ASP.NET Core - True Ultimate Guide

Section 5: Routing - Notes

Routing

At its heart, routing is the mechanism that ASP.NET Core uses to match incoming HTTP requests to specific endpoints (e.g., controller actions, Razor Pages, or minimal API handlers) within your application. This allows you to define clean and meaningful URLs that clearly indicate the resources or actions being requested.

How Routing Works in ASP.NET Core

1. **Endpoint Registration:** You define endpoints (routes) within your application, specifying:
    - The URL pattern (e.g., /products, /api/orders/{id}).
    - The HTTP method(s) the endpoint handles (GET, POST, etc.).
    - The code to execute when the endpoint is matched (RequestDelegate).

2. **Request Matching (Middleware):**
    - The UseRouting middleware component is added to the pipeline.
    - When a request arrives, UseRouting analyzes the incoming URL and HTTP method.
    - It compares the URL against your registered endpoints to find the best match.

3. **Endpoint Execution (Middleware):**
    - The UseEndpoints middleware component is added to the pipeline, following UseRouting.
    - If UseRouting found a matching endpoint, UseEndpoints executes the code (the RequestDelegate) associated with that endpoint.

UseRouting vs. UseEndpoints

- **UseRouting:**
    - It's responsible for **route matching** - finding the right endpoint for a given request.
    - It adds route data to the HttpContext, which subsequent middleware can use to make decisions.
    - It **must** come before UseEndpoints.
- **UseEndpoints:**

   - It's responsible for **endpoint execution** - invoking the code (the delegate) associated with the matched endpoint.
   - It also lets you configure the endpoints (e.g., define policies, filters) using lambda expressions.

Map* Methods: Creating Endpoints

ASP.NET Core provides a family of Map* extension methods on the IEndpointRouteBuilder interface that simplify endpoint creation:

- MapGet: Creates an endpoint that only handles GET requests.
- MapPost: Creates an endpoint that only handles POST requests.
- MapPut, MapDelete: Create endpoints for PUT and DELETE requests, respectively.
- MapMethods: Creates an endpoint that handles multiple HTTP methods.
- MapControllerRoute, MapAreaControllerRoute: Used for configuring MVC/Razor Pages controllers.
- MapFallbackToFile: Used to specify a default file to serve when no other endpoint matches.

Code: Detailed Explanation

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


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


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

app.Run(async context => {
    await context.Response.WriteAsync($"Request received at {context.Request.Path}");
});
```

1. **app.UseRouting();:** This line activates routing middleware. It sets up the machinery to analyze incoming requests and match them against your defined endpoints.
2. **app.UseEndpoints(endpoints => { ... });:** This lambda expression configures the endpoints of your application:
    - endpoints.MapGet("map1", ...);: Registers a GET endpoint that responds to the path "/map1" with the text "In Map 1".
    - endpoints.MapPost("map2", ...);: Registers a POST endpoint for the path "/map2", responding with "In Map 2".
3. **app.Run(async context => { ... });:** This is a fallback terminal middleware. If no other endpoint matches the request (e.g., if you visit "/map3"), it will execute this code, writing the requested path to the response.

GetEndpoint()

In ASP.NET Core, the GetEndpoint() method is a powerful tool for retrieving information about the specific endpoint that was selected to handle an incoming HTTP request. This method is an extension method available on the HttpContext object.

- **Purpose:** It allows you to access details about the matched endpoint, such as its display name, route pattern, metadata, and more.
- **When to Use It:** You typically use GetEndpoint() within middleware components to make decisions based on the selected endpoint or to extract information that's relevant to your custom logic.
- **Middleware Placement:** The GetEndpoint() method will return a valid Endpoint object **only after** the UseRouting middleware has executed and successfully matched the request to an endpoint.

Code:

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

var app = builder.Build();
```

```
// Middleware 1: Before Routing
app.Use(async (context, next) =>
{
   Microsoft.AspNetCore.Http.Endpoint? endPoint = context.GetEndpoint();
   if (endPoint != null)
   {
      await context.Response.WriteAsync($"Endpoint: {endPoint.DisplayName}\n");
   }
   await next(context);
});

// Enable Routing Middleware
app.UseRouting();

// Middleware 2: After Routing
app.Use(async (context, next) =>
{
   Microsoft.AspNetCore.Http.Endpoint? endPoint = context.GetEndpoint();
   if (endPoint != null)
   {
      await context.Response.WriteAsync($"Endpoint: {endPoint.DisplayName}\n");
   }
   await next(context);
});

// Creating Endpoints
app.UseEndpoints(endpoints =>
{
   endpoints.MapGet("map1", async (context) =>
   {
      await context.Response.WriteAsync("In Map 1");
```

```
    });

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

// Fallback Middleware
app.Run(async context =>
{
    await context.Response.WriteAsync($"Request received at {context.Request.Path}");
});

app.Run();
```

