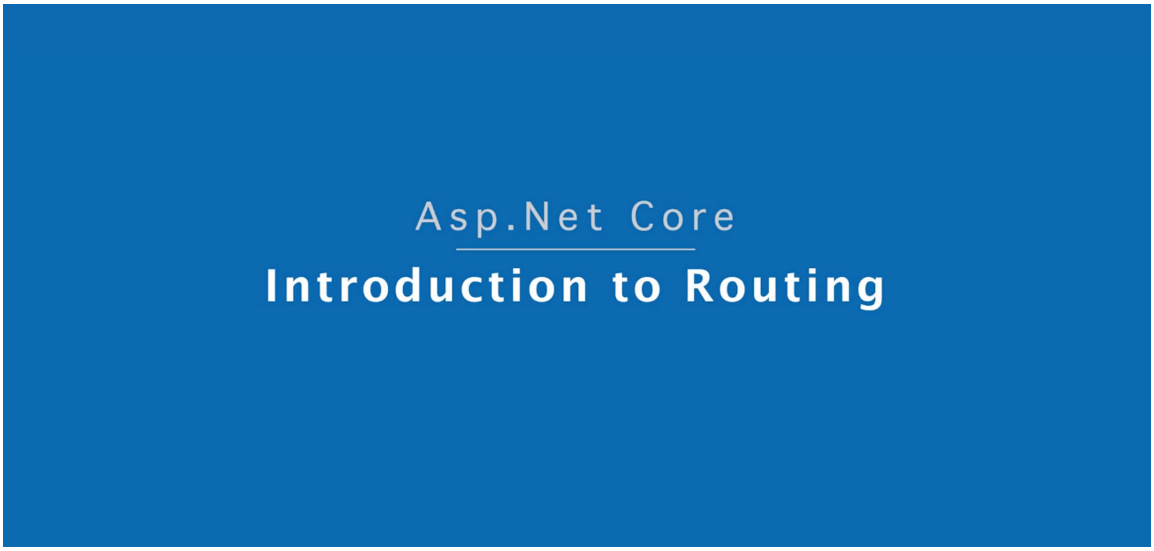
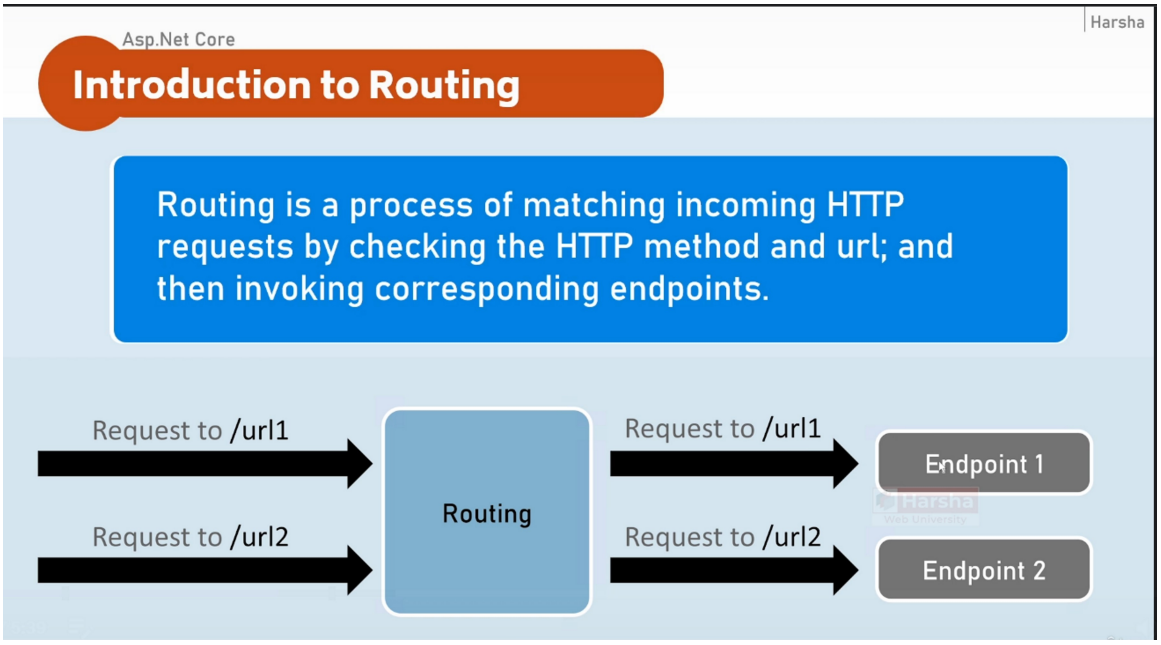
Section 5: Routing [MVC and Web API]

30 December 2024 23:04

1. Intro to Routing





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:

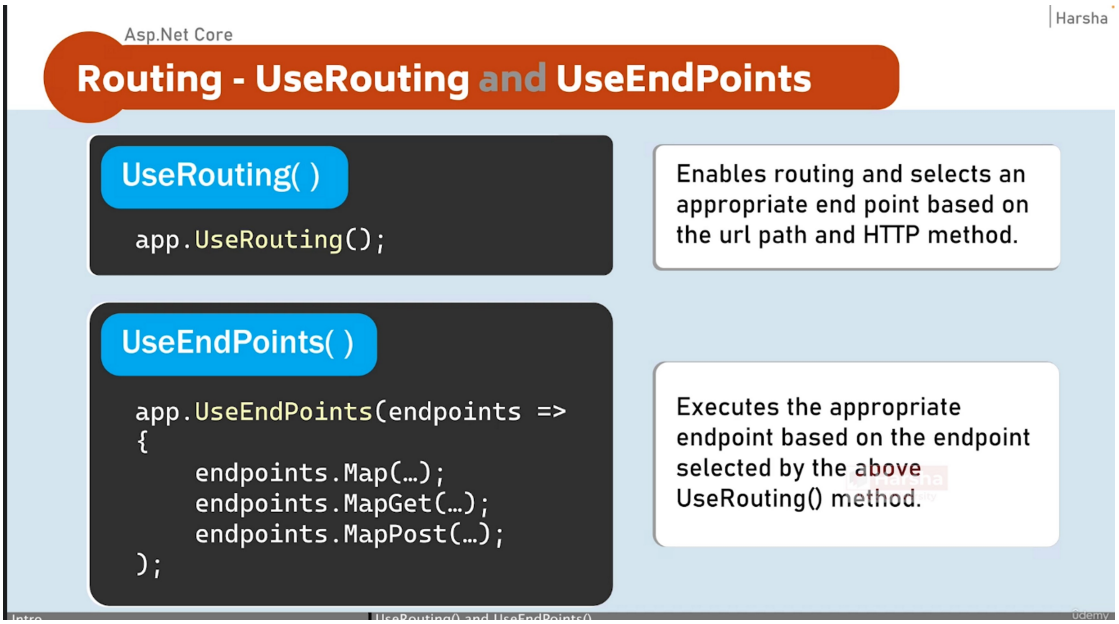
- A user requests "/home" → The Home Page endpoint is triggered.
- A user requests "/about" → The About Page endpoint is triggered.

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 ensures organized request handling and is a fundamental feature in ASP.NET Core applications.

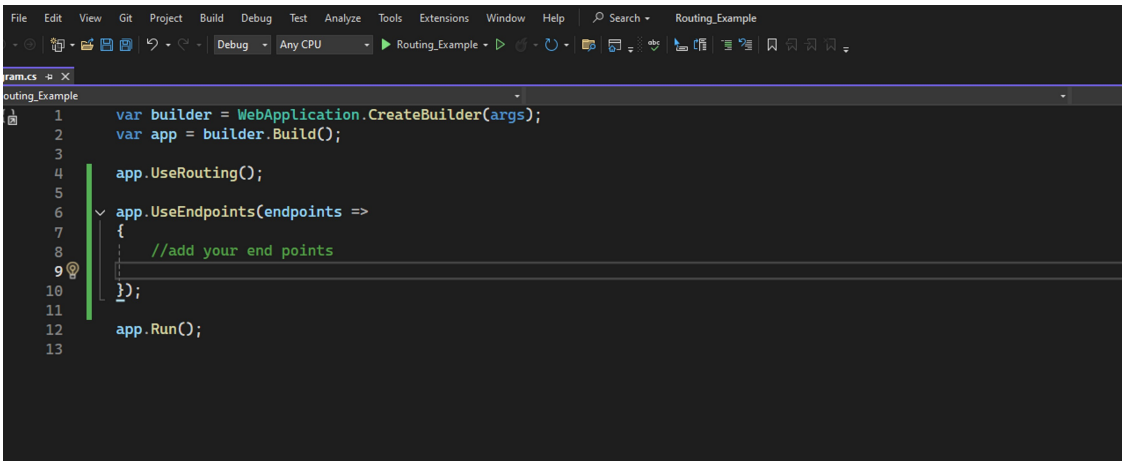


With two individual methods, that is "use routing" and "use endpoints".
You would like to invoke them in the same order, that is first "use routing" and after that only "use endpoints".

