối cơ sở dữ liệu hoặc khóa API.
   4. Các tệp này ghi đè cài đặt trong appsettings.json.

**Các phương pháp hay nhất**

* **Cấu hình dành riêng cho môi trường:** Tách cấu hình của bạn thành các tệp dành riêng cho môi trường để tránh hiển thị dữ liệu nhạy cảm (như thông tin đăng nhập cơ sở dữ liệu sản xuất) trong môi trường phát triển của bạn.
* **Pipeline phần mềm trung gian:** Điều chỉnh các đường ống phần mềm trung gian của bạn cho từng môi trường. Ví dụ: sử dụng UseDeveloperExceptionPage trong phát triển nhưng UseExceptionHandler trong sản xuất.
* **Ghi nhật ký:** Định cấu hình các cấp độ và mục tiêu ghi nhật ký khác nhau cho các môi trường khác nhau (ví dụ: ghi nhật ký dài hơn trong quá trình phát triển).
* **Cờ tính năng:** Sử dụng các biến môi trường hoặc giá trị cấu hình để bật hoặc tắt các tính năng tùy thuộc vào môi trường.

**Ví dụ (Program.cs)**

1. var builder = WebApplication.CreateBuilder(args);
2. var app = trình tạo. Xây dựng();
4. nếu (ứng dụng. Môi trường.IsDevelopment())
5. {
6. Cấu hình dành riêng cho phát triển
7. }
8. else if (app. Environment.IsStaging())
9. {
10. Cấu hình dành riêng cho giai đoạn
11. }
12. else // Sản xuất
13. {
14. Cấu hình dành riêng cho sản xuất
15. }
17. // ... Phần còn lại của thiết lập ứng dụng của bạn...

**Ghi chú**

* **Tính linh hoạt:** Môi trường cho phép bạn dễ dàng điều chỉnh ứng dụng của mình cho các tình huống khác nhau.
* **Cấu hình:** Sử dụng các tệp cấu hình dành riêng cho môi trường (appsettings.{ Môi trường}.json) cho tổ chức.
* **Phần mềm trung gian:** Tùy chỉnh đường ống phần mềm trung gian dựa trên môi trường.
* **Best Practices:** Thực hiện theo các phương pháp hay nhất nêu trên để đảm bảo quá trình triển khai diễn ra suôn sẻ và hành vi tối ưu trong từng môi trường.

Hiểu launchSettings.json

Tệp này chủ yếu được Visual Studio sử dụng để định cấu hình cách ứng dụng ASP.NET Core của bạn khởi chạy trong quá trình phát triển. Nó chứa cài đặt cho các cấu hình khác nhau (ví dụ: IIS Express, ProjectName) và cung cấp một cách thuận tiện để thiết lập các biến môi trường mà không cần sửa đổi các biến môi trường toàn cầu của hệ thống của bạn.

**Vị trí**

Bạn sẽ tìm thấy launchSettings.json trong thư mục Properties trong thư mục gốc của dự án.

**Cấu trúc**

1. {
2. "iisSettings": { ... }, // Cài đặt cho IIS Express (nếu được sử dụng)
3. "hồ sơ": {
4. "IIS Express": { ... }, // Cấu hình cho cấu hình IIS Express
5. "YourProjectName": { // Cấu hình để chạy dự án trực tiếp
6. "commandName": "Dự án",
7. "dotnetRunMessages": "đúng",
8. "launchBrowser": đúng,
9. "applicationUrl": "https://localhost:7272; http://localhost:5248", // URL để khởi chạy
10. "biến môi trường": {
11. "ASPNETCORE\_ENVIRONMENT": "Phát triển" // Thiết lập môi trường
12. }
13. }
14. }
15. }

**Đặt biến ASPNETCORE\_ENVIRONMENT**

Trong phần environmentVariables của hồ sơ mong muốn (ví dụ: "YourProjectName"), bạn có thể đặt biến ASPNETCORE\_ENVIRONMENT thành một trong các giá trị chuẩn:

* **Phát triển:** Để phát triển và gỡ lỗi cục bộ.
* **Dàn dựng:** Để thử nghiệm trước khi sản xuất.
* **Sản xuất:** Đối với môi trường trực tiếp.

Bạn cũng có thể sử dụng tên môi trường tùy chỉnh nếu cần.

**Ví dụ: Thiết lập môi trường phát triển**

1. "biến môi trường": {
2. "ASPNETCORE\_ENVIRONMENT": "Phát triển"
3. }

**Cách thức hoạt động**

Khi bạn khởi chạy ứng dụng của mình từ Visual Studio bằng cách sử dụng một cấu hình cụ thể, các thiết lập environmentVariables sẽ được áp dụng cho quá trình đang chạy. Điều này đảm bảo rằng ứng dụng của bạn đọc đúng giá trị cho ASPNETCORE\_ENVIRONMENT, từ đó ảnh hưởng đến cài đặt cấu hình nào được tải (từ appsettings.json, appsettings. Development.json, v.v.) và những đường ống phần mềm trung gian nào được sử dụng.

**Những cân nhắc quan trọng**

* **Tệp cấu hình dành riêng cho môi trường:** Hãy nhớ rằng bạn vẫn cần tạo các tệp cấu hình dành riêng cho môi trường (ví dụ: cài đặt ứng dụng. Development.json) để lưu trữ các cài đặt khác nhau giữa các môi trường. launchSettings.json chỉ đặt biến môi trường.
* **Phát triển cục bộ:** Tệp launchSettings.json chủ yếu dành cho phát triển cục bộ với Visual Studio. Khi bạn triển khai ứng dụng của mình lên máy chủ, bạn thường sẽ đặt biến ASPNETCORE\_ENVIRONMENT thông qua cấu hình của môi trường lưu trữ (ví dụ: trong tệp cấu hình của máy chủ web hoặc các biến môi trường).
* **Nhiều cấu hình:** launchSettings.json có thể chứa nhiều cấu hình, mỗi cấu hình có bộ biến môi trường riêng. Điều này cho phép bạn dễ dàng chuyển đổi giữa các cấu hình khác nhau trong quá trình phát triển.

Trang ngoại lệ dành cho nhà phát triển

Trang ngoại lệ dành cho nhà phát triển là một công cụ mạnh mẽ trong ASP.NET Core để chẩn đoán các ngoại lệ trong quá trình phát triển. Nó cung cấp một cái nhìn chi tiết về ngoại lệ, bao gồm:

* Dấu vết ngăn xếp
* Chi tiết yêu cầu (tiêu đề, chuỗi truy vấn, cookie)
* Thông tin định tuyến
* Cài đặt cấu hình

Thông tin này là vô giá để xác định và khắc phục sự cố một cách nhanh chóng.

**Hành vi cụ thể của môi trường**

* **Phát triển:** Trang ngoại lệ dành cho nhà phát triển được bật theo mặc định trong môi trường Phát triển. Điều này có ý nghĩa vì trong quá trình phát triển, bạn muốn có càng nhiều thông tin càng tốt để giúp bạn khắc phục sự cố.
* **Môi trường sản xuất và môi trường khác:** Trong môi trường sản xuất hoặc môi trường không phát triển khác, trang này thường bị vô hiệu hóa do lo ngại về bảo mật. Việc tiết lộ thông tin ngoại lệ chi tiết cho công chúng có thể tiết lộ các lỗ hổng hoặc chi tiết nhạy cảm về hoạt động nội bộ của ứng dụng của bạn.

**Giao diện IWebHostEnvironment: Truy cập thông tin môi trường**

Giao diện IWebHostEnvironment cung cấp cho bạn quyền truy cập vào thông tin về môi trường lưu trữ của ứng dụng ASP.NET Core của bạn. Nó bao gồm các thuộc tính như:

* **EnvironmentName:** Tên của môi trường hiện tại (Phát triển, Giai đoạn, Sản xuất hoặc tên tùy chỉnh).
* **WebRootPath:** Đường dẫn đến thư mục gốc web của ứng dụng.
* **ContentRootPath:** Đường dẫn đến thư mục gốc nội dung của ứng dụng.

**Sử dụng IWebHostEnvironment và ứng dụng. Môi trường**

1. HomeController.cs
2. lớp công cộng HomeController: Bộ điều khiển
3. {
4. \_webHostEnvironment IWebHostEnvironment chỉ đọc riêng; Tiêm
6. HomeController công cộng (IWebHostEnvironment webHostEnvironment)
7. {
8. \_webHostEnvironment = webHostEnvironment;
9. }
11. [Tuyến ("/")]
12. Chỉ số IActionResult công khai()
13. {
14. ViewBag.CurrentEnviornment = \_webHostEnvironment.EnvironmentName;
15. trả về View();
16. }
17. }

Trong mã này, IWebHostEnvironment được đưa vào HomeController. Tên môi trường hiện tại (\_webHostEnvironment.EnvironmentName) sau đó được gán cho ViewBag.CurrentEnvironment và được gửi đến chế độ xem để hiển thị.

**Bật trang ngoại lệ dành cho nhà phát triển trong các môi trường cụ thể**

1. Program.cs
2. nếu (ứng dụng. Environment.IsDevelopment() || App. Environment.IsStaging() || App. Environment.IsEnvironment("Beta"))
3. {
4. App. UseDeveloperExceptionPage();
5. }

Trong đoạn mã này, Trang ngoại lệ dành cho nhà phát triển chỉ được bật nếu môi trường là Phát triển, Giai đoạn hoặc môi trường tùy chỉnh có tên "Beta". Bạn có thể sử dụng ứng dụng. Environment.IsDevelopment() v.v. vì chúng chỉ là viết tắt của ứng dụng. Environment.EnvironmentName == "Phát triển". Điều này có thể hữu ích trong các tình huống mà bạn muốn đưa giai đoạn vào quy trình phát triển của mình.

**Ghi chú**

* **Mục đích:** Trang ngoại lệ dành cho nhà phát triển cung cấp thông tin lỗi chi tiết trong quá trình phát triển.
* **Môi trường:** Nó được bật theo mặc định trong Phát triển, nhưng bạn có thể tùy chỉnh hành vi của nó dựa trên các môi trường khác.
* **IWebHostEnvironment:** Sử dụng giao diện này để truy cập thông tin môi trường trong bộ điều khiển hoặc phần mềm trung gian của bạn.
* **Bảo mật:** Luôn tắt Trang ngoại lệ dành cho nhà phát triển trong quá trình sản xuất để tránh tiết lộ các chi tiết nhạy cảm.
* **Trang lỗi tùy chỉnh:** Đối với sản xuất, hãy sử dụng ứng dụng. UseExceptionHandler để tạo các trang lỗi tùy chỉnh cung cấp thông báo thân thiện với người dùng mà không tiết lộ thông tin nội bộ.

<môi trường> Tag Helper

Trình trợ giúp thẻ <môi trường> là một công cụ linh hoạt cho phép bạn bao gồm hoặc loại trừ nội dung cụ thể trong chế độ xem của mình tùy thuộc vào môi trường hiện tại mà ứng dụng ASP.NET Core của bạn đang chạy.

Điều này đặc biệt hữu ích cho các tình huống mà bạn muốn:

* **Bao gồm tài nguyên phát triển:** Tải các tệp CSS hoặc JavaScript chưa thu nhỏ trong quá trình phát triển để tạo điều kiện gỡ lỗi và kiểm tra.
* **Tối ưu hóa cho sản xuất:** Tải các tài sản đã thu nhỏ và đóng gói trong sản xuất để cải thiện hiệu suất.
* **Nội dung dành riêng cho môi trường hiển thị:** Hiển thị các thông báo, cảnh báo hoặc tính năng khác nhau dựa trên môi trường.

**Cú pháp**

1. <môi trường bao gồm="Môi trường1,Môi trường2,...">
2. Nội dung để hiển thị nếu môi trường khớp với bất kỳ môi trường nào được bao gồm
3. < / môi trường>
5. <environment exclude="Environment1,Environment2,...">
6. Nội dung để hiển thị nếu môi trường KHÔNG khớp với bất kỳ môi trường nào bị loại trừ
7. < / môi trường>

* **include:** Danh sách các tên môi trường được phân tách bằng dấu phẩy mà nội dung sẽ được hiển thị.
* **exclude:** Danh sách các tên môi trường được phân tách bằng dấu phẩy mà nội dung không nên được hiển thị.

