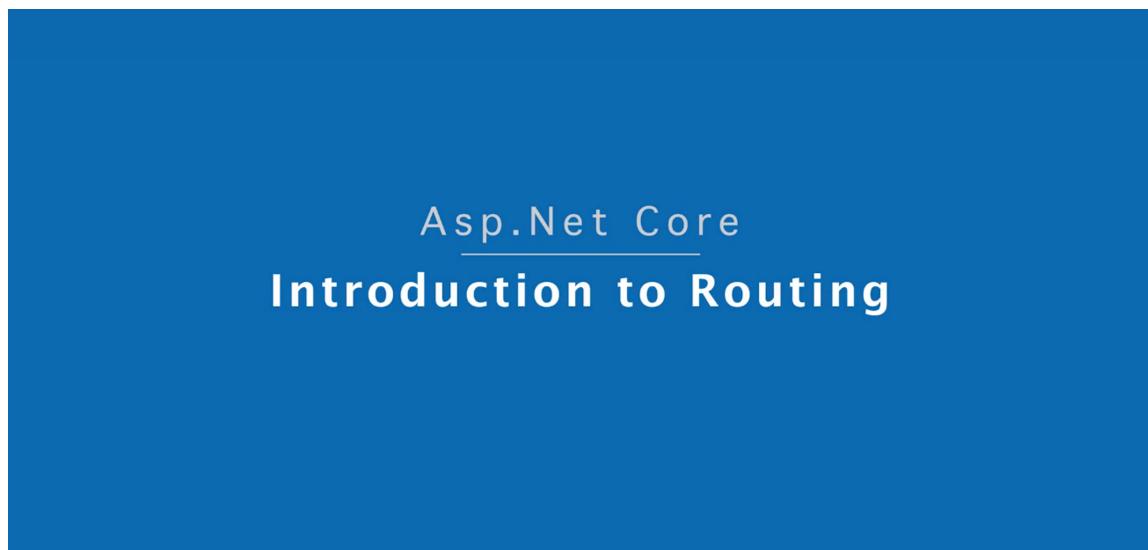


## 1. Intro to Routing



Asp.Net Core Harsha

## Introduction to Routing

**Routing is a process of matching incoming HTTP requests by checking the HTTP method and url; and then invoking corresponding endpoints.**

The diagram illustrates the routing process. On the left, two arrows point to a central blue box labeled "Routing". One arrow is labeled "Request to /url1" and the other is labeled "Request to /url2". From the "Routing" box, two arrows emerge: one labeled "Request to /url1" pointing to a box labeled "Endpoint 1", and another labeled "Request to /url2" pointing to a box labeled "Endpoint 2".

Routing is the process of matching incoming HTTP requests (based on URL and HTTP method) to corresponding endpoints, enabling the server to invoke the right functionality.

### How It Works:

- You define a list of URLs and their corresponding endpoints in your application.
- When an HTTP request arrives:
  1. The server checks the request's URL and HTTP method (e.g., GET, POST).
  2. It matches the request with the appropriate URL in the list.
  3. If a match is found, the associated endpoint (middleware or controller) is invoked.

### Example:

- A user requests "/home" → The Home Page endpoint is triggered.
- A user requests "/about" → The About Page endpoint is triggered.

This process enables the application to serve different pages or data dynamically.

### Key Notes:

- Endpoints in ASP.NET Core are middleware that handle specific functionality or return specific responses.
- Routing is essential for serving tailored content based on the URL structure.
- Example URLs and Mapping:
  - URL: /home → Endpoint: HomeController
  - URL: /about → Endpoint: AboutController

## Routing - UseRouting and UseEndPoints

### UseRouting( )

```
app.UseRouting();
```

Enables routing and selects an appropriate end point based on the url path and HTTP method.

### UseEndPoints( )

```
app.UseEndpoints(endpoints =>
{
    endpoints.Map(...);
    endpoints.MapGet(...);
    endpoints.MapPost(...);
});
```

Executes the appropriate endpoint based on the endpoint selected by the **above** a UseRouting() method.

Udemy

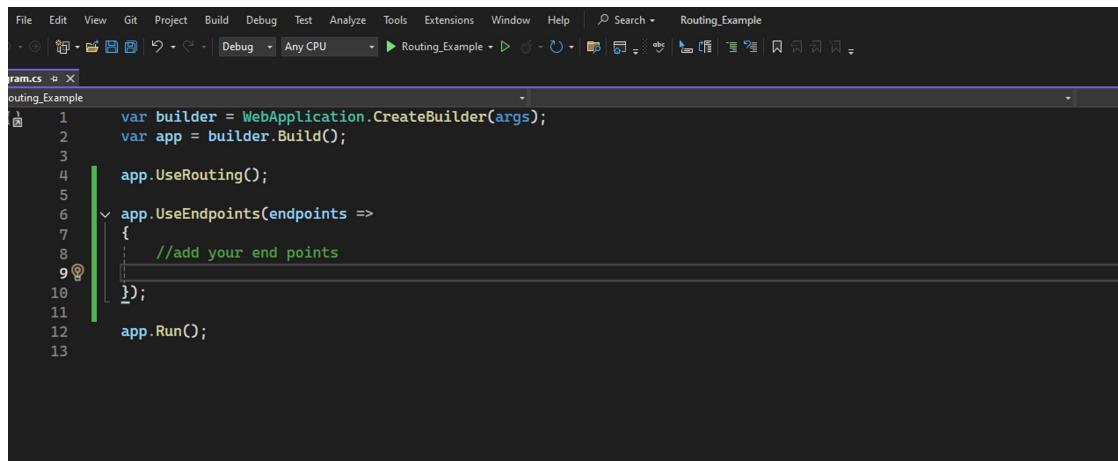
With two individual methods, that is "use routing" and "use endpoints".