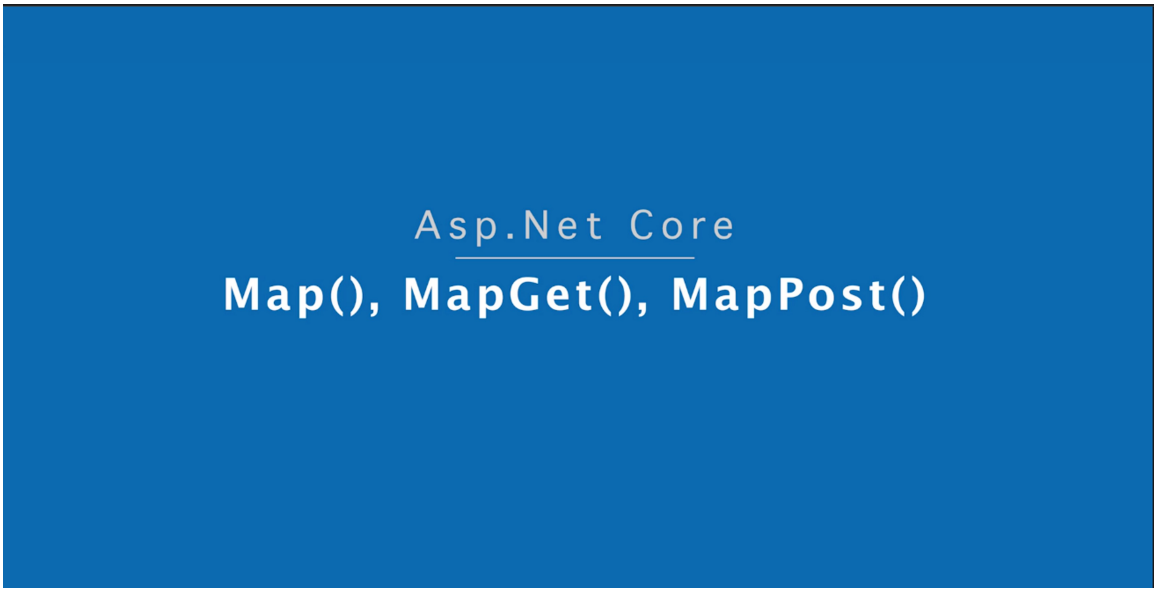
So, what's the difference between them?

"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".



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

Harsha

Routing - UseRouting and UseEndPoints

UseRouting( )

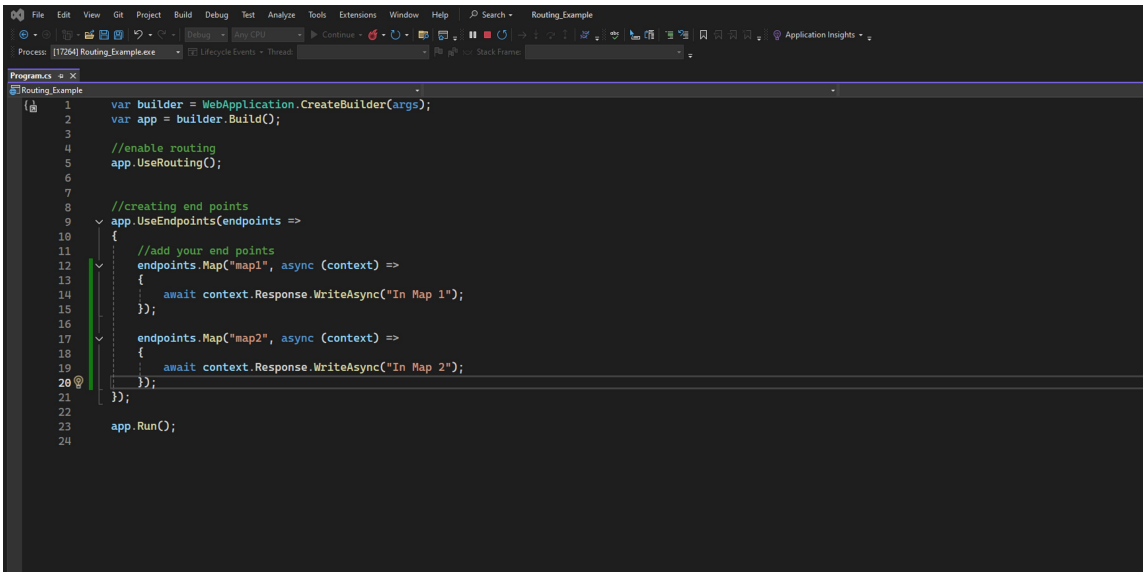
```
app.UseRouting();
```

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

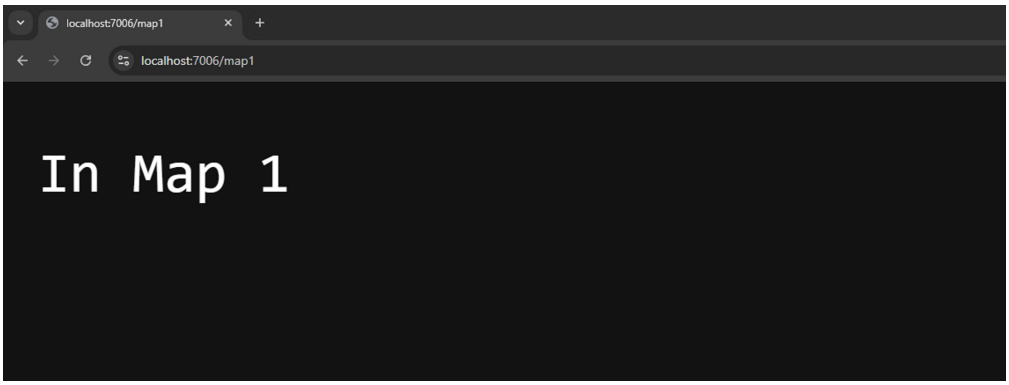
UseEndPoints( )

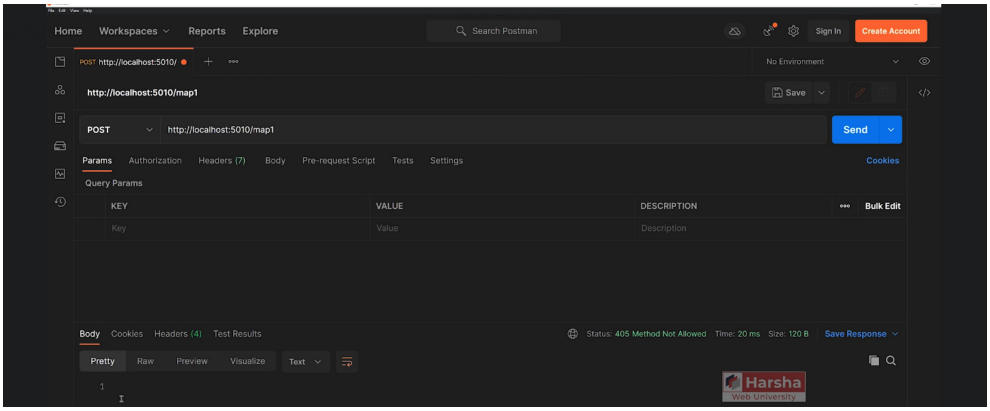
```
app.UseEndPoints(endpoints =>
{
    endpoints.Map(…);
    endpoints.MapGet(…);
    endpoints.MapPost(…);
);
```