**Tên môi trường**

* **Tiêu chuẩn:** Phát triển, Dàn dựng, Sản xuất.
* **Tùy chỉnh:** Bạn cũng có thể xác định và sử dụng tên môi trường tùy chỉnh của riêng mình.

**Cách thức hoạt động**

1. **Kiểm tra môi trường:** Trình trợ giúp thẻ <môi trường> đọc giá trị của biến môi trường ASPNETCORE\_ENVIRONMENT để xác định môi trường hiện tại.
2. **Kết xuất có điều kiện:** Dựa trên các thuộc tính bao gồm hoặc loại trừ và môi trường hiện tại, nó sẽ hiển thị hoặc bỏ qua nội dung trong trình trợ giúp thẻ.

**Ví dụ về mã**

1. <môi trường bao gồm="Phát triển">
2. <link rel="stylesheet" href="~/css/site.css" />
3. < / môi trường>
4. <environment exclude="Phát triển">
5. <link rel="stylesheet" href="~/css/site.min.css" />
6. < / môi trường>

Trong ví dụ này:

* Tệp site.css chưa thu nhỏ chỉ được tải trong môi trường Phát triển.
* Tệp site.min.css đã thu nhỏ được tải trong tất cả các môi trường khác.

**Ghi chú**

* **Tính linh hoạt:** Dễ dàng điều chỉnh chế độ xem của bạn cho phù hợp với các môi trường khác nhau mà không cần logic điều kiện phức tạp.
* **Tối ưu hóa hiệu suất:** Phục vụ tài sản được tối ưu hóa trong sản xuất trong khi vẫn giữ được sự linh hoạt trong phát triển.
* **Nội dung dành riêng cho môi trường:** Chỉ hiển thị cảnh báo, thông báo hoặc công cụ gỡ lỗi khi cần.

**Các phương pháp hay nhất**

* **Sử dụng cho Tài sản tĩnh:** Chủ yếu tận dụng trình trợ giúp thẻ <môi trường> để bao gồm hoặc loại trừ các tệp tĩnh (CSS, JavaScript) dựa trên môi trường.
* **Tránh logic phức tạp:** Giữ nội dung trong thẻ <môi trường> tương đối đơn giản. Nếu bạn cần logic phức tạp hơn, hãy cân nhắc sử dụng chế độ xem một phần hoặc thành phần chế độ xem.
* **Môi trường tùy chỉnh:** Nếu bạn cần nhiều hơn môi trường tiêu chuẩn, hãy xác định và sử dụng môi trường của riêng bạn.

Đặt môi trường từ thiết bị đầu cuối

* **Tính linh hoạt:** Cài đặt biến môi trường trực tiếp từ thiết bị đầu cuối cho phép bạn dễ dàng chuyển đổi giữa các môi trường khác nhau (phát triển, dàn dựng, sản xuất) mà không cần sửa đổi tệp cấu hình hoặc cài đặt IDE.
* **Tự động hóa:** Cách tiếp cận này có thể dễ dàng viết kịch bản, cho phép bạn tự động hóa các quy trình triển khai và thay đổi cấu hình liền mạch cho các môi trường khác nhau.
* **Môi trường không phải Windows:** Nếu bạn đang làm việc trên macOS hoặc Linux, thiết bị đầu cuối là cách chính để quản lý các biến môi trường.

**Cài đặt ASPNETCORE\_ENVIRONMENT trong PowerShell**

* **Windows, macOS, Linux:**

1. $env:ASPNETCORE\_ENVIRONMENT = "Phát triển" # Đặt thành Phát triển
2. $env:ASPNETCORE\_ENVIRONMENT = "Sản xuất" # Đặt thành sản xuất

* **Phạm vi:** Trong PowerShell, các biến môi trường được đặt bằng $env: thường được giới hạn trong phiên hiện tại. Để làm cho chúng bền vững, bạn cần sửa đổi hệ thống hoặc các biến môi trường người dùng (xem bên dưới).

**Cài đặt ASPNETCORE\_ENVIRONMENT trong Command Prompt**

* **Windows:**

1. set ASPNETCORE\_ENVIRONMENT=Development # Đặt thành Development
2. set ASPNETCORE\_ENVIRONMENT=Production # Đặt thành Production

**Phạm vi:** Theo mặc định, các biến được đặt bằng lệnh set là tạm thời và chỉ áp dụng cho phiên dấu nhắc lệnh hiện tại. Để làm cho chúng bền vững, hãy sử dụng công tắc /M:

1. setx ASPNETCORE\_ENVIRONMENT Development /M # Đặt cho tài khoản người dùng (liên tục)

* **macOS và Linux (bash):**

1. export ASPNETCORE\_ENVIRONMENT=Development # Đặt thành Development
2. export ASPNETCORE\_ENVIRONMENT=Production # Đặt thành Production

**Làm cho các biến môi trường bền vững**

* **Windows (Thuộc tính hệ thống):**
  1. Nhấp chuột phải vào "PC này" và chọn "Thuộc tính".
  2. Nhấp vào "Cài đặt hệ thống nâng cao".
  3. Nhấp vào nút "Biến môi trường".
  4. Trong phần "Biến hệ thống" (hoặc "Biến người dùng" cho một người dùng cụ thể), hãy nhấp vào "Mới".
  5. Nhập ASPNETCORE\_ENVIRONMENT làm tên biến và môi trường mong muốn làm giá trị.
* **macOS và Linux (Tệp cấu hình Shell):**
  1. Chỉnh sửa tệp cấu hình shell của bạn (.bashrc, .zshrc, v.v.) và thêm dòng sau (thay thế Phát triển bằng môi trường mong muốn của bạn):Bash
     1. export ASPNETCORE\_ENVIRONMENT=Phát triển
  2. Sau khi lưu tệp, hãy chạy source ~/.bashrc (hoặc lệnh thích hợp cho shell của bạn) để tải lại cấu hình.

**Những cân nhắc quan trọng**

* **Ghi đè:** Khi sử dụng nhiều cách thiết lập môi trường, cách cụ thể nhất sẽ được ưu tiên. Ví dụ: một giá trị được đặt trong thiết bị đầu cuối sẽ ghi đè giá trị trong launchSettings.json.
* **Phân biệt chữ hoa chữ thường (Linux/macOS):** Tên biến môi trường phân biệt chữ hoa chữ thường trên Linux và macOS. Đảm bảo sử dụng cách viết hoa chính xác (ASPNETCORE\_ENVIRONMENT).
* **Tệp cấu hình dành riêng cho môi trường:** Ngay cả sau khi đặt biến môi trường, hãy đảm bảo bạn có cài đặt ứng dụng tương ứng. Environment}.json tệp trong dự án của bạn để tải cài đặt chính xác cho môi trường đó.

**Ví dụ: Chạy ứng dụng của bạn với các môi trường khác nhau**

1. # PowerShell
2. $env:ASPNETCORE\_ENVIRONMENT = "Phát triển"
3. chạy dotnet
5. # Command Prompt (Windows)
6. đặt ASPNETCORE\_ENVIRONMENT = Sản xuất
7. chạy dotnet
9. # bash (macOS / Linux)
10. export ASPNETCORE\_ENVIRONMENT=Dàn dựng
11. chạy dotnet

Những điểm chính cần nhớ

* **Mục đích:** Cung cấp cấu hình được đặt tên để điều chỉnh hành vi của ứng dụng cho các tình huống khác nhau (phát triển, dàn dựng, sản xuất, v.v.).
* **Biến môi trường:**
  + ASPNETCORE\_ENVIRONMENT là biến môi trường quan trọng.
  + Giá trị của nó quyết định môi trường hoạt động.
* **Thiết lập môi trường:**
  + **launchSettings.json (Phát triển):** Đặt trong phần environmentVariables của hồ sơ.
  + **Biến môi trường hệ thống:** Đặt trực tiếp trên máy của bạn (liên tục).
  + **Dòng lệnh:** Sử dụng cờ --environment hoặc -e khi chạy ứng dụng (ví dụ: dotnet run --environment Staging).
* **Giao diện IWebHostEnvironment:**
  + Sử dụng nó trong mã của bạn để truy cập thông tin môi trường (ví dụ: EnvironmentName, WebRootPath).
  + Đưa nó vào bộ điều khiển hoặc phần mềm trung gian của bạn:
  + \_env IWebHostEnvironment chỉ đọc riêng;
  + MyController công khai (IWebHostEnvironment env)
  + {
  + \_env = env;
  + }
* **Cấu hình dành riêng cho môi trường:**
  + Tạo các tệp như cài đặt ứng dụng. Development.json, cài đặt ứng dụng. Staging.json, v.v.
  + ASP.NET Core tự động tải tệp thích hợp dựa trên môi trường.
  + Ghi đè cài đặt cơ sở trong appsettings.json.
* **Cấu hình có điều kiện (trong Program.cs):**
  + Sử dụng nếu (ứng dụng. Environment.IsDevelopment()) hoặc các phương thức tương tự để áp dụng cài đặt hoặc phần mềm trung gian dựa trên môi trường.
  + nếu (ứng dụng. Môi trường.IsDevelopment())
  + {
  + App. UseDeveloperExceptionPage();
  + }
* **Môi trường mặc định:**
  + Phát triển: Mặc định cho phát triển địa phương.
  + Giai đoạn: Thường được sử dụng để thử nghiệm trước khi sản xuất.
  + Sản xuất: Môi trường trực tiếp.
* **Môi trường tùy chỉnh:** Bạn có thể xác định và sử dụng tên môi trường của riêng mình.
* **Các phương pháp hay nhất:**
  + **Cấu hình riêng biệt:** Giữ các cài đặt dành riêng cho môi trường trong các tệp riêng biệt.
  + **Điều chỉnh phần mềm trung gian:** Sử dụng các đường ống phần mềm trung gian khác nhau cho các môi trường khác nhau (ví dụ: chỉ bật DeveloperExceptionPage trong quá trình phát triển).
  + **Ghi nhật ký:** Điều chỉnh mức ghi nhật ký dựa trên môi trường.
  + **Cờ tính năng:** Sử dụng các biến môi trường để bật/tắt các tính năng.

**Mẹo phỏng vấn**

* **Giải thích lý do:** Có thể nêu rõ lý do sử dụng môi trường (cấu hình, bảo mật, tính linh hoạt).
* **Cấu hình:** Hiển thị cách bạn sẽ sử dụng appsettings.{ Môi trường}.json tệp để quản lý các cài đặt dành riêng cho môi trường.
* **Phần mềm trung gian:** Giải thích cách bạn sẽ tùy chỉnh các đường ống phần mềm trung gian dựa trên môi trường.
* **Triển khai:** Thảo luận về cách bạn sẽ thiết lập biến môi trường khi triển khai cho các máy chủ khác nhau.

Giới thiệu về môi trường

Môi trường đại diện là một hệ thống trong đó ứng dụng được triển khai và thực thi.

**Phát triển**

Môi trường, nơi nhà phát triển thực hiện các thay đổi trong mã, cam kết mã cho kiểm soát nguồn.

**Dàn**

Môi trường, nơi ứng dụng chạy trên máy chủ, từ đó các nhà phát triển và kiểm soát chất lượng khác truy cập ứng dụng.

**Sản xuất**

Môi trường, nơi người dùng cuối thực sự truy cập ứng dụng.

Ngay sau đó, đó là nơi ứng dụng "sống" với khán giả.