Let's analyze the code step-by-step:

1. **Middleware 1 (Before Routing):**
    - Here, GetEndpoint() will return null because routing hasn't happened yet. The request hasn't been matched to any specific endpoint.
2. **app.UseRouting();**
    - This enables the routing middleware, which is responsible for matching the request to an endpoint.
3. **Middleware 2 (After Routing):**
    - Now, GetEndpoint() will return the matched endpoint object (if a match was found). You can access its DisplayName (or other properties) to get information about the selected endpoint.

- For a GET request to "/map1", the display name would be "map1".
- For a POST request to "/map2", the display name would be "map2".
- For any other path, the display name would be null (since the fallback middleware handles those cases).

4. **Endpoint Creation:**
    - The app.UseEndpoints section defines your endpoints (routes).

5. **Fallback Middleware:**
    - This middleware handles requests that didn't match any defined endpoints. It simply writes the requested path to the response.

Route Parameters

Route parameters are placeholders within your URL patterns that capture values from incoming requests. These values can then be used within your endpoint handlers to customize the response or perform specific actions.

Types of Route Parameters

1. **Required Parameters:**
    - **Syntax:** Enclosed in curly braces {}.
    - **Behavior:** Must be provided in the URL for the route to match. If not present, the request won't match this endpoint.
    - **Example:** /products/{id} (The id parameter is required).
2. **Optional Parameters:**
    - **Syntax:** Enclosed in curly braces {} and followed by a question mark ?.
    - **Behavior:** Can be omitted from the URL. If not present, the parameter's value will be null.
    - **Example:** /products/details/{id?} (The id parameter is optional).
3. **Parameters with Default Values:**
    - **Syntax:** Enclosed in curly braces {}, followed by an equals sign =, and then the default value.
    - **Behavior:** If not provided in the URL, the parameter will take the specified default value.
    - **Example:** /employee/profile/{EmployeeName=harsha} (The EmployeeName parameter defaults to "harsha").

Code:

```
// ... (UseRouting and other middleware) ...


app.UseEndpoints(endpoints =>
{
   // Required Parameters
```

```
    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}");
    });

    // Default Parameter
    endpoints.Map("employee/profile/{EmployeeName=harsha}", async context =>
    {
        string? employeeName = Convert.ToString(context.Request.RouteValues["employeename"]);
        await context.Response.WriteAsync($"In Employee profile - {employeeName}");
    });

    // Optional Parameter
    endpoints.Map("products/details/{id?}", async context => {
        if (context.Request.RouteValues.ContainsKey("id"))
        {
            int id = Convert.ToInt32(context.Request.RouteValues["id"]);
            await context.Response.WriteAsync($"Products details - {id}");
        }
        else
        {
            await context.Response.WriteAsync($"Products details - id is not supplied");
        }
    });
});

// ... (Fallback middleware) ...
```

1. **Required Parameters Example:**
   - The route files/{filename}.{extension} expects both filename and extension to be present in the URL (e.g., /files/sample.txt).
   - The endpoint handler extracts these values from context.Request.RouteValues and uses them in the response.
2. **Default Parameter Example:**
   - The route employee/profile/{EmployeeName=harsha} has a default value for EmployeeName.
   - If you visit /employee/profile, the response will be "In Employee profile - harsha".
   - If you visit /employee/profile/john, the response will be "In Employee profile - john".
3. **Optional Parameter Example:**
   - The route products/details/{id?} allows the id parameter to be omitted.
   - If you visit /products/details/123, it will show the product details for ID 123.
   - If you visit /products/details, it will indicate that the ID was not provided.