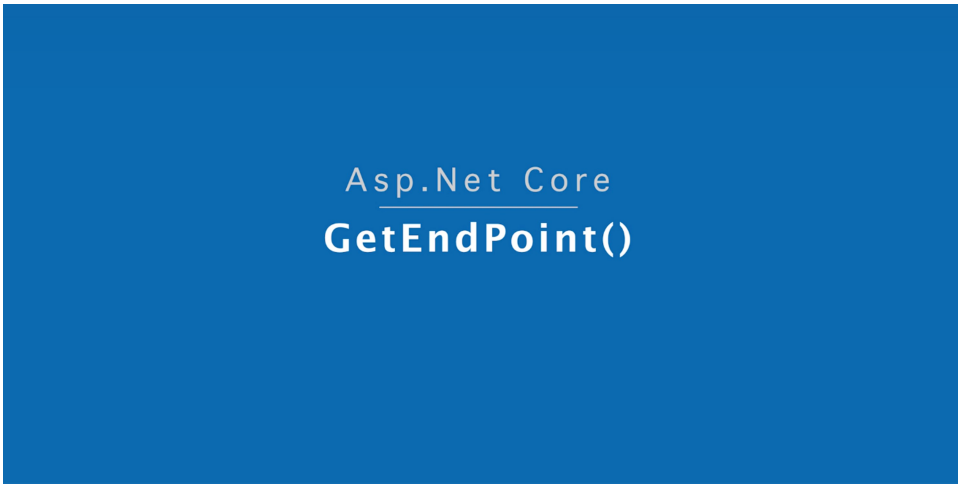
Executes the appropriate endpoint based on the endpoint selected by the above UseRouting() method.

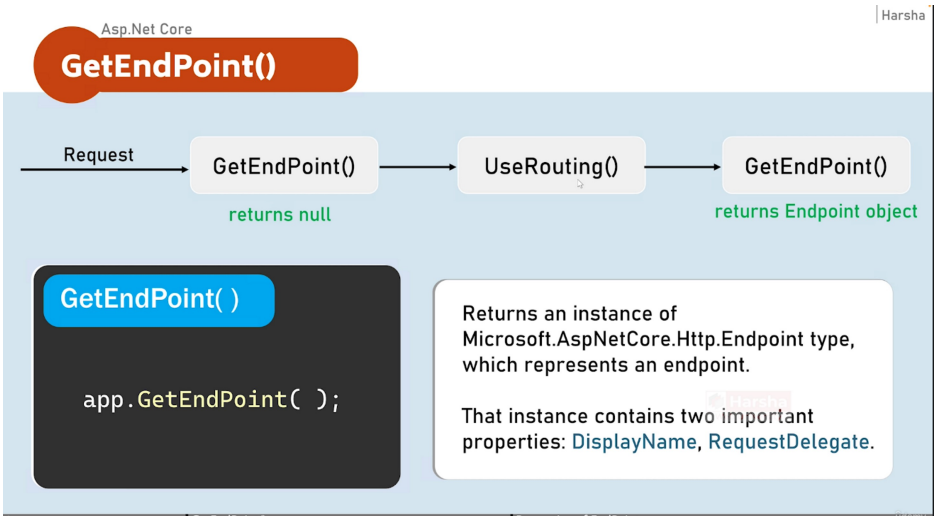


```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

//enable routing
app.UseRouting();


//creating end points
app.UseEndpoints(endpoints =>
{
    //add your end points
    endpoints.Map("map1", async (context) =>
    {
        await context.Response.WriteAsync("In Map 1");
    });

    endpoints.Map("map2", async (context) =>
    {
        await context.Response.WriteAsync("In Map 2");
    });
});

app.Run();
```









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.

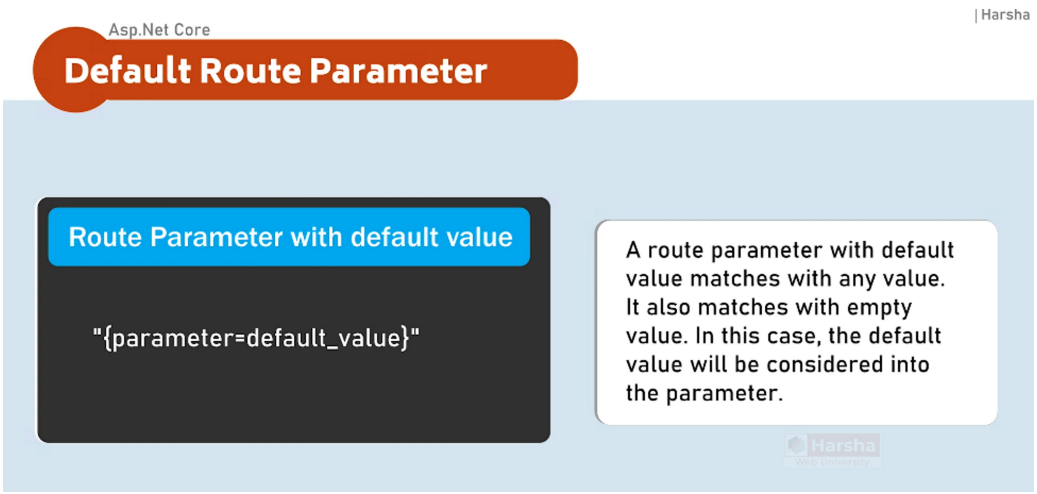
So this particular segment; means this particular portion is literal. Means fixed value. But the other value can vary, right! It can be any file name and any file extension. So whichever parts of the URL can vary; those are called route parameters.

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

Optional Route Parameters

Asp.Net Core

Optional Route Parameter

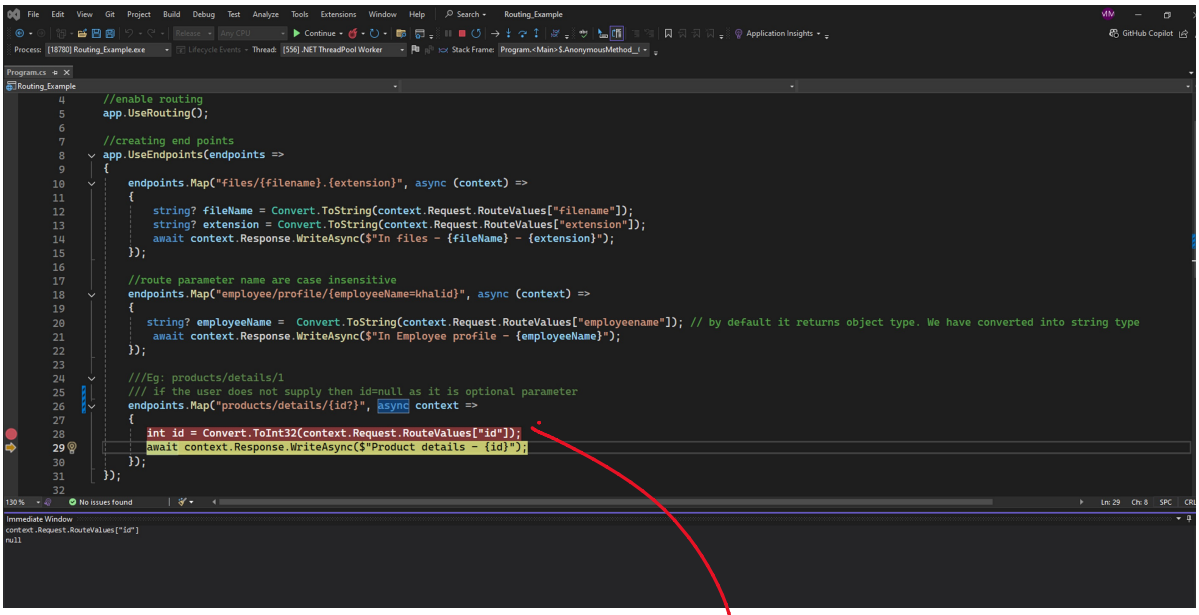
Optional Route Parameter

"{parameter?}"

"?" indicates an optional parameter.