Cài đặt môi trường

**Đặt môi trường trong launchSettings.json**

trong launchSettings.json

1. {
2. "hồ sơ":
3. {
4. "profileName":
5. {
6. "environmentVariables":
7. {
8. "DOTNET\_ENVIRONMENT": "Tên môi trường ở đây",
9. "ASPNETCORE\_ENVIRONMENT": "EnvironmentNameHere"
10. }
11. }
12. }
13. }

**Truy cập môi trường trong Program.cs**

App. Môi trường

IWebHostMôi trường

**Tên môi trường**

Nhận hoặc đặt tên của môi trường.

Theo mặc định, nó đọc giá trị từ DOTNET\_ENVIRONMENT hoặc ASPNETCORE\_ENVIRONMENT.

**ContentRootPath**

Nhận hoặc đặt đường dẫn tuyệt đối của thư mục ứng dụng.

**IsDevelopment()**

Trả về Boolean true, nếu tên môi trường hiện tại là "Development".

**IsStaging()**

Trả về Boolean true, nếu tên môi trường hiện tại là "Staging".

**IsProduction()**

Trả về Boolean true, nếu tên môi trường hiện tại là "Production".

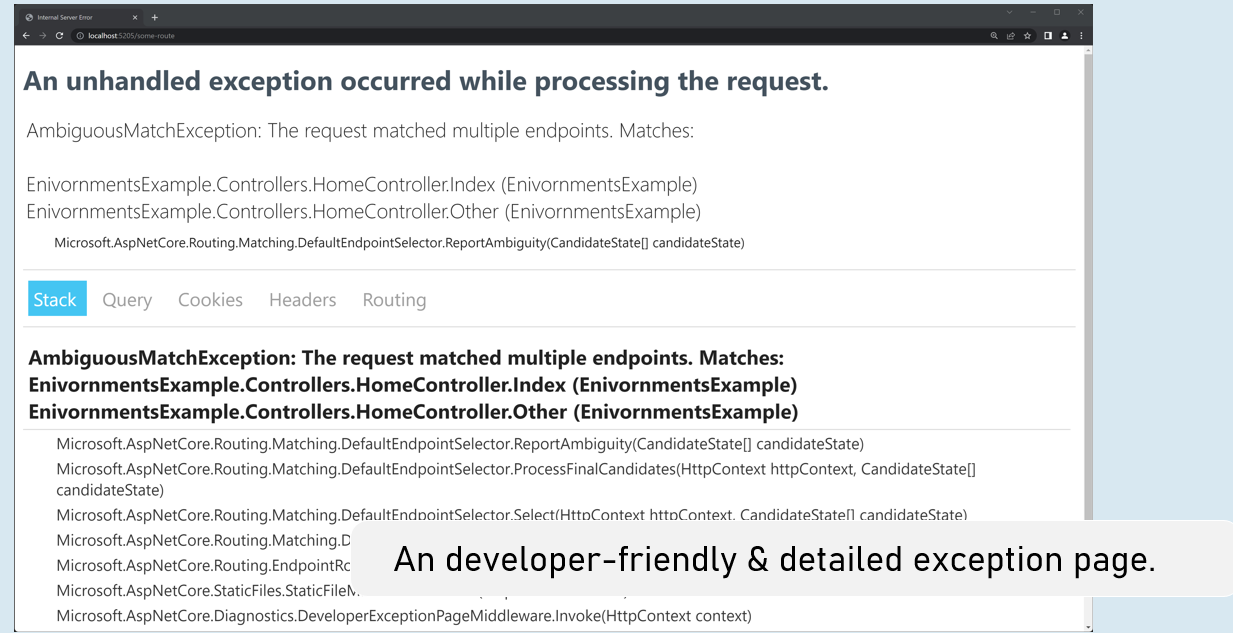
**IsEnvironment(chuỗi environmentName)**

Trả về Boolean true, nếu tên môi trường hiện tại khớp với môi trường được chỉ định.

Truy cập Môi trường trong Bộ điều khiển và các lớp khác

1. sử dụng Microsoft.AspNetCore.Mvc;
2. sử dụng Microsoft.AspNetCore.Hosting;
4. lớp công cộng ControllerName : Bộ điều khiển
5. {
6. \_webHost IWebHostEnvironment chỉ đọc riêng;
8. Tên bộ điều khiển công cộng (IWebHostEnvironment webHost)
9. {
10. \_webHost = webHost;
11. }
12. }

Trang ngoại lệ dành cho nhà phát triển

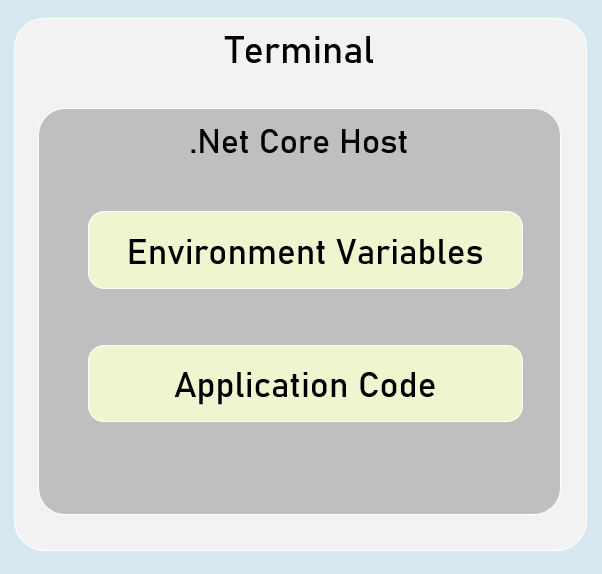


Bật trang ngoại lệ dành cho nhà phát triển

trong Program.cs

1. nếu (ứng dụng. Môi trường.IsDevelopment()
2. {
3. App. UseDeveloperExceptionPage();
4. }

Môi trường cấp quy trình



Các biến môi trường chỉ được lưu trữ và có thể truy cập trong cùng một quy trình.

Cài đặt các biến môi trường trong quy trình

trong "Windows PowerShell" / "PowerShell dành cho nhà phát triển trong VS"

$Env:Environment="Tên môi trường"

dotnet run --no-launch-profile

<môi trường> tag trợ giúp

**bao gồm**

1. <môi trường bao gồm="Môi trường1,Môi trường2">
2. Nội dung HTML tại đây
3. < / môi trường>

Nó chỉ hiển thị nội dung khi tên môi trường hiện tại khớp với một trong các tên môi trường được chỉ định trong thuộc tính "include".

**Loại trừ**

1. <environment exclude="Môi trường1,Môi trường2">
2. Nội dung HTML tại đây
3. < / môi trường>

Nó chỉ hiển thị nội dung khi tên môi trường hiện tại không khớp với một trong hai tên môi trường được chỉ định trong thuộc tính "exclude".

# **12. Cấu hình [MVC vs API]**

Cấu hình lõi ASP.NET

Cấu hình là nền tảng của bất kỳ ứng dụng nào, cung cấp các cài đặt và giá trị cần thiết thúc đẩy hành vi của nó. Hệ thống cấu hình của ASP.NET Core rất linh hoạt và có thể mở rộng, cho phép bạn truy xuất dữ liệu cấu hình từ nhiều nguồn khác nhau và ưu tiên chúng theo nhu cầu của mình.

**Khái niệm cốt lõi**

* **Nhà cung cấp cấu hình:** Các thành phần này đọc dữ liệu cấu hình từ các nguồn khác nhau và điền vào kho cấu hình trung tâm.
* **Nguồn cấu hình:** Vị trí hoặc cơ chế thực tế nơi dữ liệu cấu hình của bạn nằm (ví dụ: tệp, biến môi trường, đối số dòng lệnh).
* **Cặp khóa-giá trị:** Dữ liệu cấu hình được lưu trữ dưới dạng cặp khóa-giá trị, trong đó khóa là mã định danh chuỗi và giá trị là dữ liệu cấu hình (chuỗi, số, boolean, v.v.).

Các nguồn cấu hình phổ biến

1. **Tệp (JSON, XML, INI):**
   * **Mục đích:** Lưu trữ dữ liệu cấu hình trong các tệp có cấu trúc. JSON là định dạng mặc định và phổ biến nhất trong ASP.NET Core.
   * **Ưu điểm:** Dễ đọc và chỉnh sửa, hỗ trợ cấu trúc phân cấp.
   * **Nhược điểm:** Có thể không thích hợp để lưu trữ bí mật hoặc dữ liệu nhạy cảm cao.
2. **Các biến môi trường:**
   * **Mục đích:** Đọc các giá trị cấu hình từ các biến môi trường.
   * **Ưu điểm:** Lý tưởng cho các cài đặt dành riêng cho môi trường (ví dụ: chuỗi kết nối cơ sở dữ liệu) và bí mật.
   * **Nhược điểm:** Có thể khó quản lý đối với các cấu hình phức tạp hoặc số lượng lớn cài đặt.
3. **Đối số dòng lệnh:**
   * **Mục đích:** Ghi đè các giá trị cấu hình khi chạy ứng dụng từ dòng lệnh.
   * **Ưu điểm:** Cung cấp sự linh hoạt cho cấu hình động một cách nhanh chóng.
   * **Nhược điểm:** Có thể không thích hợp để lưu trữ dữ liệu phức tạp hoặc nhạy cảm.
4. **Các đối tượng .NET trong bộ nhớ:**
   * **Mục đích:** Lưu trữ dữ liệu cấu hình trong từ điển hoặc các đối tượng tùy chỉnh trực tiếp trong mã của bạn.
   * **Ưu điểm:** Tính linh hoạt cho các tình huống cấu hình động hoặc lập trình.
   * **Nhược điểm:** Không dai dẳng, ít thích hợp hơn để quản lý một số lượng lớn cài đặt.
5. **Azure Key Vault:**
   * **Mục đích:** Lưu trữ an toàn bí mật và dữ liệu cấu hình nhạy cảm trên đám mây.
   * **Ưu điểm:** Quản lý tập trung, bảo mật cao các bí mật.
   * **Nhược điểm:** Yêu cầu đăng ký và thiết lập Azure.
6. **Cấu hình ứng dụng Azure:**
   * **Mục đích:** Một dịch vụ dựa trên đám mây mạnh mẽ để quản lý cờ tính năng và cài đặt cấu hình.
   * **Ưu điểm:** Quản lý cờ tính năng, cấu hình tập trung, cập nhật động.
   * **Nhược điểm:** Yêu cầu đăng ký và thiết lập Azure.
7. **Bí mật người dùng (Phát triển):**
   * **Mục đích:** Lưu trữ dữ liệu nhạy cảm (ví dụ: khóa API) trong quá trình phát triển mà không cam kết chúng với kiểm soát nguồn.
   * **Ưu điểm:** An toàn và thuận tiện cho sự phát triển của địa phương.
   * **Nhược điểm:** Không dành cho môi trường sản xuất.

Thêm và quản lý nguồn cấu hình trong Program.cs

1. var builder = WebApplication.CreateBuilder(args);
2. var cấu hình = trình tạo. Cấu hình;
4. Thêm các nguồn cấu hình theo thứ tự ưu tiên mong muốn (được thêm lần cuối cùng thắng)
5. cấu hình. AddJsonFile("appsettings.json", tùy chọn: false, reloadOnChange: true);
6. cấu hình. AddJsonFile($"appsettings.{ môi trường. EnvironmentName}.json", tùy chọn: true, reloadOnChange: true);
7. cấu hình. AddEnvironmentVariables();
8. cấu hình. AddUserSecrets<Program>(); Đối với bí mật phát triển
9. // ... các nguồn khác ...
10. AddJsonFile: Tải cấu hình từ các tệp JSON.
11. AddEnvironmentVariables: Tải cấu hình từ các biến môi trường.
12. AddUserSecrets<Program>(): Tải cấu hình từ kho bí mật người dùng (để phát triển).