You would like to invoke them in the same order, that is first "use routing" and after that only "use endpoints".

So, what's the difference between them?

"use routing" enables routing in your application and selects the appropriate endpoint based on the incoming request. It mainly considers the URL path and also the HTTP method—either GET, POST, PUT, or DELETE. But it only just selects the appropriate endpoint; it doesn't execute that endpoint.

The "use endpoints" method will actually execute the appropriate endpoint that was selected by the "use routing".



A screenshot of the Visual Studio IDE showing the `Program.cs` file for a project named "Routing\_Example". The code demonstrates the use of the `WebApplication.CreateBuilder` method to build a web application. It includes calls to `app.UseRouting()` and `app.UseEndpoints`, which takes a lambda expression for defining endpoints. The code ends with `app.Run()`. The code editor shows syntax highlighting and line numbers from 1 to 13.

```
File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help | Search Routing Example  
Program.cs | X Routing_Example | Debug Any CPU | Routing_Example | |  
1 var builder = WebApplication.CreateBuilder(args);  
2 var app = builder.Build();  
3  
4 app.UseRouting();  
5  
6 app.UseEndpoints(endpoints =>  
7 {  
8     //add your end points  
9 }  
10  
11 app.Run();  
12  
13
```

When the "use routing" method executes, the appropriate endpoint will be selected because, at the time of compilation itself, all the endpoints are defined and stored in the compiled source code. So, based on the incoming request, the appropriate endpoint will be selected when this "use routing" method executes.

But the actual endpoint will not be executed. It will be executed when we call the "use endpoints". Here, you require to pass a lambda expression that contains one argument called "endpoints", and within here, you require to add your endpoints.

## Asp.Net Core

---

### **Map(), MapGet(), MapPost()**

## Routing - UseRouting and UseEndPoints

### UseRouting()

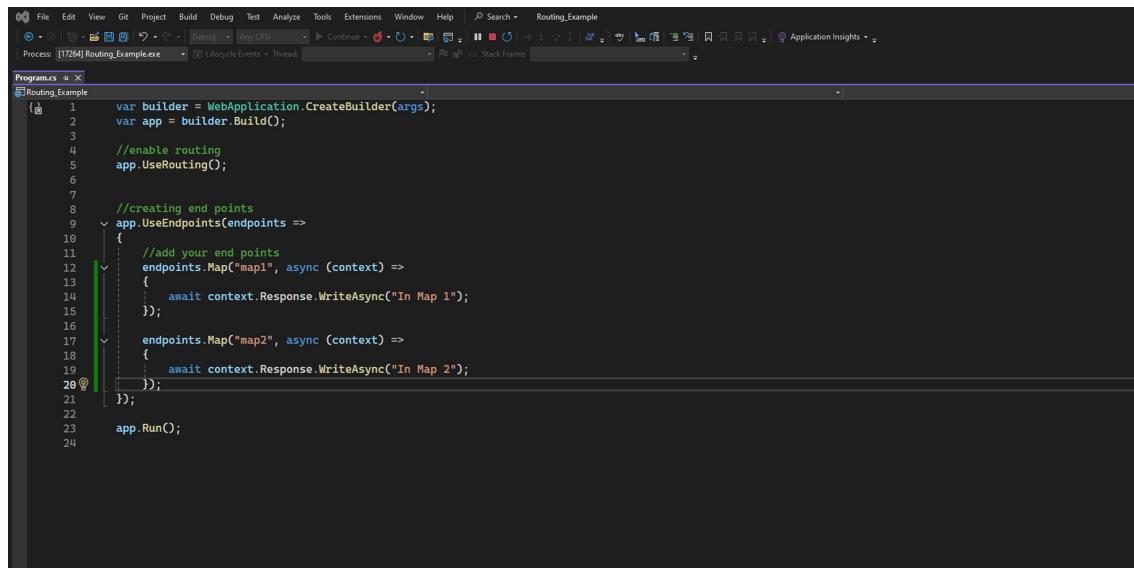
```
app.UseRouting();
```

Enables routing and selects an appropriate end point based on the url path and HTTP method.