Route Constraints

Route constraints are an essential tool in ASP.NET Core routing that allows you to add extra validation to your route parameters. They define rules that restrict the values a parameter can accept, helping you filter out invalid requests before they reach your endpoint handlers.

Why Use Route Constraints?

- **Enhanced Validation:** Ensure that only requests with valid parameter values are handled.
- **Improved Security:** Prevent malicious input by rejecting requests with potentially harmful values.
- **Cleaner Code:** Avoid cluttering your endpoint handlers with validation logic.
- **Explicit Routing:** Make your routes more self-documenting and easier to understand.

Common Route Constraints

ASP.NET Core provides a variety of built-in route constraints:

- **int:** Requires the parameter value to be an integer.
- **bool:** Requires the parameter value to be a boolean (true or false).

- **datetime:** Requires the parameter value to be a valid date and time string.
- **decimal, double, float, long:** Require the parameter value to be of the specified numeric type.
- **guid:** Requires the parameter value to be a valid GUID (Globally Unique Identifier).
- **alpha:** Requires the parameter value to consist only of alphabetic characters (a-z, A-Z).
- **regex:** Requires the parameter value to match a regular expression pattern.
- **length:** Requires the parameter value to have a specific length or within a specified range.
- **min, max, range:** Require the parameter value to be greater than or equal to the minimum (min), less than or equal to the maximum (max), or within a specific range (range).

Code

```
// ... (UseRouting and other middleware) ...

app.UseEndpoints(endpoints =>
{
  // ... (other endpoints) ...

  // Alphabetic and Length Constraint
  endpoints.Map("employee/profile/{EmployeeName:length(4,7):alpha=harsha}", async context =>
  {
    // ...
  });



  // Integer, Range, and Optional Constraint
  endpoints.Map("products/details/{id:int:range(1,1000)?}", async context => {
    // ...
  });

  // DateTime Constraint
  endpoints.Map("daily-digest-report/{reportdate:datetime}", async context =>
  {
    // ...
  });


  // GUID Constraint
  endpoints.Map("cities/{cityid:guid}", async context =>
  {
    // ...
  });


  // Int, Min, Regex Constraint
  endpoints.Map("sales-report/{year:int:min(1900)}/{month:regex(^(apr|jul|oct|jan)$)}", async
context =>
  {
    // ...
  });
});


// ... (Fallback middleware) ...
```

1. **Alphabetic and Length Constraint:** /employee/profile/{EmployeeName:length(4,7):alpha=harsha}: Ensures EmployeeName is 4-7 characters long and consists only of alphabetic characters. If not supplied, it defaults to "harsha".
2. **Integer, Range, and Optional Constraint:** /products/details/{id:int:range(1,1000)?}: Requires id to be an integer between 1 and 1000. The question mark makes it optional.
3. **DateTime Constraint:** /daily-digest-report/{reportdate:datetime}: Requires reportdate to be a valid date-time string.
4. **GUID Constraint:** /cities/{cityid:guid}: Requires cityid to be a valid GUID.
5. **Integer, Min, and Regex Constraint:** /sales-report/{year:int:min(1900)}/{month:regex(^(apr|jul|oct|jan)$)}: Requires year to be an integer greater than or equal to 1900, and month to be one of the specified values (apr, jul, oct, jan).

Custom Route Constraint Classes

While ASP.NET Core offers a variety of built-in route constraints, sometimes your application requires more specialized validation rules. Custom route constraint classes allow you to define your own criteria for determining whether a parameter value is valid.