**Khi nào nên sử dụng nguồn cấu hình nào**

* **appsettings.json:** Đối với cài đặt mặc định, cấu hình cơ sở, dữ liệu không nhạy cảm.
* **cài đặt ứng dụng. {Environment}.json:** Đối với ghi đè dành riêng cho môi trường.
* **Biến môi trường:** Đối với cài đặt dành riêng cho môi trường, dữ liệu nhạy cảm (khóa API, chuỗi kết nối).
* **Đối số dòng lệnh:** Để ghi đè cài đặt trong quá trình phát triển hoặc triển khai.
* **Bí mật người dùng:** Đối với dữ liệu nhạy cảm trong quá trình phát triển cục bộ.
* **Azure Key Vault:** Để lưu trữ bí mật và dữ liệu nhạy cảm khác một cách an toàn trong sản xuất.
* **Cấu hình ứng dụng Azure:** Để cập nhật cấu hình động, cờ tính năng và quản lý tập trung.

**Các phương pháp hay nhất**

* **Cấu hình nhiều lớp:** Sử dụng nhiều nguồn với thứ tự ưu tiên được xác định rõ ràng để giữ cho cấu hình của bạn có tổ chức và linh hoạt.
* **Cài đặt dành riêng cho môi trường:** Tách các cài đặt nhạy cảm và môi trường cụ thể thành các tệp thích hợp.
* **Quản lý bí mật:** Sử dụng Azure Key Vault hoặc các cơ chế bảo mật khác để lưu trữ dữ liệu nhạy cảm.
* **Nhập mạnh:** Tạo các lớp cấu hình được nhập mạnh bằng cách sử dụng mẫu Tùy chọn (IOptions<T>) để cải thiện độ an toàn kiểu và truy cập dễ dàng hơn vào cài đặt của bạn trong mã.
* **Xác thực:** Xác thực các giá trị cấu hình của bạn trong quá trình khởi động để phát hiện sớm lỗi.
* **Ghi nhật ký:** Ghi nhật ký các sự kiện liên quan đến cấu hình để giúp khắc phục sự cố và gỡ lỗi.

ICcấu hình

Trong ASP.NET Core, giao diện IConfiguration là trái tim của hệ thống cấu hình. Nó đại diện cho một tập hợp các cặp khóa-giá trị có thể được tải từ nhiều nguồn khác nhau (tệp JSON, biến môi trường, v.v.). Giao diện này cung cấp một cách thống nhất để truy cập cài đặt ứng dụng của bạn, bất kể chúng được lưu trữ ở đâu.

**Các phương thức chính, thuộc tính và trình lập chỉ mục**

1. **GetSection(khóa chuỗi):**
   * **Mục đích:** Truy xuất một phần cụ thể của cấu hình dưới dạng IConfigurationSection. Các phần cho phép bạn nhóm các cài đặt liên quan.
   * **Ví dụ:**
   * var connectionStrings = cấu hình. GetSection("Chuỗi kết nối");
2. **GetValue<T>(khóa chuỗi):**
   * **Mục đích:** Truy xuất giá trị cấu hình dưới dạng loại T được chỉ định.
   * **Ví dụ:**
   * var port = cấu hình. GetValue<int>("Máy chủ:Cổng");
3. **GetConnectionString(tên chuỗi):**
   * **Mục đích:** Truy xuất chuỗi kết nối từ phần "ConnectionStrings" của cấu hình.
   * **Ví dụ:**
   * var connectionString = cấu hình. GetConnectionString("Kết nối mặc định");
4. **GetChildren():**
   * **Mục đích:** Trả về một tập hợp có thể kiểm tra được của các đối tượng IConfigurationSection đại diện cho các con trực tiếp của phần hiện tại.
   * **Ví dụ:**
   * var sections = cấu hình. GetSection("Ghi nhật ký"). GetChildren();
5. **Trình lập chỉ mục (this[string key]):**
   * **Mục đích:** Truy xuất giá trị cấu hình dưới dạng chuỗi.
   * **Ví dụ:**
   * var value = configuration["Ghi nhật ký:LogLevel:Default"];

Tiêm IConfiguration

* **Trong Bộ điều khiển:**

1. lớp công cộng HomeController: Bộ điều khiển
2. {
3. IConfiguration chỉ đọc riêng \_configuration; Trường lưu trữ IConfiguration
5. HomeController công cộng (cấu hình IConfiguration)
6. {
7. \_configuration = cấu hình;
8. }
10. Chỉ số IActionResult công khai()
11. {
12. var myKeyValue = \_configuration["MyKey"]; Giá trị cấu hình truy cập
13. trả về View();
14. }
15. }

* **Trong Dịch vụ:**

1. lớp công khai EmailService : IEmailService
2. {
3. IConfiguration chỉ đọc riêng \_configuration;
5. Public EmailService (cấu hình IConfiguration)
6. {
7. \_configuration = cấu hình;
8. }
10. public void SendEmail(string to, string subject, string body)
11. {
12. var smtpServer = \_configuration["Email:SmtpServer"]; Sử dụng cấu hình cho cài đặt email
13. // ... (logic gửi email)
14. }
15. }

Trong cả hai trường hợp, IConfiguration được đưa vào thông qua constructor bằng cách sử dụng ASP.NET dependency injection của Core.

**Các phương pháp hay nhất**

* **Cấu hình gõ mạnh:** Sử dụng mẫu Tùy chọn (IOptions<T>) để ánh xạ các giá trị cấu hình của bạn với các đối tượng được nhập mạnh để truy cập dễ dàng hơn và an toàn cho loại.
* **Cài đặt dành riêng cho môi trường:** Sử dụng appsettings.{ Môi trường}.json để lưu trữ các giá trị cấu hình khác nhau tùy thuộc vào môi trường (Phát triển, Sản xuất, v.v.).
* **Quản lý bí mật:** Lưu trữ thông tin nhạy cảm (ví dụ: mật khẩu, khóa API) trong Azure Key Vault hoặc các cơ chế lưu trữ an toàn khác.
* **Cấu hình phân lớp:** Kết hợp nhiều nguồn cấu hình (tệp, biến môi trường, v.v.) với thứ tự ưu tiên được xác định rõ ràng.
* **Tải lại khi thay đổi:** Cân nhắc sử dụng reloadOnChange: true trong nhà cung cấp cấu hình của bạn để tự động tải lại các thay đổi cấu hình mà không cần khởi động lại ứng dụng.

**Ví dụ: Mô hình quyền chọn**

1. MyOptions.cs
2. lớp công khai MyOptions
3. {
4. chuỗi công khai Option1 { get; set; }
5. public int Option2 { get; set; }
6. }
8. Program.cs (hoặc Startup.cs)
9. chủ thầu. Services.Configure<MyOptions>(trình tạo. Configuration.GetSection("Tùy chọn của tôi"));
11. MyService.cs
12. lớp công khai MyService : IMyService
13. {
14. IOptions<MyOptions> \_options chỉ đọc riêng;
16. tùy chọn MyService công khai (IOptions<MyOptions>)
17. {
18. \_options = quyền chọn;
19. }
21. Void công khai DoSomething()
22. {
23. var option1Giá trị = \_options. Giá trị.Tùy chọn1;
24. // ...
25. }
26. }

Trong ví dụ này, lớp MyOptions đại diện cho một phần cấu hình của bạn. Giao diện IOptions<MyOptions> cung cấp một cách được gõ mạnh để truy cập các cài đặt đó trong các dịch vụ của bạn.

Bằng cách làm theo các phương pháp hay nhất này và tận dụng sức mạnh của IConfiguration, bạn có thể xây dựng các ứng dụng ASP.NET Core mạnh mẽ và có thể thích ứng với các cài đặt cấu hình được tổ chức tốt và dễ quản lý.

Cấu hình phân cấp

Trong ASP.NET Core, bạn có thể sắp xếp cài đặt cấu hình của mình thành một cấu trúc phân cấp bằng cách sử dụng các tệp JSON, XML hoặc INI. Cấu trúc phân cấp này cho phép bạn nhóm các cài đặt liên quan trong các phần và tiểu mục, làm cho cấu hình của bạn dễ đọc, dễ bảo trì và có thể mở rộng hơn.

**Cấu hình phân cấp dựa trên JSON (appsettings.json):**

1. {
2. "Chuỗi kết nối": {
3. "DefaultConnection": "Máy chủ = (localdb) \\mssqllocaldb; Cơ sở dữ liệu = Cơ sở dữ liệu của tôi; Trusted\_Connection=Đúng;"
4. },
5. "Ghi nhật ký": {
6. "Cấp độ nhật ký": {
7. "Mặc định": "Thông tin",
8. "Microsoft.AspNetCore": "Cảnh báo"
9. }
10. },
11. "Hàng tồn kho": {
12. "StockAlertThreshold": 20,
13. "WarehouseLocations": [
14. "New York",
15. "Luân Đôn",
16. "Tokyo"
17. ]
18. }
19. }

Trong ví dụ này:

* **Phần:** Các khóa cấp cao nhất (ConnectionStrings, Logging, Inventory) xác định các phần trong cấu hình.
* **Các phần lồng nhau:** Phần Ghi nhật ký còn chứa một phần LogLevel lồng nhau.
* **Mảng:** Cài đặt WarehouseLocations là một mảng chuỗi trong phần Hàng tồn kho.

**Truy cập cấu hình phân cấp với IConfiguration**

Giao diện IConfiguration cung cấp các phương thức để dễ dàng điều hướng và truy xuất các giá trị từ cấu trúc phân cấp này.

* **GetSection(khóa chuỗi):**
  + Trả về một đối tượng IConfigurationSection đại diện cho phần được chỉ định.
  + Sử dụng tùy chọn này để đi sâu vào các phần lồng nhau.
* **GetValue<T>(khóa chuỗi):**
  + Truy xuất giá trị cấu hình dưới dạng loại T được chỉ định.
  + Khóa có thể bao gồm toàn bộ đường dẫn đến giá trị, sử dụng dấu hai chấm (:) để phân tách các phần.
* **Trình lập chỉ mục (this[string key]):**
  + Truy xuất giá trị cấu hình dưới dạng chuỗi.
  + Hoạt động giống như phương thức GetValue<string>().

**Ví dụ về mã**

1. var connectionString = \_configuration. GetConnectionString("Kết nối mặc định");
3. var logLevel = \_configuration. GetValue<string>("Ghi nhật ký:LogLevel:Mặc định");
5. Sử dụng IConfigurationSection:
6. var inventorySection = \_configuration. GetSection("Hàng tồn kho");
7. var stockAlertThreshold = inventorySection.GetValue<int>("StockAlertThreshold");
9. Nhận một mảng
10. var warehouseLocations = inventorySection.GetSection("WarehouseLocations"). Get<string[]>();

**Các phương pháp hay nhất**

* **Cấu trúc rõ ràng:** Sắp xếp cài đặt của bạn thành các phần và phần phụ hợp lý để dễ đọc và bảo trì tốt hơn.
* **Đặt tên nhất quán:** Sử dụng các quy ước đặt tên có ý nghĩa và nhất quán cho các khóa cấu hình của bạn.
* **Nhập mạnh với mẫu tùy chọn:** Sử dụng mẫu Tùy chọn (IOptions<T>) để ánh xạ các phần cấu hình của bạn với các lớp được gõ mạnh, giúp đảm bảo an toàn kiểu và làm cho mã của bạn dễ làm việc hơn.
* **Biến môi trường:** Cân nhắc sử dụng các biến môi trường cho các cài đặt có thể khác nhau giữa các môi trường (ví dụ: ASPNETCORE\_ENVIRONMENT).
* **Quản lý bí mật:** Không bao giờ lưu trữ thông tin nhạy cảm (mật khẩu, khóa API) trực tiếp trong tệp cấu hình. Sử dụng Azure Key Vault, Trình quản lý bí mật hoặc các cơ chế bảo mật khác để quản lý bí mật.