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

Harsha Web University



```
//enable routing
app.UseRouting();

//creating end points
app.UseEndpoints(endpoints =>
{
    endpoints.Map("files/{filename}.{extension}", async (context) =>
    {
        string? fileName = Convert.ToString(context.Request.RouteValues["filename"]);
        string? extension = Convert.ToString(context.Request.RouteValues["extension"]);
        await context.Response.WriteAsync($"In files - {fileName} - {extension}");
    });

    //route parameter name are case insensitive
    endpoints.Map("employee/profile/{employeeName=khalid}", async (context) =>
    {
       string? employeeName =  Convert.ToString(context.Request.RouteValues["employeename"]); // by default it returns object type. We have converted into string type
        await context.Response.WriteAsync($"In Employee profile - {employeeName}");
    });

    ///Eg: products/details/1
    /// if the user does not supply then id=null as it is optional parameter
    endpoints.Map("products/details/{id?}", async context =>
    {
        int id = Convert.ToInt32(context.Request.RouteValues["id"]);
        await context.Response.WriteAsync($"Product details - {id}");
    });
});
```

```
Immediate Window
context.Request.RouteValues["id"]
null
```

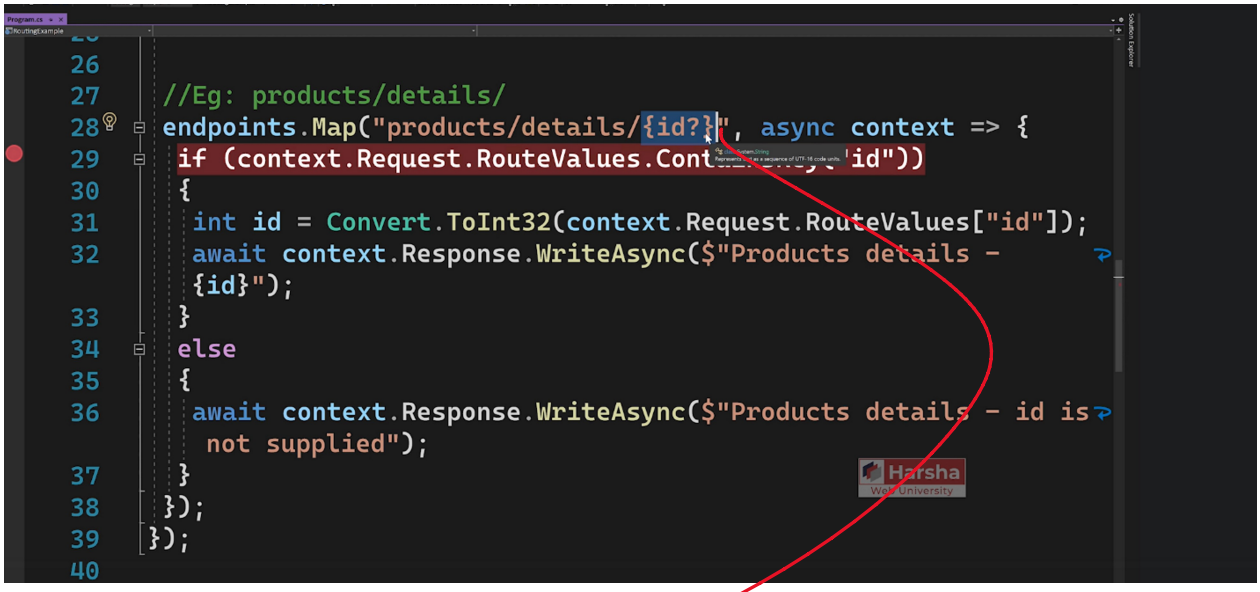
0

because of conversion

Asp.Net Core

Route Constraints

Part 1

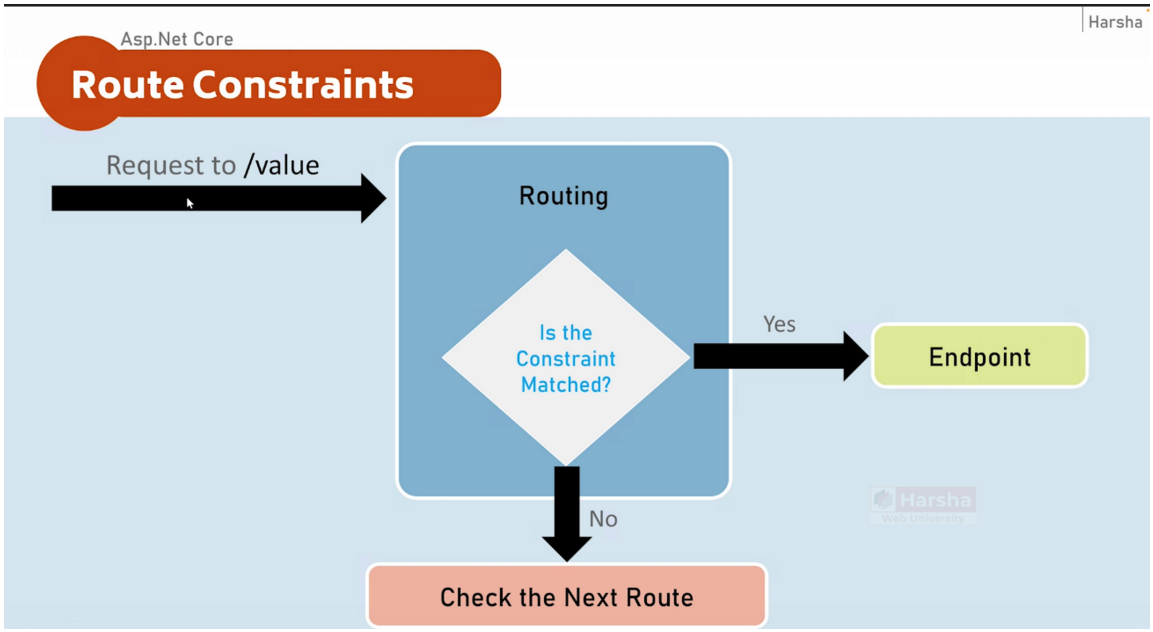


By default, a parameter accepts any value into that. For example, if you supply an alphabetical value, numerical value, or even an alphanumerical value, a date value, or a boolean value, by default, all types of values are accepted by this parameter. But generally, you would like to restrict the type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