### UseEndPoints( )

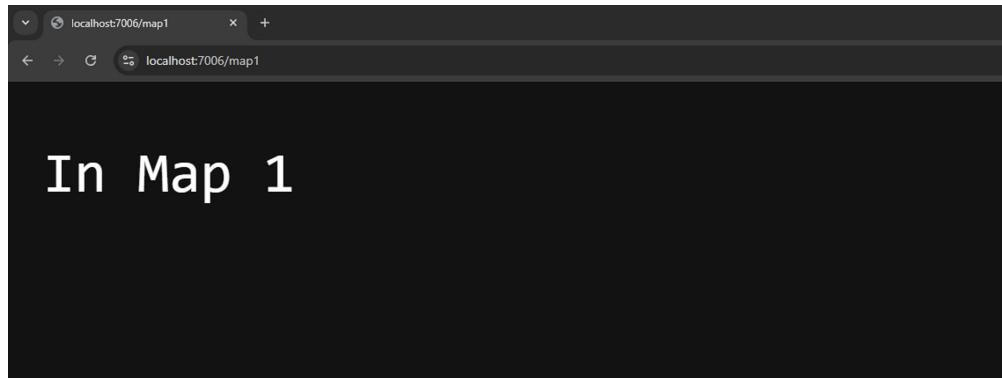
```
app.UseEndpoints(endpoints =>
{
    endpoints.Map(...);
    endpoints.MapGet(...);
    endpoints.MapPost(...);
});
```

Executes the appropriate endpoint based on the endpoint selected by the above UseRouting() method.

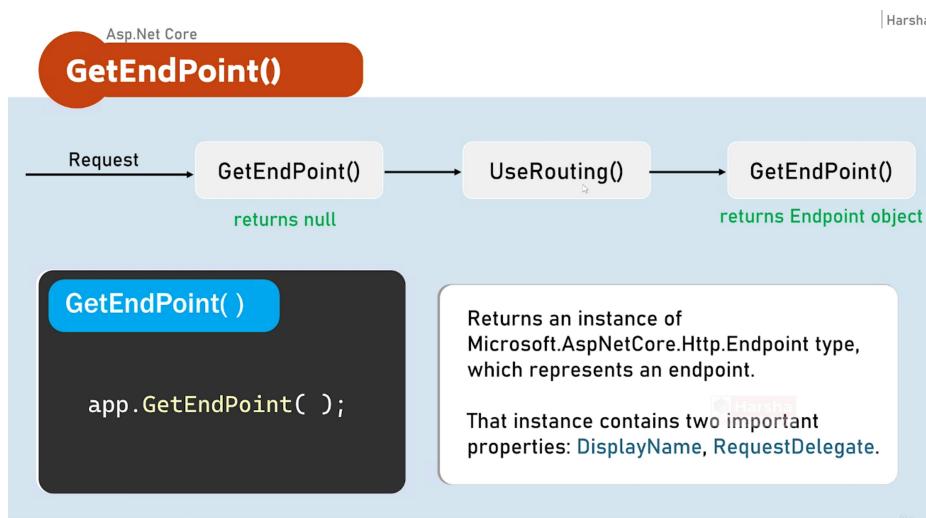
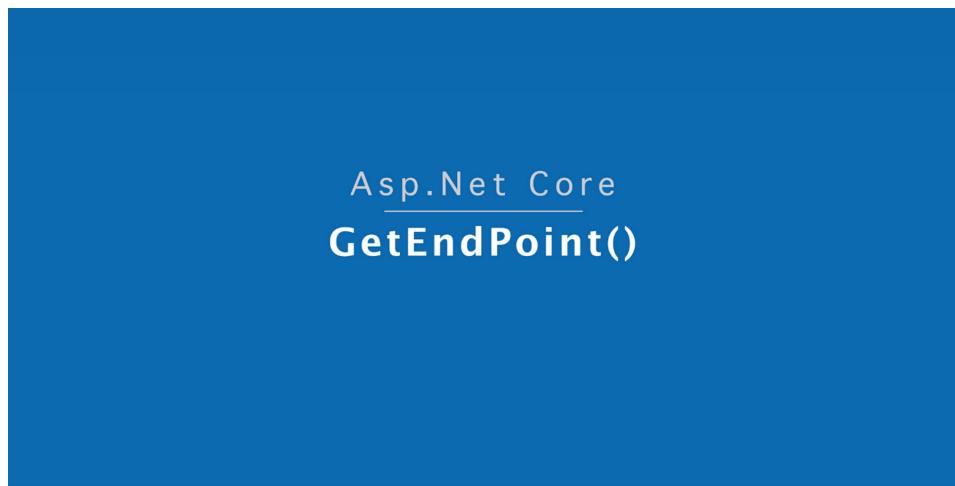


The screenshot shows the Visual Studio IDE with the 'Programs' window open. The code editor displays the following C# code:

```
1 var builder = WebApplication.CreateBuilder(args);
2 var app = builder.Build();
3
4 //enable routing
5 app.UseRouting();
6
7
8 //creating end points
9 app.UseEndpoints(endpoints =>
10 {
11     //add your end points
12     endpoints.Map("map1", async (context) =>
13     {
14         await context.Response.WriteAsync("In Map 1");
15     });
16
17     endpoints.Map("map2", async (context) =>
18     {
19         await context.Response.WriteAsync("In Map 2");
20     });
21 });
22
23 app.Run();
24
```