**Ví dụ: Mô hình quyền chọn**

1. InventoryOptions.cs
2. lớp công cộng InventoryOptions
3. {
4. public int StockAlertThreshold { get; set; }
5. chuỗi công cộng[] WarehouseLocations { get; set; }
6. }
8. Program.cs (hoặc Startup.cs)
9. chủ thầu. Services.Configure<InventoryOptions>(trình tạo. Configuration.GetSection("Hàng tồn kho"));
11. Trong dịch vụ hoặc bộ điều khiển của bạn
12. InventoryService lớp công cộng: IInventoryService
13. {
14. InventoryOptions chỉ đọc riêng \_options;
16. tùy chọn InventoryService(IOptions<InventoryOptions> công khai)
17. {
18. \_options = tùy chọn. Giá trị;
19. }
21. // ... Sử dụng \_options. StockAlertThreshold và \_options. KhoVị trí
22. }

Mẫu tùy chọn

Mẫu Tùy chọn là một mẫu thiết kế trong ASP.NET Core cho phép bạn truy cập các giá trị cấu hình theo cách được nhập mạnh. Thay vì truy xuất các giá trị cấu hình dưới dạng chuỗi và chuyển đổi chúng thành các kiểu thích hợp theo cách thủ công, bạn xác định các lớp POCO (Plain Old CLR Object) đại diện cho cấu trúc của các phần cấu hình của bạn. Các lớp này, được gọi là lớp "tùy chọn", làm cho mã cấu hình của bạn dễ đọc hơn, dễ bảo trì hơn và ít bị lỗi hơn.

**Lợi ích của mô hình quyền chọn**

* **Truy cập kiểu mạnh:** Truy cập trực tiếp các giá trị cấu hình của bạn dưới dạng thuộc tính của các lớp tùy chọn, loại bỏ nhu cầu chuyển đổi kiểu thủ công và giảm nguy cơ lỗi thời gian chạy.
* **Hỗ trợ IntelliSense:** Hoàn thành mã và kiểm tra nhập trong IDE của bạn khi làm việc với cài đặt cấu hình của bạn.
* **Validation:** You can easily add validation logic to your options classes to ensure that configuration values are valid.
* **Clean Separation:** Keep your configuration settings separate from your business logic, improving the overall organization of your code.

**When to Use the Options Pattern**

* **Related Settings:** When you have groups of related configuration settings that logically belong together (e.g., database connection settings, email settings, feature flags).
* **Strongly Typed Access:** When you want to work with your configuration values in a type-safe manner.
* **Validation:** When you want to add validation logic to ensure your configuration values are valid.

**How to Implement the Options Pattern**

1. **Create an Options Class:** Define a class that mirrors the structure of your configuration section. Make sure the property names match the keys in your configuration file.
2. public class EmailOptions
3. {
4. public string SmtpServer { get; set; } = string.Empty;
5. public int SmtpPort { get; set; } = 25;
6. public string SenderEmail { get; set; } = string.Empty;
7. public string SenderPassword { get; set; } = string.Empty;
8. }
9. **Register the Options:** In your Program.cs (or Startup.cs in older versions), register your options class using the Configure<T> extension method on IServiceCollection:
10. builder.Services.Configure<EmailOptions>(builder.Configuration.GetSection("Email"));

This tells the DI container to bind the settings in the Email section of your configuration to an instance of EmailOptions.

1. **Inject IOptions<T>:** Inject the IOptions<T> interface into your controllers or services to access the bound options:
2. public class EmailService : IEmailService
3. {
4. private readonly EmailOptions \_options;
6. public EmailService(IOptions<EmailOptions> options)
7. {
8. \_options = options.Value;
9. }
11. // ... use \_options.SmtpServer, \_options.SmtpPort, etc. ...
12. }

**Related Methods for Configuration Access**

* **ConfigurationBinder.Get<T>(IConfiguration configuration):** Binds and returns the entire configuration section to a strongly typed object of type T.
* **ConfigurationBinder.Get(IConfiguration configuration, Type type):** Binds and returns the entire configuration section to an object of the specified type.
* **ConfigurationBinder.Bind(IConfiguration configuration, object instance):** Binds the configuration to an existing object instance.

**Example: Options Pattern with GetSection and Bind**

1. // Program.cs (or Startup.cs)
2. var emailOptions = new EmailOptions();
3. builder.Configuration.GetSection("Email").Bind(emailOptions);
4. builder.Services.AddSingleton(emailOptions); // Add the bound object as a singleton

Environment-Specific Configuration Files

ASP.NET Core allows you to create configuration files that are specific to different environments. By convention, these files are named appsettings.{Environment}.json, where {Environment} is replaced with the name of the environment (e.g., appsettings.Development.json, appsettings.Production.json).

**Purpose:**

* **Environment-Specific Settings:** These files store configuration values that are unique to each environment. This could include database connection strings, API keys, logging levels, or feature flags.
* **Customization:** You can tailor your application's behavior for development, testing, staging, and production environments without having to manually modify configuration settings every time you deploy.

**Order of Precedence:**

ASP.NET Core loads configuration from multiple sources, and the order in which they are loaded determines which values take precedence in case of conflicts. The general order of precedence (from highest to lowest) is:

1. **Command-Line Arguments:** Any configuration values specified as command-line arguments when you run your application (e.g., dotnet run --Logging:LogLevel:Default=Debug) override all other sources.
2. **Environment Variables:** Configuration values set as environment variables on your system take precedence over values in configuration files. ASP.NET Core automatically maps environment variables to configuration keys using a convention. For example, the environment variable ConnectionStrings\_\_DefaultConnection would map to the configuration key ConnectionStrings:DefaultConnection.
3. **User Secrets (Development Only):** If you're in the Development environment, values from the user secrets store (secrets.json) override those from appsettings.json and appsettings.Development.json. This is useful for storing sensitive information during development.
4. **appsettings.{Environment}.json:** If present, settings from this file override values from the base appsettings.json file. This allows you to customize settings for specific environments.
5. **appsettings.json:** This is the base configuration file that is always loaded. It contains the default settings for your application.

**Example: Overriding Connection Strings**

1. // appsettings.json
2. {
3. "ConnectionStrings": {
4. "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=MyDatabaseDev;Trusted\_Connection=True;"
5. }
6. }
8. // appsettings.Production.json
9. {
10. "ConnectionStrings": {
11. "DefaultConnection": "Server=myprodserver;Database=MyDatabaseProd;User Id=myuser;Password=mypassword;"
12. }
13. }

If the ASPNETCORE\_ENVIRONMENT variable is set to "Production", the connection string from appsettings.Production.json will be used.

**Code Example: GetSection() and GetValue()**

1. var connectionString = \_configuration.GetConnectionString("DefaultConnection");
3. var logLevel = \_configuration.GetValue<string>("Logging:LogLevel:Default");

* GetConnectionString("DefaultConnection") is a convenience method to fetch a connection string specifically from the ConnectionStrings section.
* GetValue<string>() retrieves values from specific configuration sections or keys.

**Best Practices**

* **Logical Structure:** Organize your settings into sections and subsections to make your configuration files easy to read and understand.
* **Consistent Naming:** Use consistent naming conventions for your configuration keys (e.g., kebab-case, snake\_case).
* **Environment Variables for Sensitive Data:** Store sensitive information like API keys and connection strings in environment variables or Azure Key Vault, not in configuration files that might be committed to source control.
* **User Secrets for Development:** Use user secrets to store sensitive data during development without exposing it in your code repository.
* **Order Matters:** Be mindful of the order of precedence when adding configuration sources. Place the most important or specific overrides later in the process.
* **Validation:** Consider validating your configuration during application startup to ensure that all required settings are present and have valid values.

Secrets Management in ASP.NET Core

In the world of web development, you'll often need to work with sensitive information like API keys, database connection strings, or passwords. Hardcoding these values directly into your source code is a security risk. That's where Secrets Manager comes into play.

**Secrets Manager: Your Digital Vault**

Secrets Manager is a tool that provides secure storage and management for your application's secrets. It keeps your sensitive data out of your source code and makes it easier to manage and rotate secrets without redeploying your application.

**User Secrets: Keeping Development Secrets Safe**

User Secrets is a developer-friendly feature of Secrets Manager specifically designed for local development environments. It allows you to store secrets for a particular project on your local machine without having to commit them to source control, keeping them out of your code repository.

**How to Set User Secrets Using the dotnet Command**

1. **Initialize:** If you haven't already, initialize user secrets for your project:
   1. dotnet user-secrets init

This command adds a UserSecretsId property to your project's .csproj file, which links the project to a user secrets store.

1. **Set a Secret:** Use the set command to store a secret:
   1. dotnet user-secrets set "MySecretName" "MySecretValue"

Replace "MySecretName" with the desired key and "MySecretValue" with the actual secret value.

1. **List Secrets (Optional):**
   1. dotnet user-secrets list

This command lists all the secrets you've stored for the project.

1. **Remove a Secret (Optional):**
   1. dotnet user-secrets remove "MySecretName"

**Accessing User Secrets in Your Code**

1. var builder = WebApplication.CreateBuilder(args);
2. var configuration = builder.Configuration;
4. // In Program.cs (or Startup.cs):
5. if (builder.Environment.IsDevelopment())
6. {
7. configuration.AddUserSecrets<Program>();
8. }

This will add a configuration source that can read user secrets, but only when the environment is set to "Development".

Then, to access a user secret, you can use the same techniques you would for any other configuration value:

1. var mySecret = configuration["MySecretName"];

**Best Practices for Secrets Management**

* **Never Hardcode Secrets:** Always store sensitive information in a secure store like Secrets Manager.
* **Least Privilege:** Grant your application the minimum necessary permissions to access secrets.
* **Rotate Secrets Regularly:** Regularly change your secrets to minimize the risk of exposure.
* **Separate Environments:** Use different secrets for different environments (development, staging, production).
* **Automation:** Consider automating the process of secret rotation to enhance security.

**Example: Storing an API Key as a User Secret**

1. **Initialize:** dotnet user-secrets init
2. **Set Secret:** dotnet user-secrets set "StripeApiKey" "sk\_test\_1234567890"

**Accessing in Your Code (Example):**

1. var stripeApiKey = configuration["StripeApiKey"];

**Caveats**

* **Development Only:** User secrets are intended for development environments and should not be used in production.
* **Local Storage:** User secrets are stored in a JSON file on your local machine. Ensure this file is protected.

Set Configuration Values from Environment Variables

* **Flexibility:** You can dynamically change your application's settings without modifying code or configuration files.
* **Security:** Environment variables are a secure way to store sensitive information like API keys, connection strings, or passwords without embedding them in your code.
* **Deployment Environments:** Different environments (development, staging, production) often require distinct configuration values. Environment variables can be easily set and managed per environment.
* **Automation:** This approach lends itself well to automation scripts for deployment and configuration.

**How It Works**

1. **Environment Variable Prefix:** ASP.NET Core's configuration system recognizes environment variables that start with a specific prefix, by default, ASPNETCORE\_. This allows you to namespace your environment variables to avoid conflicts with other variables on your system.
2. **Key Mapping:** The part of the environment variable name after the prefix is used as the configuration key. For example, the environment variable ASPNETCORE\_Logging\_\_LogLevel\_\_Default will map to the configuration key Logging:LogLevel:Default. Double underscores (\_\_) are used to represent colons (:) in the hierarchy.
3. **Configuration Provider:** ASP.NET Core has a built-in configuration provider called EnvironmentVariablesConfigurationProvider that automatically reads these environment variables and adds them to the configuration system.

**Setting Environment Variables from the Command Line**

**PowerShell (Windows, macOS, Linux)**

1. $env:ASPNETCORE\_MyKey = "myvalue" # Simple key-value
2. $env:ASPNETCORE\_Logging\_\_LogLevel\_\_Default = "Debug" # Hierarchical key

In PowerShell, use the $env: prefix to set environment variables within the current session.