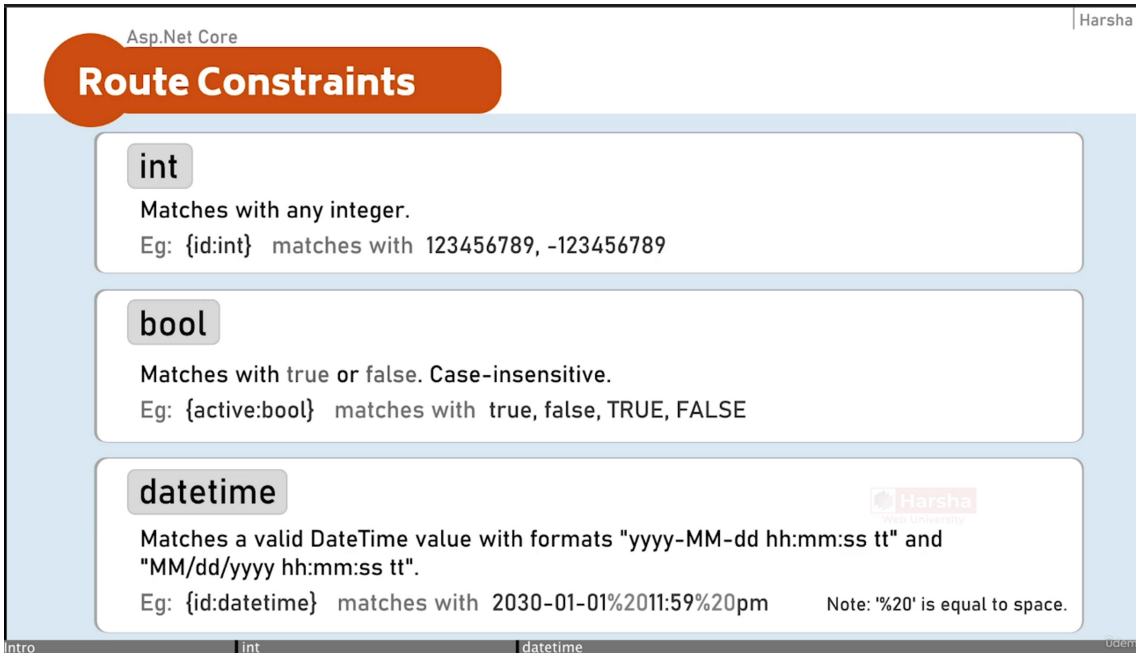
request is

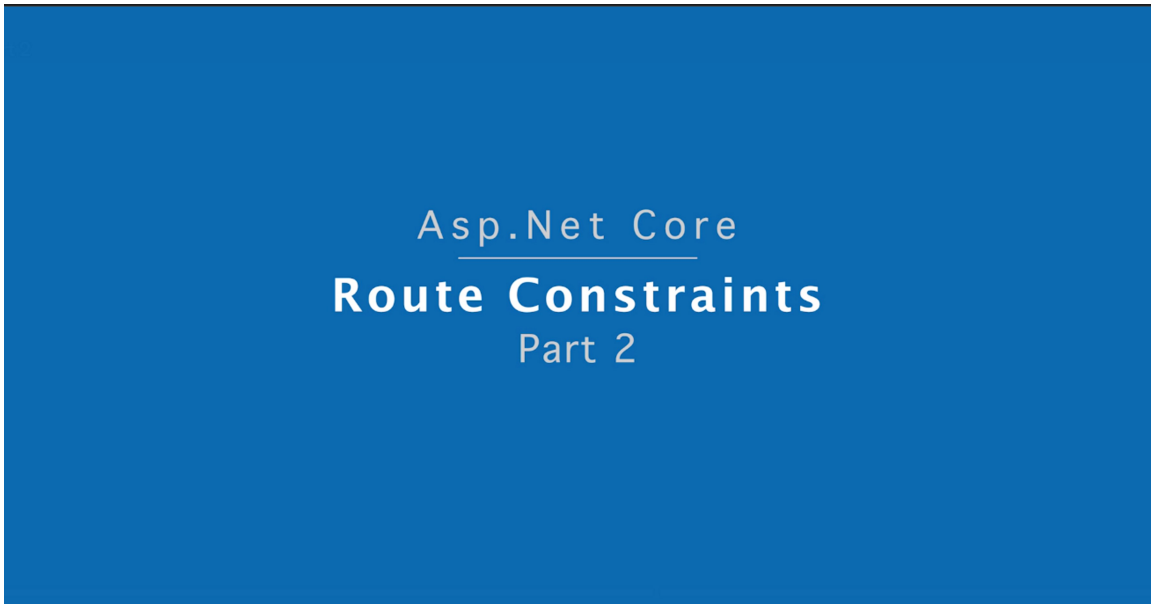


Basically, when an incoming request is received by the ASP.NET Core application and has reached the `UseRouting` method, it will verify whether the parameter value in the URL matches the constraint.

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.


Harsha
Asp.Net Core
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.
Harsha
Web University
Intro
int
datetime
Udemy


Asp.Net Core
Route Constraints
Part 2

Asp.Net Core

Route Constraints

decimal

Matches with a valid decimal value.

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

long

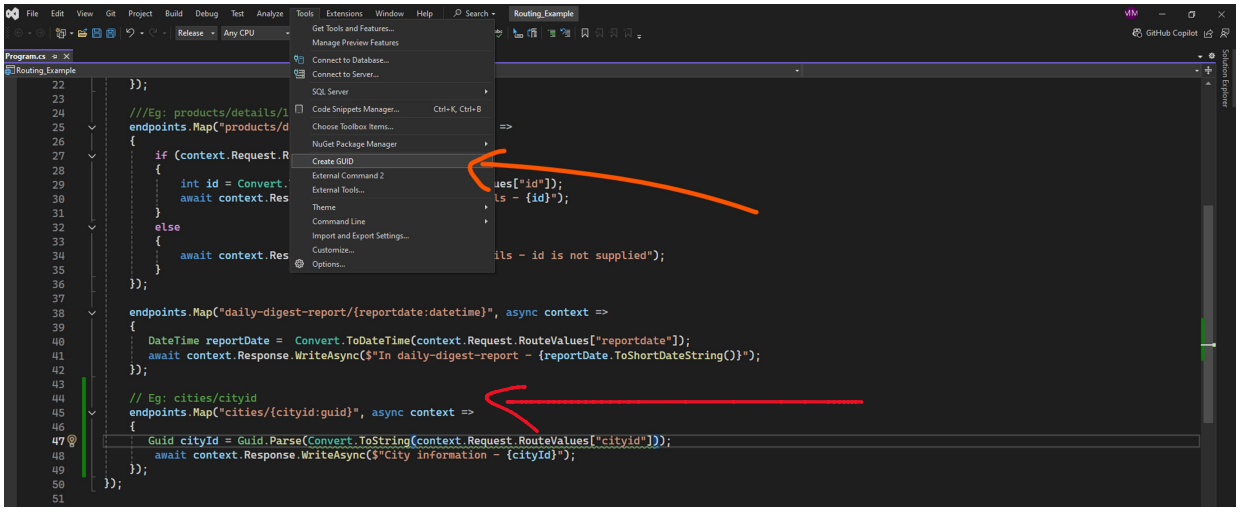
Matches a valid long value.

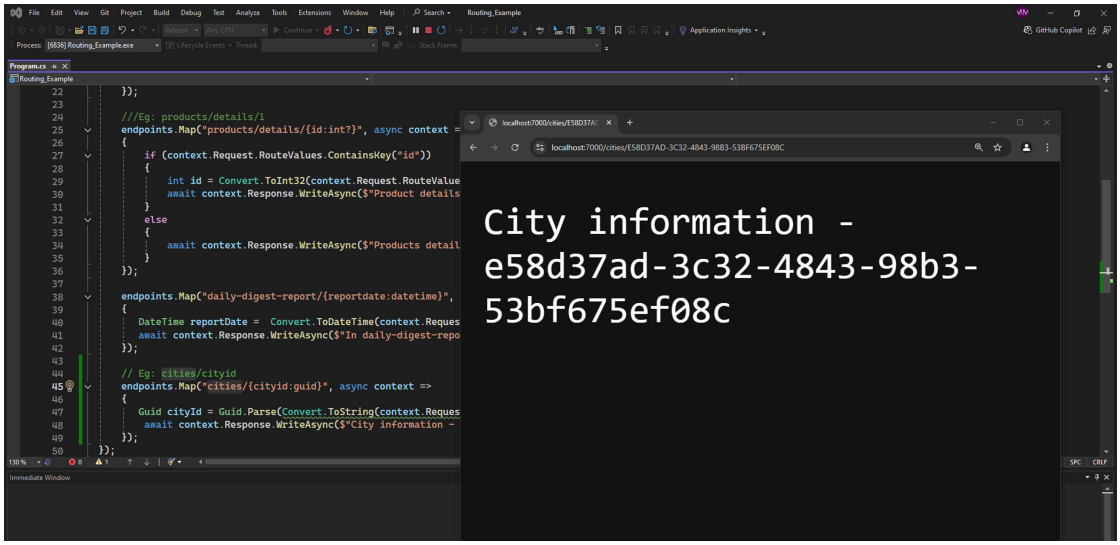
Eg: {id:long} matches with 123456789, -123456789