The screenshot shows the Postman application interface. At the top, there are tabs for Home, Workspaces, Reports, and Explore. A search bar says "Search Postman". On the right, there are buttons for Sign In and Create Account. Below the header, there's a list of requests. The first request is a POST to "http://localhost:5010/map1". Underneath it, there's a table for "Query Params" with one row: "Key" (Key) and "Value" (Value). At the bottom, there are tabs for Body, Cookies, Headers (4), Test Results, and a status bar showing Status: 405 Method Not Allowed, Time: 20 ms, Size: 120 B, and Save Response. A "Send" button is at the top right of the main request area.



When the "use routing" method executes, it identifies the appropriate endpoint based on the incoming request. By the time of executing the "use routing" method at runtime, the compiled code already has enough information about the endpoints.

That means it already knows, for which URL, which endpoint should be executed in the compiled code. So, exactly when you call the "use routing" in your application request pipeline, it identifies:

- What is the incoming request URL?
- What are the list of endpoints available in the code?

It will try to match the incoming URL with all the endpoints available in the code and pick the appropriate one based on the matching URL and HTTP method.

For example, if the user makes a "GET" request with the URL "map1", the corresponding first endpoint will be picked up. It stores that information of the endpoint in the form of an endpoint object, which you can get programmatically by using "get endpoint".

You can get the endpoint object only after "use routing" executes because, before calling the "use routing", the appropriate endpoint was not recognized by ASP.NET Core.

This is the reason why, if you call the "get endpoint" method before "use routing", it returns null. You can get the appropriate endpoint after calling "use routing".

In that case, it returns the corresponding endpoint object, which is of the 'Microsoft.AspNetCore.Http.Endpoint' class and contains two important properties:  
- \*\*"Display Name"\*\*: Probably the same as your URL.  
- \*\*"Request Delegate"\*\*: The actual endpoint that should be executed.

## Asp.Net Core Route Parameters

```
/files/sample.txt  
  
/employee/profile/john
```

So this particular segment; means this particular portion is literal. Means fixed value. But the other value can vary, right! It can be any file name and any file extension. So whichever parts of the URL can vary; those are called route parameters.

```
/files/{filename}.{extension}
```

```
/employee/profile/{employeename}
```

For example, you can consider this as a route parameter; and you can represent the same with curly braces; that means a parameter; then you can provide any parameter name. Just like a method parameter, a route parameter can receive a value. For example, I am going to name it as "filename". So this parameter name can be anything. That means the user can supply any value to this particular parameter. I may supply "sample"; you may supply some other file name, whatever you like. It can be anything. Similarly, the file extension also can be anything, right! So you can represent that part as another parameter. Here "." is the literal text. So whichever name is outside the curly braces is called literal text; whichever name is inside the curly braces is called a parameter. That means it is ready to accept any value.

Now, in this second URL, guess what are the parameters and what is the literal text. "Employee"; this is the fixed value, right! So it is the literal text. Now see the second segment. "Profile". It is also fixed. So leave it. It is the literal text.

# Asp.Net Core

## Default Route Parameters

Asp.Net Core

| Harsha

### Default Route Parameter

#### Route Parameter with default value

```
"{parameter=default_value}"
```

A route parameter with default value matches with any value. It also matches with empty value. In this case, the default value will be considered into the parameter.



# Asp.Net Core

## Optional Route Parameters

## Optional Route Parameter

### Optional Route Parameter

"{parameter?}"

"?" indicates an optional parameter.

That means, it matches with a value or empty value also.



```

4     //enable routing
5     app.UseRouting();
6
7     //creating end points
8     app.UseEndpoints(endpoints =>
9     {
10        endpoints.Map("files/{filename}.{extension}", async (context) =>
11        {
12            string? fileName = Convert.ToString(context.Request.RouteValues["filename"]);
13            string? extension = Convert.ToString(context.Request.RouteValues["extension"]);
14            await context.Response.WriteAsync($"In files - {fileName} - {extension}");
15        });
16
17        //route parameter name are case insensitive
18        endpoints.Map("employee/profile/{employeeName=khalid}", async (context) =>
19        {
20            string? employeeName = Convert.ToString(context.Request.RouteValues["employeeName"]); // by default it returns object type. We have converted into string type
21            await context.Response.WriteAsync($"In Employee profile - {employeeName}");
22        });
23
24        //Eg: products/details/1
25        //if the user does not supply then id=null as it is optional parameter
26        endpoints.Map("products/details/{id?", async context =>
27        {
28            int id = Convert.ToInt32(context.Request.RouteValues["id"]);
29            await context.Response.WriteAsync($"Product details - {id}");
30        });
31    });
32