**Command Prompt (Windows)**

1. set ASPNETCORE\_MyKey=myvalue # Simple key-value
2. set ASPNETCORE\_Logging\_\_LogLevel\_\_Default=Debug # Hierarchical key

**Bash (macOS, Linux)**

1. export ASPNETCORE\_MyKey="myvalue" # Simple key-value
2. export ASPNETCORE\_Logging\_\_LogLevel\_\_Default="Debug" # Hierarchical key

**Example: Setting a Database Connection String**

Let's say you want to set your database connection string using an environment variable. Here's how you would do it:

1. **Set the Environment Variable:**
   1. # In PowerShell
   2. $env:ASPNETCORE\_ConnectionStrings\_\_DefaultConnection = "Server=myServer;Database=myDb;Trusted\_Connection=True;"
   4. # In Command Prompt (Windows)
   5. set ASPNETCORE\_ConnectionStrings\_\_DefaultConnection="Server=myServer;Database=myDb;Trusted\_Connection=True;"
   7. # In Bash (macOS/Linux)
   8. export ASPNETCORE\_ConnectionStrings\_\_DefaultConnection="Server=myServer;Database=myDb;Trusted\_Connection=True;"

Note the double underscores (\_\_) used to represent the colon (:) in the configuration path.

1. **Access in Your Code:** You can then retrieve this connection string in your ASP.NET Core application using:
2. var connectionString = \_configuration.GetConnectionString("DefaultConnection");

**Notes**

* **Prefix:** Remember to use the ASPNETCORE\_ prefix for your environment variables.
* **Key Mapping:** Double underscores (\_\_) in the environment variable name are translated to colons (:) in the configuration key.
* **Override:** Environment variable values will override those set in appsettings.json or appsettings.{Environment}.json.
* **Sensitive Data:** This is an excellent way to manage sensitive data without exposing it in your code or configuration files.
* **Deployment:** Make sure to set the appropriate environment variables on your production server before deploying your application.

The Mechanics of Environment Variable Configuration

1. **Environment Variable Prefix:** ASP.NET Core's configuration system recognizes environment variables that start with a specific prefix. By default, this prefix is ASPNETCORE\_. You can customize this prefix if needed. This prefix helps to namespace your environment variables and avoid conflicts with other variables on your system.
2. **Key Mapping:** The part of the environment variable name after the prefix is used as the configuration key. A double underscore (\_\_) is used to represent a colon (:) in the hierarchical structure of your configuration. For example:
   * Environment Variable: ASPNETCORE\_Logging\_\_LogLevel\_\_Default
   * Configuration Key: Logging:LogLevel:Default
3. **Configuration Provider:** ASP.NET Core includes a built-in configuration provider called EnvironmentVariablesConfigurationProvider. This provider automatically reads environment variables that match the prefix and adds them to the application's configuration. The values from environment variables override any matching values found in appsettings.json or environment-specific configuration files.

**Setting Environment Variables from the Command Line**

**PowerShell (Windows, macOS, Linux)**

1. $env:ASPNETCORE\_MyKey = "myvalue" # Simple key-value
2. $env:ASPNETCORE\_Logging\_\_LogLevel\_\_Default = "Debug" # Hierarchical key

**Command Prompt (Windows)**

1. set ASPNETCORE\_MyKey=myvalue # Simple key-value
2. set ASPNETCORE\_Logging\_\_LogLevel\_\_Default=Debug # Hierarchical key

**Bash (macOS, Linux)**

1. export ASPNETCORE\_MyKey="myvalue" # Simple key-value
2. export ASPNETCORE\_Logging\_\_LogLevel\_\_Default="Debug" # Hierarchical key

**Example: Setting a Database Connection String**

Let's say you want to set your database connection string using an environment variable. Here's how you would do it:

1. **Set the Environment Variable:**
   1. # In PowerShell or Bash
   2. $env:ASPNETCORE\_ConnectionStrings\_\_DefaultConnection = "Server=myServer;Database=myDb;User Id=myuser;Password=mypassword;"
   4. # In Command Prompt (Windows)
   5. set ASPNETCORE\_ConnectionStrings\_\_DefaultConnection="Server=myServer;Database=myDb;User Id=myuser;Password=mypassword;"
2. **Access in Your Code:** You can then retrieve this connection string in your ASP.NET Core application as usual:
   1. var connectionString = \_configuration.GetConnectionString("DefaultConnection");

**Important Considerations**

* **Prefix Customization:** You can change the default ASPNETCORE\_ prefix using the AddEnvironmentVariables method. For example, configuration.AddEnvironmentVariables("CUSTOM\_PREFIX\_");
* **Case Sensitivity:** On Linux and macOS, environment variable names are case-sensitive.
* **Deployment:** When deploying your application, ensure that the appropriate environment variables are set on the target server.
* **Security:** While environment variables are more secure than hardcoding values, they might not be suitable for extremely sensitive secrets. In those cases, consider using a dedicated secret management solution like Azure Key Vault or HashiCorp Vault.

Custom JSON Files

While ASP.NET Core natively supports appsettings.json and environment-specific variations, there are scenarios where using custom JSON files for configuration might be advantageous:

* **Modularity:** You can organize settings into multiple files based on functional areas or components, making your configuration more manageable and easier to navigate.
* **Customization:** You can load custom JSON files conditionally, based on specific requirements or runtime decisions.
* **Separation of Concerns:** This approach allows you to keep default settings in appsettings.json while maintaining custom settings separately.

**Adding Custom JSON Files as Configuration Sources**

1. **Create the File:** Create a JSON file with your custom configuration settings. Let's call it customsettings.json:
2. {
3. "CustomSettings": {
4. "APIKey": "your\_api\_key",
5. "FeatureEnabled": true,
6. "NotificationSettings": {
7. "EmailEnabled": true,
8. "SMSEnabled": false
9. }
10. }
11. }
12. **Add to Configuration:** In your Program.cs, use the AddJsonFile extension method to include your custom JSON file:
13. var builder = WebApplication.CreateBuilder(args);
14. var configuration = builder.Configuration;
16. // ... (other configuration sources) ...
18. // Add the custom JSON file:
19. configuration.AddJsonFile("customsettings.json", optional: true, reloadOnChange: true);
21. var app = builder.Build();
22. // ... (rest of the application) ...

* **optional: true:** Set this to true if the file might not exist (e.g., in certain environments).
* **reloadOnChange: true:** Enables automatic reloading of the configuration if the file changes.

**Accessing Custom Configuration Values**

You can access values from your custom JSON file using the same mechanisms as you would for appsettings.json:

1. // Option 1: Directly using IConfiguration
2. var apiKey = configuration["CustomSettings:APIKey"];
3. var featureEnabled = configuration.GetValue<bool>("CustomSettings:FeatureEnabled");
5. // Option 2: Options Pattern
6. var notificationSettings = configuration.GetSection("CustomSettings:NotificationSettings").Get<NotificationSettings>();

**Best Practices**

* **Naming:** Choose descriptive and meaningful names for your custom JSON files.
* **Organization:** Structure your custom configuration files with sections and subsections to enhance readability and maintainability.
* **Environment-Specific Overlays:** Create environment-specific versions of your custom files (e.g., customsettings.Development.json) to override settings in different environments.
* **Secrets Management:** Store sensitive information (API keys, passwords) in a secure store like Azure Key Vault or User Secrets.
* **Error Handling:** Handle potential errors, such as missing or invalid configuration files, gracefully.
* **Strong Typing with Options:** Strongly recommend using Options Pattern for type safety and better code structure.

**Example: Options Pattern with Custom JSON File**

1. // CustomSettings.cs (Options Class)
2. public class CustomSettings
3. {
4. public string APIKey { get; set; }
5. public bool FeatureEnabled { get; set; }
6. public NotificationSettings NotificationSettings { get; set; }
7. }
9. // ... (other options classes if needed) ...
11. // Program.cs
12. builder.Services.Configure<CustomSettings>(configuration.GetSection("CustomSettings"));
14. // In your controller or service
15. public class MyController : Controller
16. {
17. private readonly CustomSettings \_settings;
19. public MyController(IOptions<CustomSettings> settings)
20. {
21. \_settings = settings.Value;
22. }
23. }

HttpClient

The HttpClient class is a powerful and versatile tool in the .NET ecosystem for interacting with web-based resources over the HTTP protocol. You use it to send requests (GET, POST, PUT, DELETE, etc.) to APIs and retrieve responses containing data in various formats (JSON, XML, HTML).

**Key Features of HttpClient**

* **Sending Requests:** Craft and send HTTP requests to any URL.
* **Receiving Responses:** Process the server's response (status code, headers, body content).
* **Async Operations:** Designed for asynchronous programming, allowing your application to perform other tasks while waiting for network responses.
* **Customization:** Configure request headers, timeouts, authentication, and more.

**Using HttpClient in ASP.NET Core**

While you can create and manage HttpClient instances directly, ASP.NET Core offers a more robust approach through the IHttpClientFactory interface. The factory handles the following for you:

* **Connection Pooling:** Manages a pool of HTTP connections, optimizing performance and preventing socket exhaustion.
* **Lifetime Management:** Ensures proper disposal of HttpClient instances to avoid resource leaks.
* **Named Clients:** Lets you define and configure named clients for different APIs, each with its own settings (base address, headers, etc.).

**Integrating HttpClient with Your Stock App**

Let's analyze how your stock application uses HttpClient and IHttpClientFactory:

1. **FinnhubService:**
   * **Injection:** The constructor injects IHttpClientFactory to create HttpClient instances.
   * **Request Building:** The GetStockPriceQuote method constructs an HttpRequestMessage object, specifying the URL (including the Finnhub API token) and the HTTP method (GET).
   * **Sending the Request:** It uses httpClient.SendAsync to send the request asynchronously.
   * **Response Processing:** It reads the response content as a stream and deserializes the JSON data into a dictionary.
   * **Error Handling:** It checks for errors in the response and throws exceptions accordingly.
2. **HomeController:**
   * **Injection:** It injects both the FinnhubService and IOptions<TradingOptions> for configuration.
   * **Data Fetching:** The Index action calls \_finnhubService.GetStockPriceQuote to get stock data.
   * **Model Creation:** It maps the retrieved data to a Stock model object.
   * **View Rendering:** The Stock model is passed to the view for display.

**Code Breakdown**

* **IFinnhubService:** Defines an interface for the Finnhub service, allowing for different implementations if needed.
* **FinnhubService:** Implements the interface and uses HttpClient to interact with the Finnhub API.
* **TradingOptions:** A class to hold configuration options for the default stock symbol (read from appsettings.json).
* **Stock:** A model class to represent the stock data.
* **HomeController:** The controller that fetches stock data and renders the view.

**Best Practices**

* **IHttpClientFactory:** Always use IHttpClientFactory instead of directly creating HttpClient instances to benefit from connection pooling and proper lifetime management.
* **Named Clients:** For multiple APIs, use named clients (\_httpClientFactory.CreateClient("name");) to configure different settings for each API.
* **Error Handling:** Handle exceptions that might occur during HTTP requests, such as network errors or invalid responses.
* **Resilience:** Consider using Polly or other libraries to implement retries and circuit breaker patterns for increased resilience in the face of transient errors.

Key Points to Remember

* **Purpose:** Provide named configurations to tailor your app's behavior for different scenarios (development, staging, production, etc.).
* **Environment Variable:**
  + ASPNETCORE\_ENVIRONMENT is the key environment variable.
  + Its value determines the active environment.
* **Setting the Environment:**
  + **launchSettings.json (Development):** Set within the environmentVariables section of a profile.
  + **System Environment Variables:** Set directly on your machine (persistent).
  + **Command Line:** Use --environment or -e flag when running the app (e.g., dotnet run --environment Staging).
  + **Azure App Service:** In the Azure portal, under Configuration > Application settings.
* **IWebHostEnvironment Interface:**
  + Use it in your code to access environment information (e.g., EnvironmentName, WebRootPath).
  + Inject it into your controllers or middleware:
  + private readonly IWebHostEnvironment \_env;
  + public MyController(IWebHostEnvironment env)
  + {
  + \_env = env;
  + }
* **Environment-Specific Configuration:**
  + Create files like appsettings.Development.json, appsettings.Staging.json, etc.
  + ASP.NET Core automatically loads the appropriate file based on the environment.
  + Override base settings in appsettings.json.
* **Conditional Configuration (In Program.cs):**
  + Use if (app.Environment.IsDevelopment()) or similar methods to apply settings or middleware based on the environment.
  + if (app.Environment.IsDevelopment())
  + {
  + app.UseDeveloperExceptionPage();
  + }
* **Default Environments:**
  + Development: Default for local development.
  + Staging: Typically used for pre-production testing.
  + Production: The live environment.
* **Custom Environments:** You can define and use your own environment names.
* **Best Practices:**
  + **Separate Configurations:** Keep environment-specific settings in separate files.
  + **Tailor Middleware:** Use different middleware pipelines for different environments (e.g., enable DeveloperExceptionPage only in development).
  + **Logging:** Adjust logging levels based on the environment.
  + **Feature Flags:** Use environment variables to toggle features on/off.