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

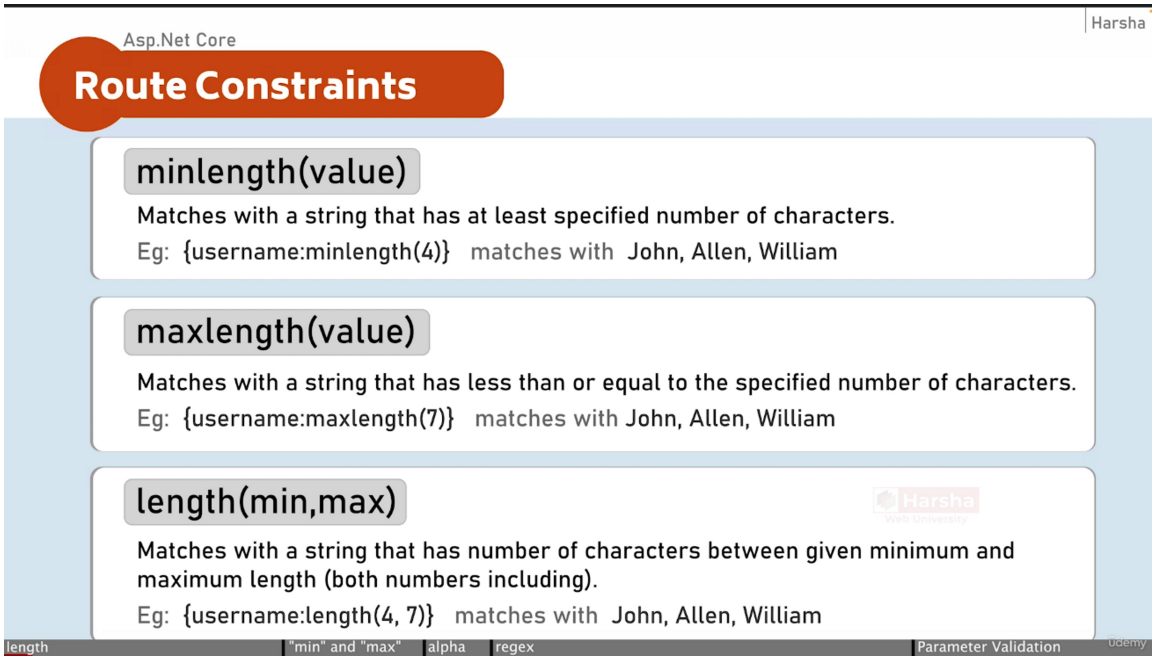


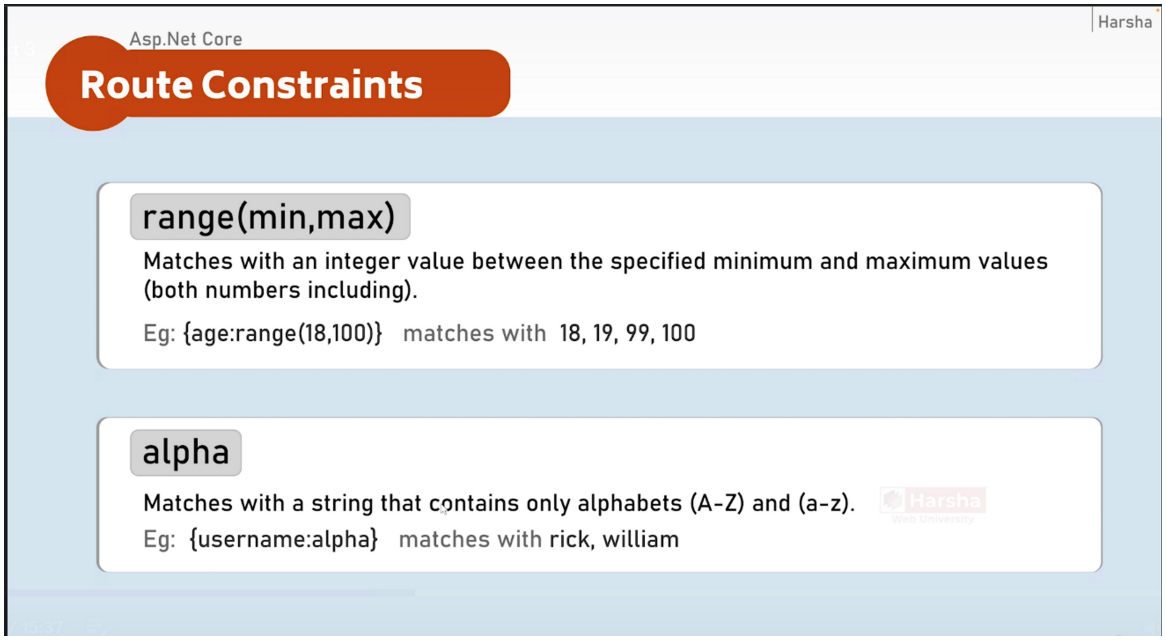


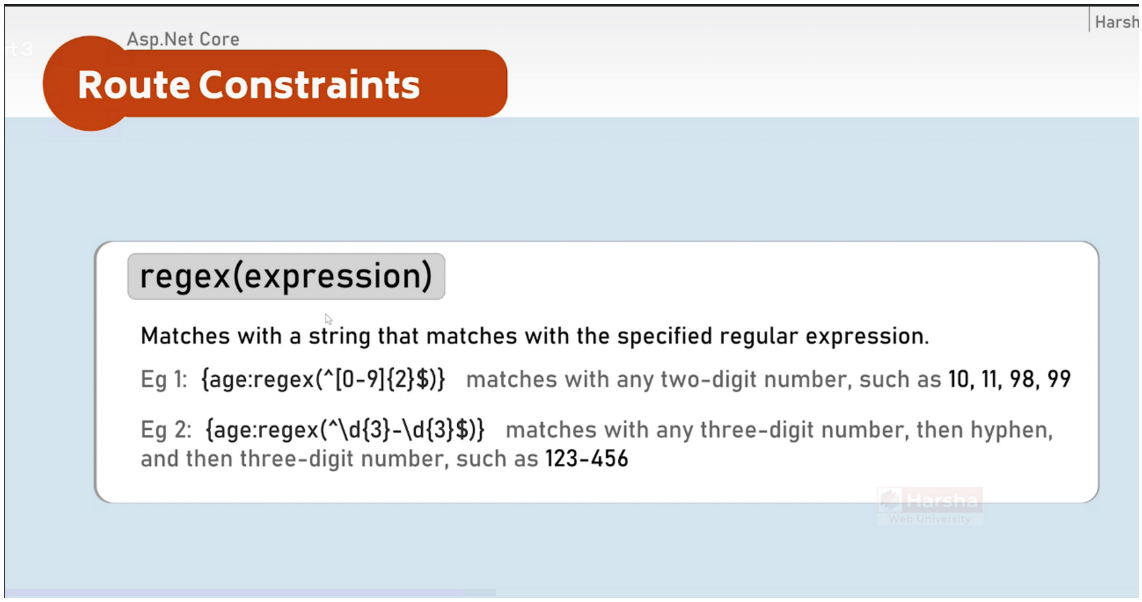
Asp.Net Core

Route Constraints

Part 3



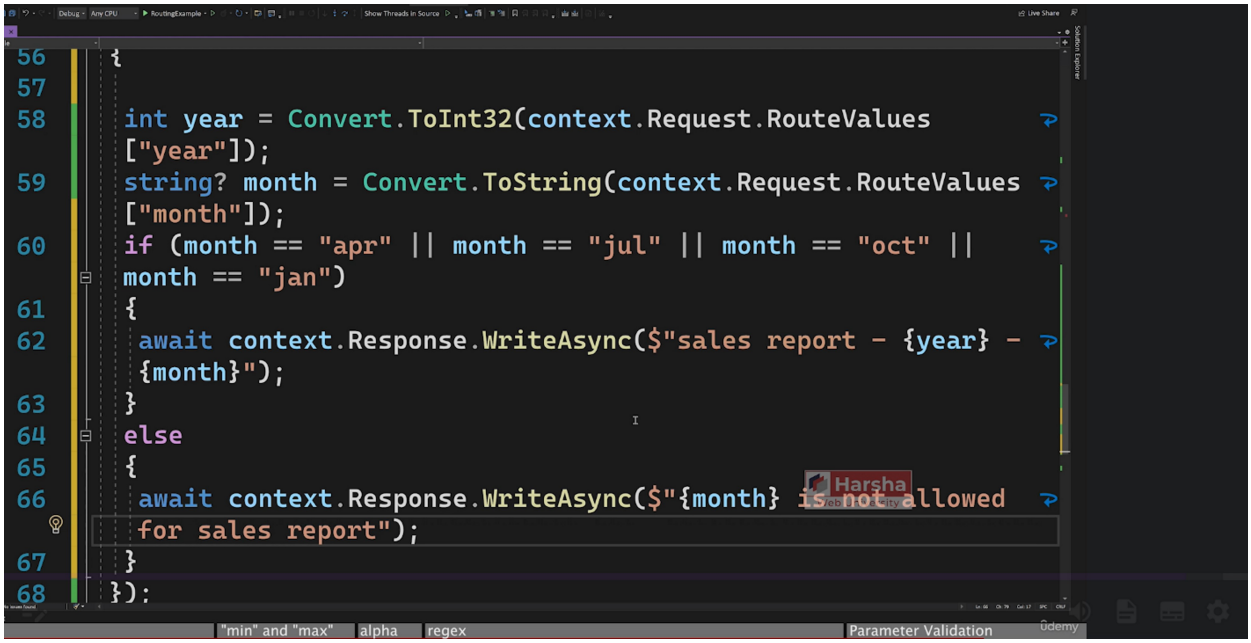




But in general, as per Microsoft Docs, you should not use route constraints to validate values.