```

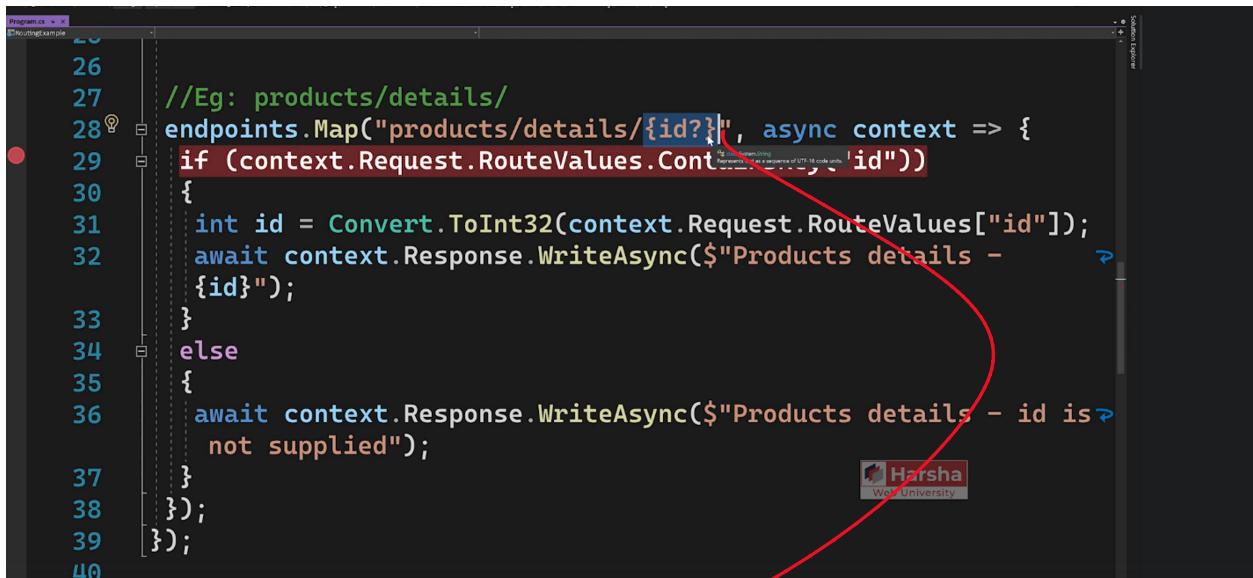


because of conversion

# Asp.Net Core

## Route Constraints

### Part 1



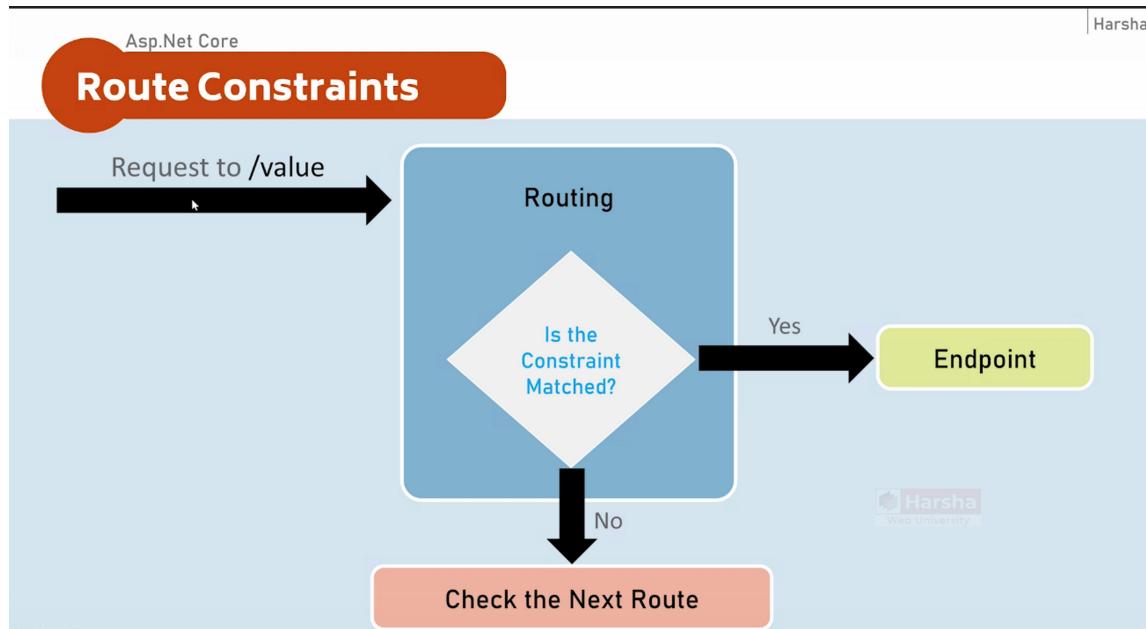
```
program.cs //efcorestexample
25
26
27     //Eg: products/details/
28     endpoints.Map("products/details/{id?}", async context => {
29         if (context.Request.RouteValues.ContainsKey("id"))
30         {
31             int id = Convert.ToInt32(context.Request.RouteValues["id"]);
32             await context.Response.WriteAsync($"Products details - {id}");
33         }
34         else
35         {
36             await context.Response.WriteAsync($"Products details - id is not supplied");
37         }
38     });
39 });
40 }
```

By default, a parameter accepts any value into that. For example, if you supply an alphabetical value, numerical value, or even an alphanumerical value, a date value, or a boolean value, by default, all types of values are accepted by this parameter. But generally, you would like to restrict the type of values, right!