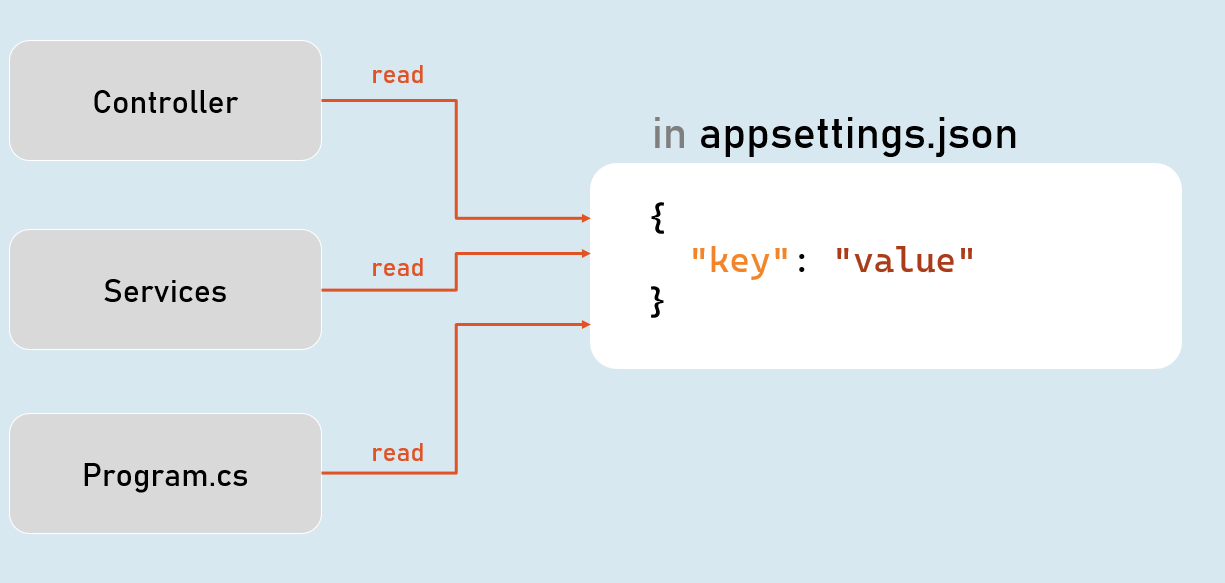
**Interview Tips**

* **Explain the Why:** Be able to articulate the reasons for using environments (configuration, security, flexibility).
* **Configuration:** Show how you would use appsettings.{Environment}.json files to manage environment-specific settings.
* **Middleware:** Explain how you would customize middleware pipelines based on the environment.
* **Deployment:** Discuss how you would set the environment variable when deploying to different servers.

**Configuration Settings**

**Configuration (or configuration settings) are the constant key/value pairs that are set at a common location and can be read from anywhere in the same application.**

**Examples: connection strings, Client ID & API keys to make REST-API calls, Domain names, Constant email addresses etc.**

****

**Configuration Sources**

1. **appsettings.json**
2. **Environment Variables**
3. **File Configuration (JSON, INI or XML files)**
4. **In-Memory Configuration**
5. **Secret Manager**

**Access Configuration**

**in Program.cs:**

**app.Configuration**

**IConfiguration**

**[string key]**

**Gets or sets configuration value at the specified key.**

**GetValue<T>(string key, object defaultValue)**

**Gets the configuration value at the specified key; returns the default value if the key doesn't exists.**

**IConfiguration in Controller**

**in Controller and other classes**

1. **using Microsoft.AspNetCore.Mvc;**
2. **using Microsoft.Extensions.Configuration;**
4. **public class ControllerName : Controller**
5. **{**
6. **private readonly IConfiguration \_configuration;**
8. **public ControllerName(IConfiguration configuration)**
9. **{**
10. **\_configuration = configuration;**
11. **}**
12. **}**

**Hierarchical Configuration**

**in appsettings.json**

1. **{**
2. **"MasterKey":**
3. **{**
4. **"Key1": "value"**
5. **"Key2": "value"**
6. **}**
7. **}**

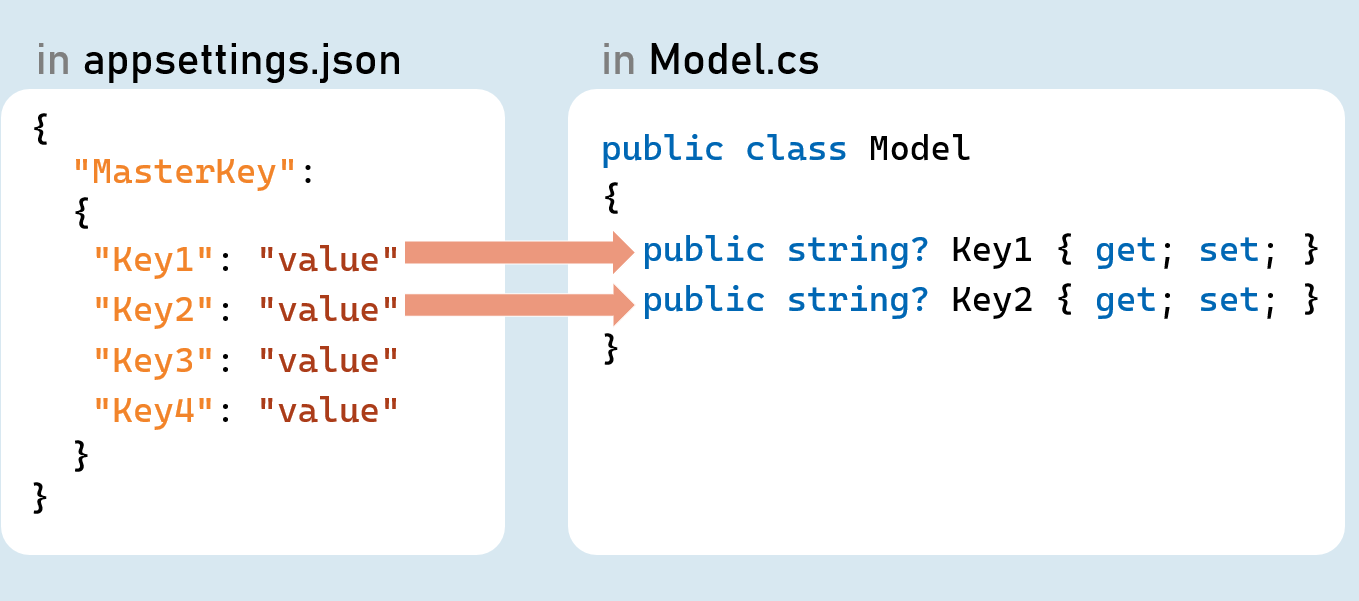
**to read configuration**

**Configuration["MasterKey:Key1"]**

**IConfiguration.GetSection(string key)**

**Returns an IConfigurationSection based on the specified key.**

**Options Pattern**

****

**Options pattern uses custom classes to specify what configuration settings are to be loaded into properties.**

**Examples: Reading the specific connections strings out of many configuration settings.**

**The option class should be a non-abstract class with a public parameterless constructor.**

**Public read-write properties are bound.**

**Fields are not bound.**

**IConfiguration.GetSection(string key)**

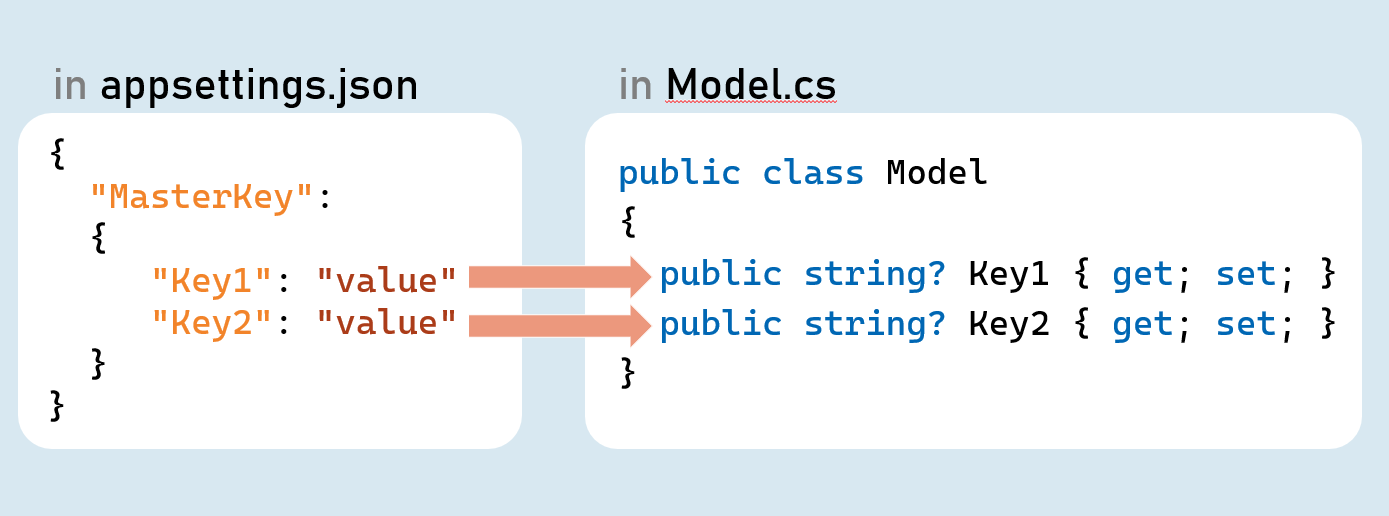
**Returns an IConfigurationSection based on the specified key.**

**IConfiguration.Bind(object instance) and IConfiguration.Get<T>()**

**Binds (loads) configuration key/value pairs into a new object of the specified type.**

**Configuration as Service**

**Inject Configuration as Service**

****

**Add Configuration as Service**

**in Program.cs:**

**builder.Services.Configure<Model>(builder.Configuration.GetSection("MasterKey"));**

**Inject Configuration as Service in Controller in Controller and other classes**

1. **using Microsoft.AspNetCore.Mvc;**
2. **using Microsoft.Extensions.Options;**
4. **public class ControllerName : Controller**
5. **{**
6. **private readonly Model \_options;**
8. **public ControllerName(IOptions<Model> options)**
9. **{**
10. **\_options = options.Value;**
11. **}**
12. **}**