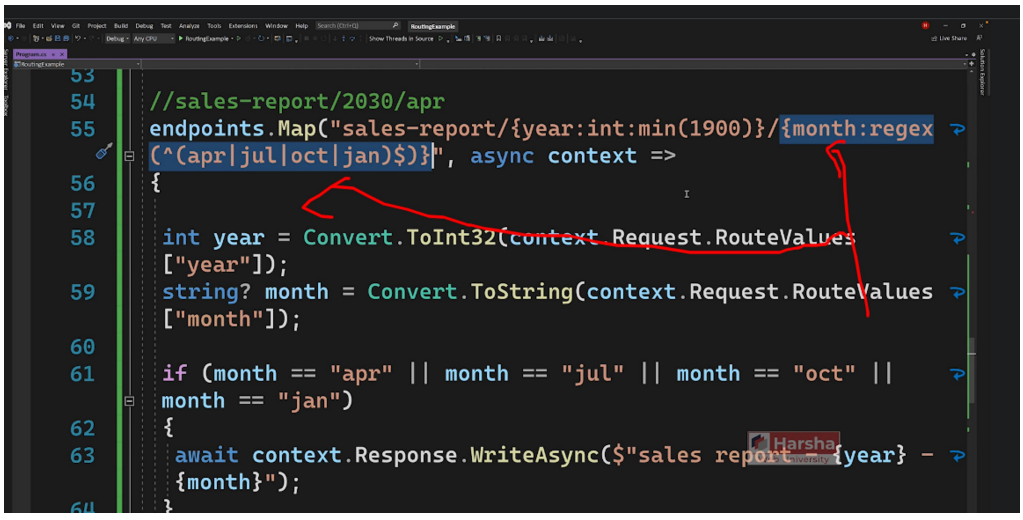
The better approach is to accept invalid values into the route and then validate them in your code using "if" statements. If the value is found to be invalid, you can provide an appropriate response.

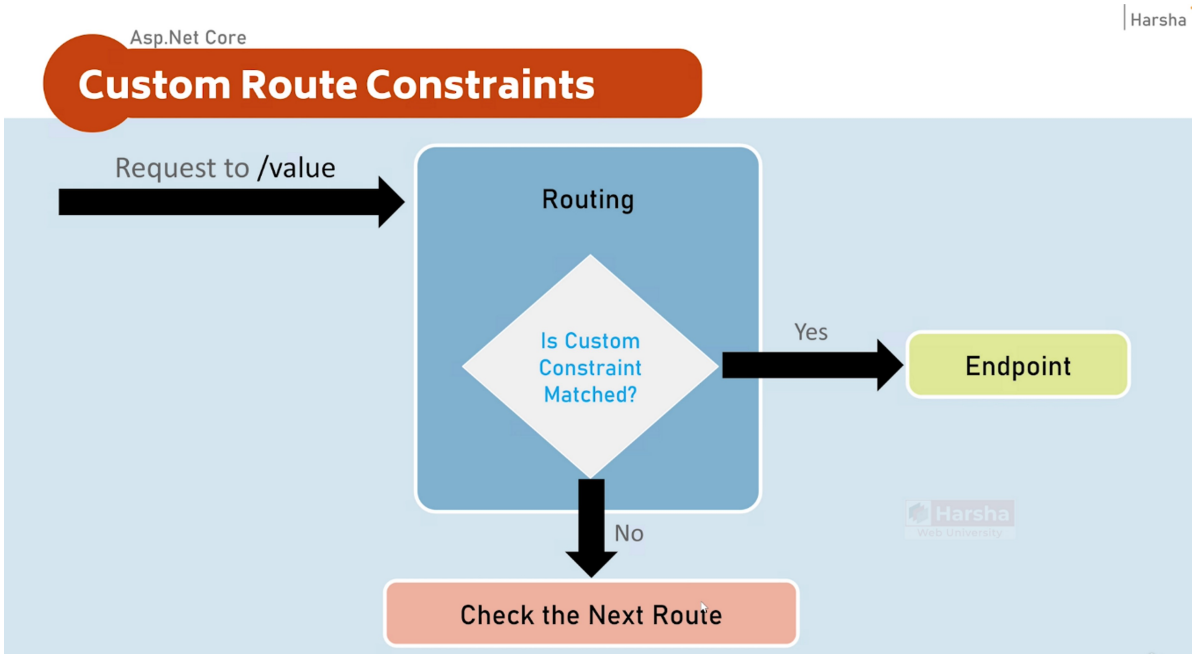
For example, you can return a status code 400 (Bad Request), which informs the client about the expected or accepted values for this route. This approach provides more clarity and flexibility in handling invalid inputs.