For example, I want to accept only integer values in this ID parameter. In some other situation, maybe, for example, in this employee profile example, you would like to say some other condition, saying that the employee name should be alphabetical.

So, to specify restrictions on the parameters, there is a concept called constraints. Basically, when an incoming

request is



Basically, when an incoming request is received by the ASP.NET Core application and has reached the 'UseRouting' method, it will verify whether the parameter value in the URL matches the constraint.

The constraint is stored as part of the endpoint and is verified here. A constraint is nothing but a condition or restriction, specifying that the parameter value is expected to meet certain criteria.

If the constraint matches the incoming URL, the endpoint executes as usual. Otherwise, it looks for the next available route. If none of the routes match, it executes the final 'app.Run' statement present at the end of the application request pipeline.

## Route Constraints

### int

Matches with any integer.

Eg: {id:int} matches with 123456789, -123456789

### bool

Matches with true or false. Case-insensitive.

Eg: {active:bool} matches with true, false, TRUE, FALSE

### datetime

Matches a valid DateTime value with formats "yyyy-MM-dd hh:mm:ss tt" and "MM/dd/yyyy hh:mm:ss tt".

Eg: {id:datetime} matches with 2030-01-01%2011:59%20pm Note: '%20' is equal to space.



Intro

| int

| datetime

| udemy

Asp.Net Core

## Route Constraints

Part 2

## Route Constraints

### decimal

Matches with a valid decimal value.

Eg: {price:decimal} matches with 49.99, -1, 0.01

### long

Matches a valid long value.

Eg: {id:long} matches with 123456789, -123456789

### guid



Matches with a valid Guid value (Globally Unique Identifier - A hexadecimal number that is universally unique).

Eg: {id:guid} matches with 123E4567-E89B-12D3-A456-426652340000

```

22    });
23
24    // Eg: products/details/{id:int}
25    endpoints.Map("products/{id}")
26    {
27        if (context.Request.RouteValues.ContainsKey("id"))
28        {
29            int id = Convert.ToInt32(context.Request.RouteValues["id"]);
30            await context.Response.WriteAsync($"Product details - {id}");
31        }
32        else
33        {
34            await context.Response.WriteAsync("Product details - id is not supplied");
35        }
36    };
37
38    endpoints.Map("daily-digest-report/{reportdate:datetime}", async context =>
39    {
40        DateTime reportDate = Convert.ToDateTime(context.Request.RouteValues["reportdate"]);
41        await context.Response.WriteAsync($"In daily-digest-report - {reportDate.ToShortDateString()}");
42    });
43
44    // Eg: cities/{cityid}
45    endpoints.Map("cities/{cityid:guid}", async context =>
46    {
47        Guid cityId = Guid.Parse(Convert.ToString(context.Request.RouteValues["cityid"]));
48        await context.Response.WriteAsync($"City information - {cityId}");
49    });
50
51

```

```

22    });
23
24    // Eg: products/details/{id:int?}, async context =>
25    endpoints.Map("products/{id:int?}")
26    {
27        if (context.Request.RouteValues.ContainsKey("id"))
28        {
29            int id = Convert.ToInt32(context.Request.RouteValues["id"]);
30            await context.Response.WriteAsync($"Product details - {id}");
31        }
32        else
33        {
34            await context.Response.WriteAsync("Product details - id is not supplied");
35        }
36    };
37
38    endpoints.Map("daily-digest-report/{reportdate:datetime}", async context =>
39    {
40        DateTime reportDate = Convert.ToDateTime(context.Request.RouteValues["reportdate"]);
41        await context.Response.WriteAsync($"In daily-digest-report - {reportDate.ToShortDateString()}");
42    });
43
44    // Eg: cities/{cityid}
45    endpoints.Map("cities/{cityid:guid}", async context =>
46    {
47        Guid cityId = Guid.Parse(Convert.ToString(context.Request.RouteValues["cityid"]));
48        await context.Response.WriteAsync($"City information - {cityId}");
49    });
50
51

```

# Asp.Net Core

## Route Constraints

### Part 3

Asp.Net Core

Harsha

## Route Constraints

### minlength(value)

Matches with a string that has at least specified number of characters.

Eg: {username:minlength(4)} matches with John, Allen, William

### maxlength(value)

Matches with a string that has less than or equal to the specified number of characters.

Eg: {username:maxlength(7)} matches with John, Allen, William

### length(min,max)

Matches with a string that has number of characters between given minimum and maximum length (both numbers including).

Eg: {username:length(4, 7)} matches with John, Allen, William



length

| "min" and "max" | alpha | regex

| Parameter Validation |

Udemy

## Route Constraints

### range(min,max)

Matches with an integer value between the specified minimum and maximum values (both numbers including).