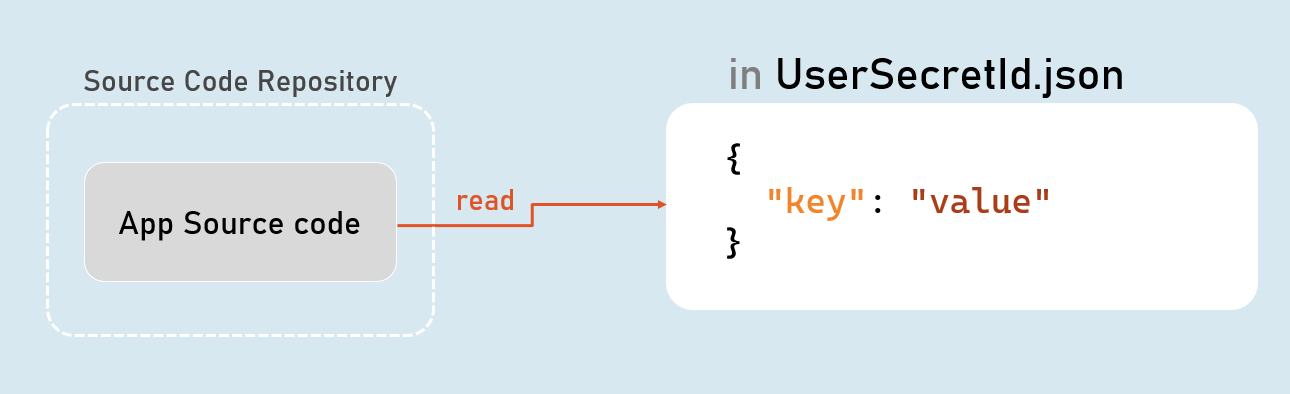
**Environment Specific Configuration**

**Order of Precedence of Configuration Sources**

****

**Secrets Manager**

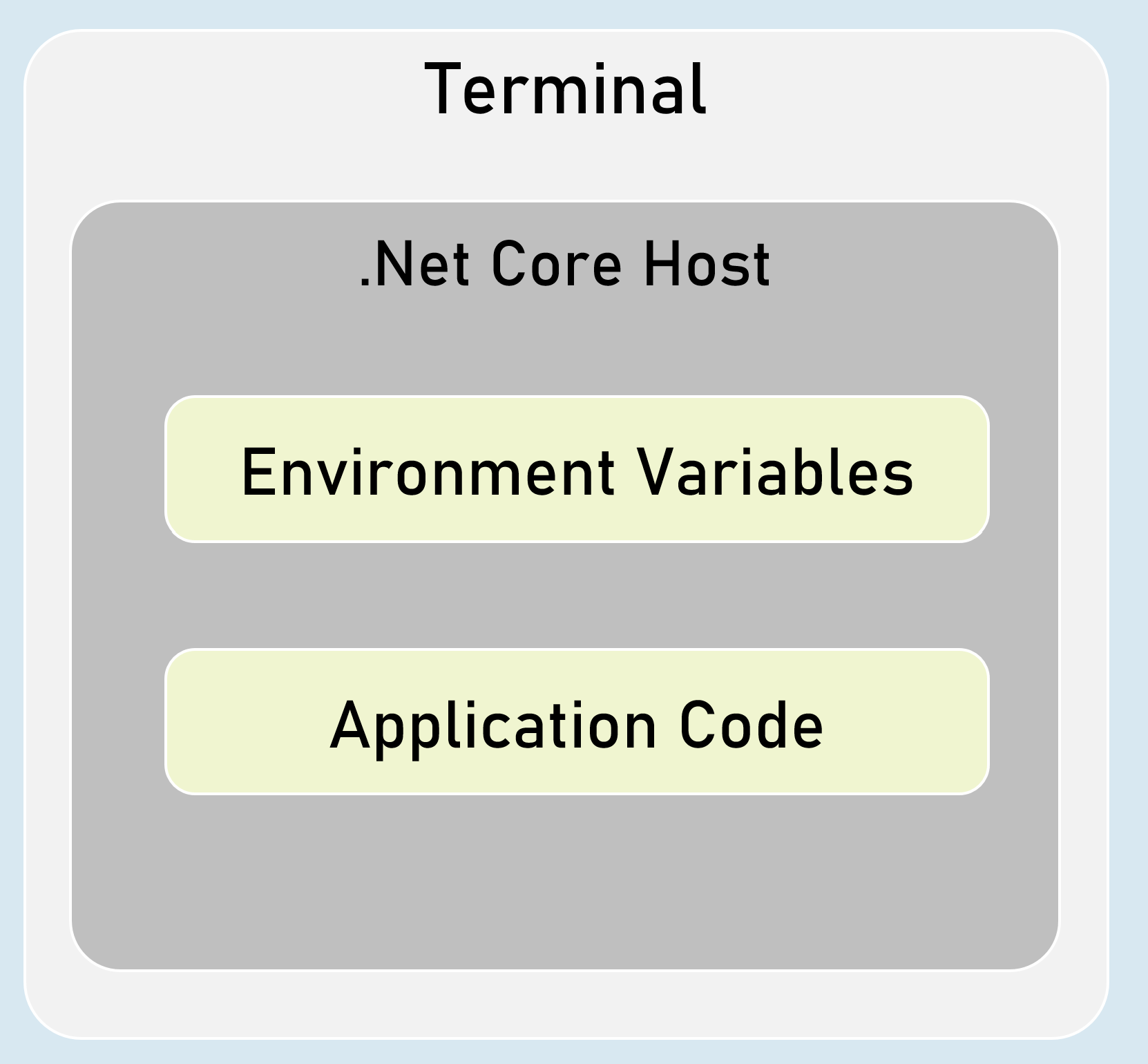
**The 'secrets manager ' stores the user secrets (sensitive configuration data) in a separate location on the developer machine.**

****

**Enable Secrets Manager in "Windows PowerShell" / "Developer PowerShell in VS"**

1. **dotnet user-secrets init**
2. **dotnet user-secrets set "Key" "Value"**
3. **dotnet user-secrets list**

**Environment Variables Configuration**

****

**You can set configuration values as in-process environment variables.**

**Set Configuration as Environment Variables**

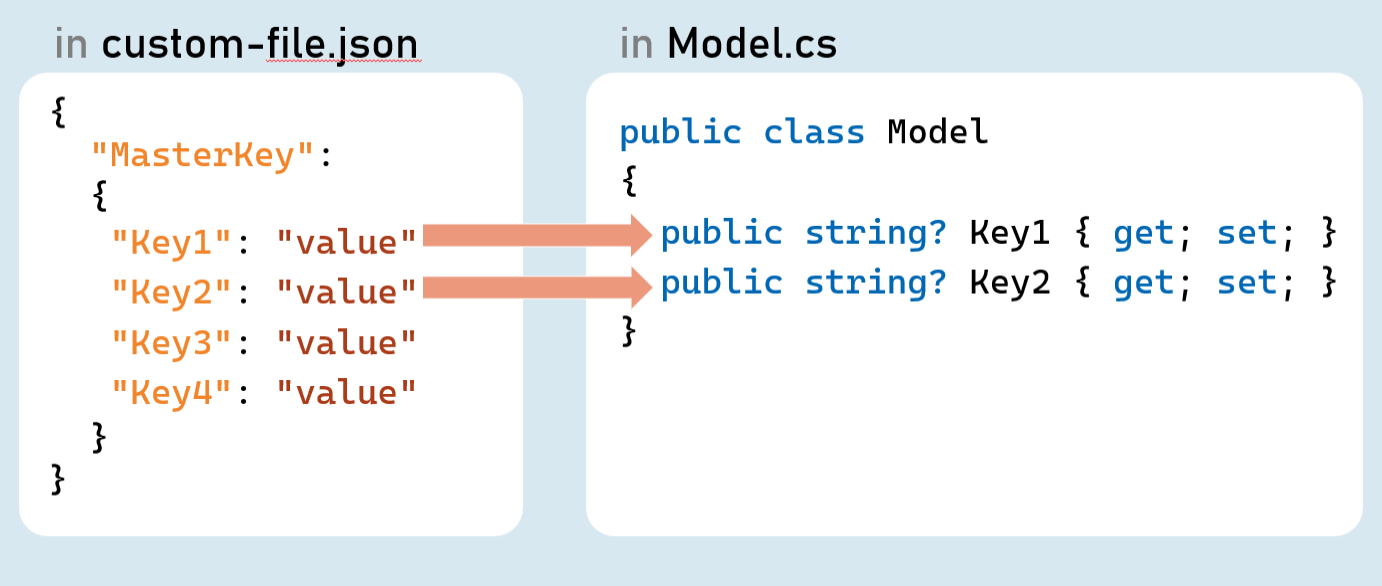
**in "Windows PowerShell" / "Developer PowerShell in VS":**

1. **$Env:ParentKey\_\_ChildKey="value"**
2. **dotnet run --no-launch-profile**

**It is one of the most secured way of setting-up sensitive values in configuration.**

**\_\_ (underscore and underscore) is the separator between parent key and child key.**

**Custom Json Configuration**

****

**Add Custom Json file as Configuration Source**

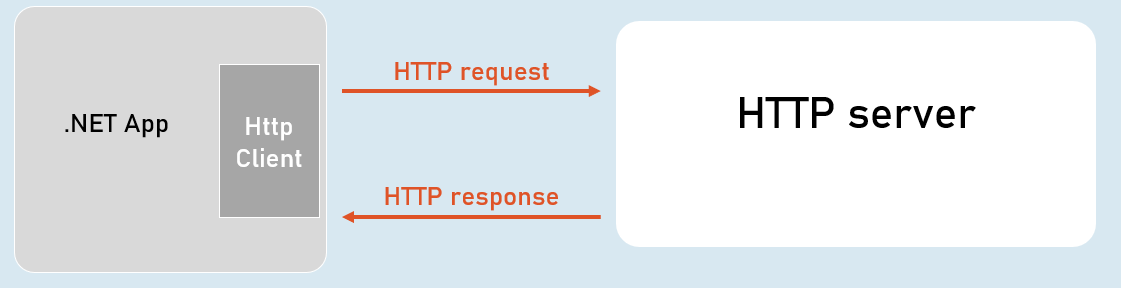
**in Program.cs:**

1. **builder.Host.ConfigureAppConfiguration( (hostingContext, config) => {**
2. **config.AddJsonFile("filename.json", optional: true, reloadOnChange: true);**
3. **});**

**Http Client**

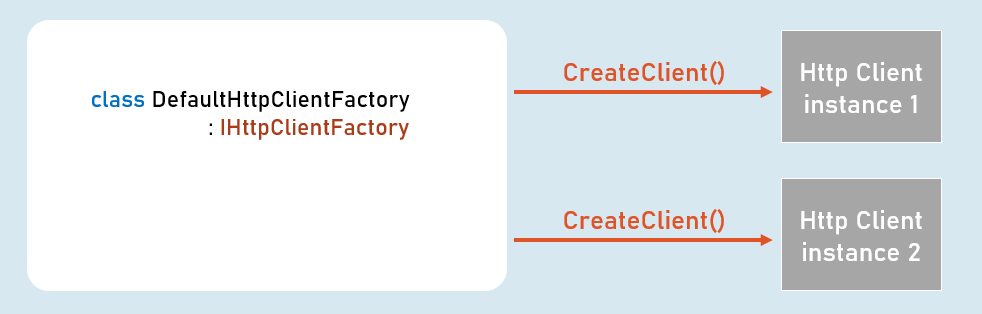
**HttpClient is a class for sending HTTP requests to a specific HTTP resource (using its URL) and receiving HTTP responses from the same.**

**Examples: Making a request to a third-party weather API, ChatGPT etc.**

****

**IHttpClientFactory**

**IHttpClientFactory is an interface that provides a method called CreateClient() that creates a new instance of HttpClient class and also automatically disposes the same instance (closes the connection) immediately after usage.**

****

**HttpClient**

**Properties**

* **BaseAddress**
* **DefaultRequestHeaders**

**Methods**

* **GetAsync()**
* **PostAsync()**
* **PutAsync()**
* **DeleteAsync()**

# **13. xUnit [MVC vs API]**

# **14. CRUD Operations [MVC]**

# **15. Tag Helpers [MVC]**

# **16. Entity Framework core [MVC vs API]**

# **17. Advanced Unit Testing – Moq and Repository Pattern [MVC vs API]**

# **18. Logging and Serilog [MVC vs API]**

# **19. Filter [MVC vs API]**

# **20. Error handling [MVC vs API]**

# **21. SOLID Principles [MVC vs API]**

# **22. Clean Architecture [MVC vs API]**

# **23. Identity , Authorization, Security [MVC vs API]**

# **24. Asp.net Core Web API [API]**

# **25. Swagger and OpenAPI [API]**

# **26. Angular and CORS [API]**

# **27. JWT and Web API Authentication [API]**

# **28. Minimal API [API]**