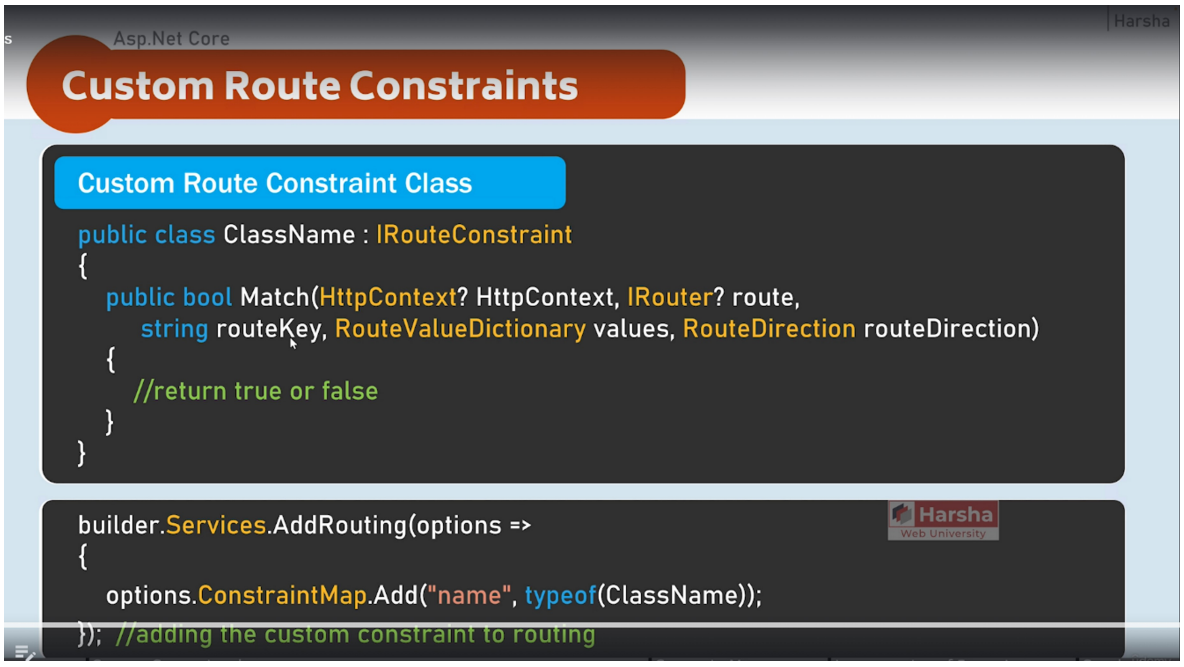
Asp.Net Core

Custom Route Constraints



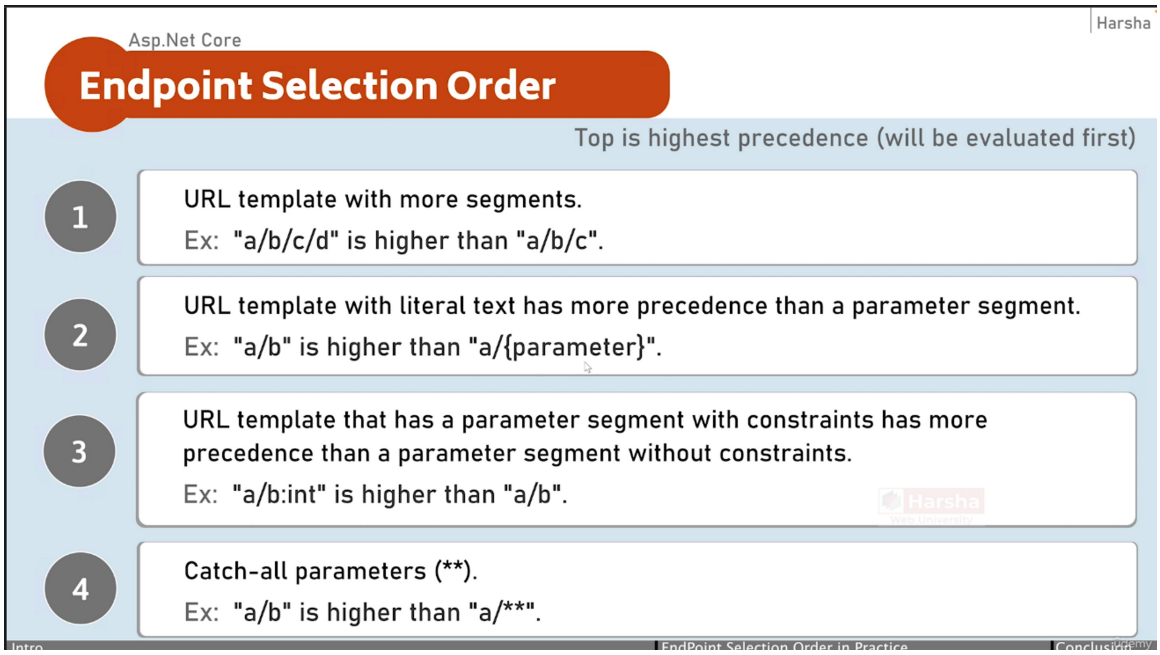


Suppose a particular regular expression constraint needs to be applied in multiple places across your application. Instead of manually repeating the same regular expression everywhere, you can convert it into a custom constraint.
This means you create a class specifically for the constraint. A custom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



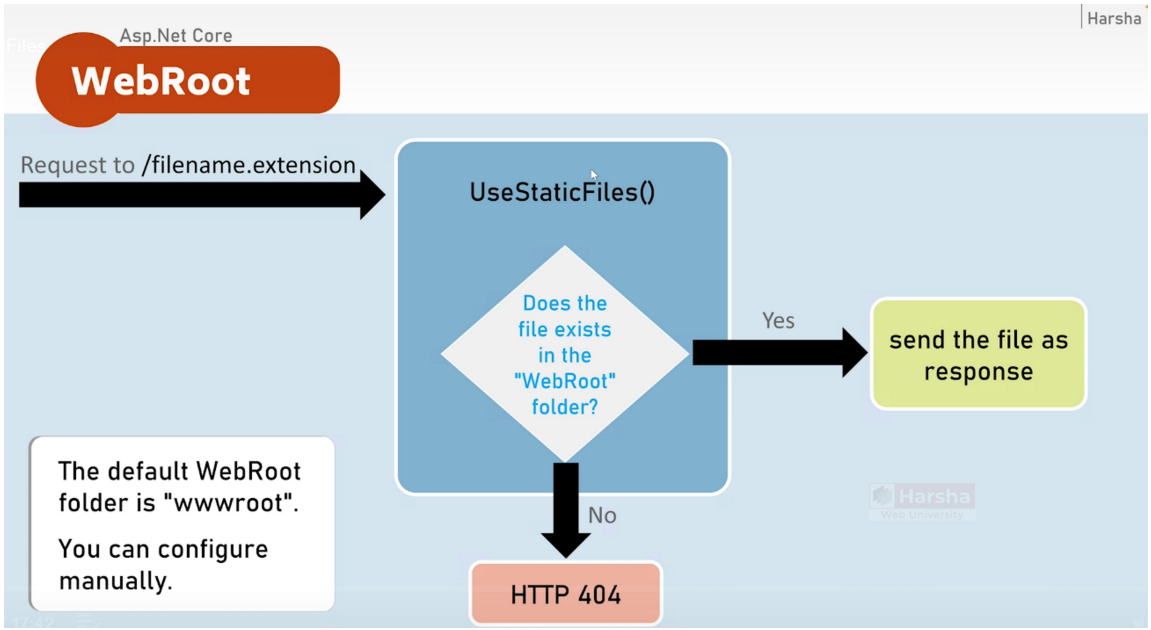
Asp.Net Core

Endpoint Selection Order



Asp.Net Core

WebRoot and UseStaticFiles()



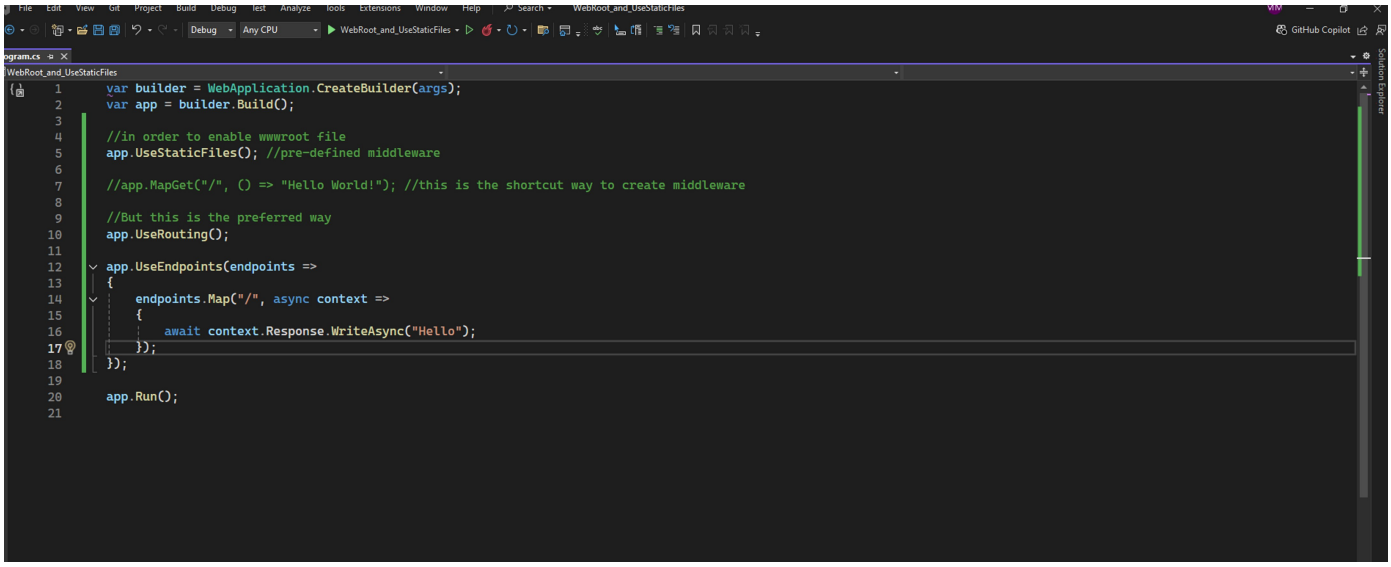
An ASP.NET Core web application may contain static files or content such as images, text files, PDFs, JavaScript files, CSS files, etc. Generally, while working with views, you need to use these types of files.

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.



```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

//in order to enable wwwroot file
app.UseStaticFiles(); //pre-defined middleware

//app.MapGet("/", () => "Hello World!"); //this is the shortcut way to create middleware

//But this is the preferred way
app.UseRouting();

app.UseEndpoints(endpoints =>
{
    endpoints.Map("/", async context =>
    {
        await context.Response.WriteAsync("Hello");
    });
});

app.Run();
```