Eg: {age:range(18,100)} matches with 18, 19, 99, 100

### alpha

Matches with a string that contains only alphabets (A-Z) and (a-z).

Eg: {username:alpha} matches with rick, william

## Route Constraints

### regex(expression)

Matches with a string that matches with the specified regular expression.

Eg 1: {age:regex(^[0-9]{2}\$)} matches with any two-digit number, such as 10, 11, 98, 99

Eg 2: {age:regex(^\\d{3}-\\d{3}\$)} matches with any three-digit number, then hyphen, and then three-digit number, such as 123-456

But in general, as per Microsoft Docs, you should not use route constraints to validate values.

The better approach is to accept invalid values into the route and then validate them in your code using "if" statements. If the value is found to be invalid, you can provide an appropriate response.

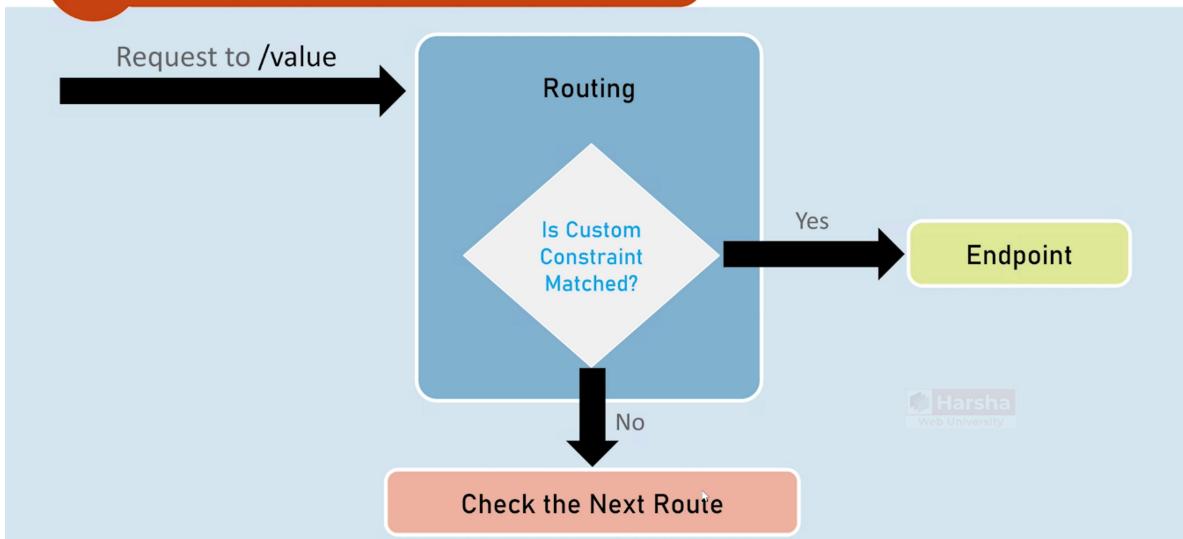
For example, you can return a status code 400 (Bad Request), which informs the client about the expected or accepted values for this route. This approach provides more clarity and flexibility in handling invalid inputs.

```
56     i
57
58     int year = Convert.ToInt32(context.Request.RouteValues
59     ["year"]);
59     string? month = Convert.ToString(context.Request.RouteValues
60     ["month"]);
60     if (month == "apr" || month == "jul" || month == "oct" ||
61     month == "jan")
61     {
62         await context.Response.WriteAsync($"sales report - {year} -
62         {month}");
63     }
64     else
65     {
66         await context.Response.WriteAsync($"{month} is not allowed
66         for sales report");
67     }
68 }
```

## Asp.Net Core Custom Route Constraints

```
53 //sales-report/2030/apr
54 endpoints.Map("sales-report/{year:int:min(1900)}/{month:regex
55 (^apr|jul|oct|jan)$}", async context =>
56 {
57
58     int year = Convert.ToInt32(context.Request.RouteValues
58     ["year"]);
59     string? month = Convert.ToString(context.Request.RouteValues
59     ["month"]);
60
61     if (month == "apr" || month == "jul" || month == "oct" ||
61     month == "jan")
62     {
63         await context.Response.WriteAsync($"sales report - {year} -
63         {month}");
64     }
65 }
```

## Custom Route Constraints



Suppose a particular regular expression constraint needs to be applied in multiple places across your application. Instead of manually repeating the same regular expression everywhere, you can convert it into a custom constraint.

This means you create a class specifically for the constraint. A custom constraint works like a normal constraint, but instead of being a simple condition, it is encapsulated in a class.

Upon receiving a request, while verifying the route, the custom constraint is checked. The custom constraint class executes a method called `Match` to verify whether the incoming request value matches the constraint.

If it matches, the endpoint executes; otherwise, it moves to the next route.

The custom constraint class implements an interface called `IRouteConstraint`. This interface enforces the implementation of the `Match` method, which verifies whether the incoming request matches the constraint.

```

public class ClassName : IRouteConstraint
{
    public bool Match(HttpContext? HttpContext, IRouter? route,
        string routeKey, RouteValueDictionary values, RouteDirection routeDirection)
    {
        //return true or false
    }
}

builder.Services.AddRouting(options =>
{
    options.ConstraintMap.Add("name", typeof(ClassName));
}); //adding the custom constraint to routing

```

The code block shows a C# class named `ClassName` that implements the `IRouteConstraint` interface. It contains a single `Match` method. Below the class definition, a configuration block registers this constraint with the name "name".

# Asp.Net Core

## Endpoint Selection Order

Asp.Net Core | Harsha

### Endpoint Selection Order

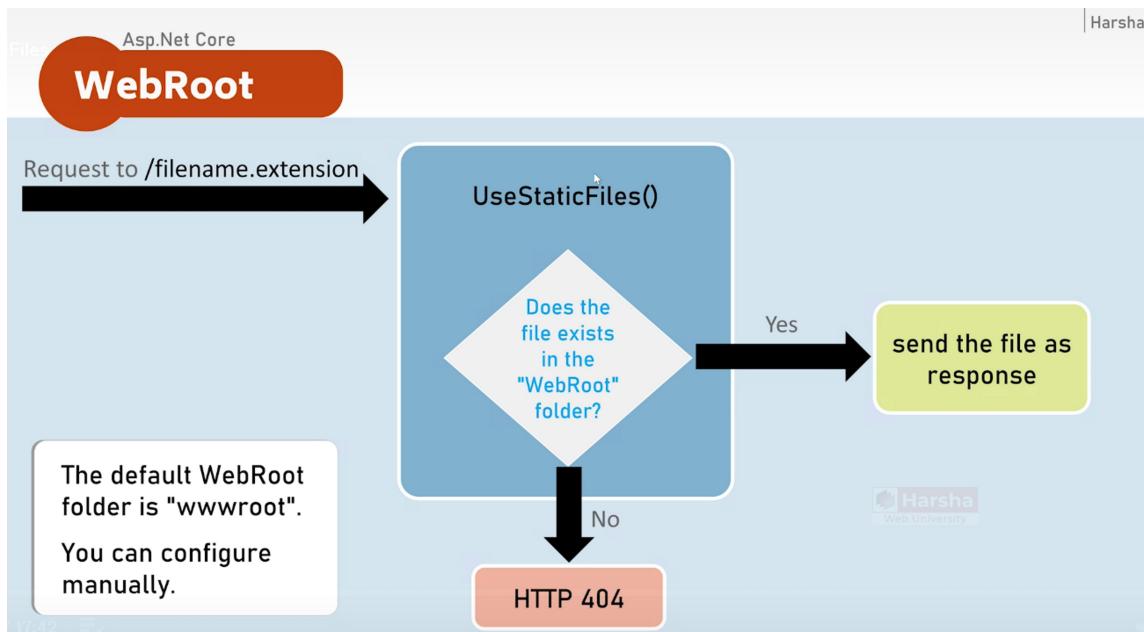
Top is highest precedence (will be evaluated first)

- 1 URL template with more segments.  
Ex: "a/b/c/d" is higher than "a/b/c".
- 2 URL template with literal text has more precedence than a parameter segment.  
Ex: "a/b" is higher than "a/{parameter}".
- 3 URL template that has a parameter segment with constraints has more precedence than a parameter segment without constraints.  
Ex: "a/b:int" is higher than "a/b".
- 4 Catch-all parameters (\*\*).  
Ex: "a/b" is higher than "a/\*\*".

Intro | EndPoint Selection Order in Practice | Conclusion | Q&A

# Asp.Net Core

## WebRoot and UseStaticFiles()



An ASP.NET Core web application may contain static files or content such as images, text files, PDFs, JavaScript files, CSS files, etc. Generally, while working with views, you need to use these types of files.

By default, these files cannot be served by ASP.NET Core. You need to enable it using the '`UseStaticFiles`' middleware, which allows the application to serve static files when the URL contains the file path.

For example, if a request is sent for '`sample.jpg`' and the file exists in the application, it will be served as a response to the browser, enabling the browser to display the file's content.

To serve static files like images or text files, you need to enable '`UseStaticFiles`'.

By default, ASP.NET Core recommends placing all static files, such as images and JavaScript files, in the web root folder, commonly named '`wwwroot`'. While this is the default, it is not fixed. You can configure the name or even enable multiple '`wwwroot`' folders.

```
File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Search WebRoot_and_UseStaticFiles
Program.cs X
/WebRoot_and_UseStaticFiles
1 var builder = WebApplication.CreateBuilder(args);
2 var app = builder.Build();
3 //in order to enable wwwroot file
4 app.UseStaticFiles(); //pre-defined middleware
5 //app.MapGet("/", () => "Hello World!"); //this is the shortcut way to create middleware
6 //But this is the preferred way
7 app.UseRouting();
8
9 app.UseEndpoints(endpoints =>
10 {
11     endpoints.Map("/", async context =>
12     {
13         await context.Response.WriteAsync("Hello");
14     });
15 });
16
17 app.Run();
18
19
20
21
```