Key Requirements

1. **Implement IRouteConstraint:** Create a class that implements the IRouteConstraint interface.
2. **Match Method:** Implement the Match method, which will contain your custom validation logic. This method receives several parameters:
   - httpContext: The current HttpContext.
   - route: The IRouter object associated with the route.
   - routeKey: The name of the route parameter being validated.
   - values: A dictionary containing the route values.
   - routeDirection: Indicates whether the route is being matched for an incoming request or for generating a URL.
3. **Return true or false:** The Match method must return true if the parameter value is valid according to your constraint, and false otherwise.

Code

```
// MonthsCustomConstraint.cs
public class MonthsCustomConstraint : IRouteConstraint
{
  public bool Match(HttpContext? httpContext, IRouter? route, string routeKey,
RouteValueDictionary values, RouteDirection routeDirection)
  {
    // Check if the parameter value exists
    if (!values.ContainsKey(routeKey))
    {
      return false; // Not a match
    }

    Regex regex = new Regex("^(apr|jul|oct|jan)$");
    string? monthValue = Convert.ToString(values[routeKey]);
```

```
        if (regex.IsMatch(monthValue))
        {
            return true; // It's a match
        }
        return false; // Not a match
    }
}
```

Let's break this down:

1. **Implementation of IRouteConstraint:** The MonthsCustomConstraint class clearly implements this interface, signaling that it's a custom route constraint.
2. **Match Method:**
   - It first checks if the values dictionary contains the route parameter being validated (routeKey). If not, it's an immediate mismatch, and false is returned.
   - A regular expression (^(apr|jul|oct|jan)$) is used to define the valid month values.
   - The value associated with the routeKey is retrieved from the values dictionary and converted to a string.
   - The Regex.IsMatch method tests whether the retrieved value matches the allowed month pattern.
   - Returns true if the value matches, and false otherwise.

Using the Custom Constraint

```
// ... (in your endpoint configuration) ...
endpoints.Map("sales-report/{year:int:min(1900)}/{month:months}", async context =>
{
    // ... your endpoint handler logic ...
});
```

- Notice the :months constraint after the month parameter. This indicates that the value for month should be validated against the MonthsCustomConstraint class.

Endpoint Selection

When a request arrives at your ASP.NET Core application, the routing middleware (UseRouting) analyzes the URL and HTTP method. It then compares this information against the collection of endpoints you've defined using methods like MapGet, MapPost, etc. The goal is to find the most suitable endpoint to handle the request.

However, what happens when multiple endpoints seem like potential matches? ASP.NET Core employs a well-defined algorithm to determine the winning endpoint.

Endpoint Selection Algorithm

1. **Precedence:**
   - **Explicit Matches:** Endpoints defined with more specific patterns (e.g., /products/{id}) take precedence over those with broader patterns (e.g., /products).
   - **Order of Registration:** If multiple endpoints with equally specific patterns could match, the endpoint that was registered *first* wins.
2. **HTTP Method:**
   - **Exact Match:** If the request method (GET, POST, etc.) exactly matches the method specified for an endpoint, that endpoint is preferred.
3. **Route Constraints:**
   - **More Specific Constraints:** Endpoints with more restrictive route constraints (e.g., id:int:range(1,100) vs. id:int) are favored.
4. **Catch-All (Fallback):**
   - If no other endpoint matches, and you have a catch-all endpoint (defined using MapFallback), it will be selected.

Order of Precedence: A Visual Summary

1. Explicit Match with Exact HTTP Method and More Specific Route Constraints
2. Explicit Match with Exact HTTP Method and Less Specific Route Constraints
3. Explicit Match with Any HTTP Method and More Specific Route Constraints
4. Explicit Match with Any HTTP Method and Less Specific Route Constraints
5. Order of Registration (if specificity is equal)
6. Catch-All Endpoint (if no other match is found)

Practical Implications and Tips

- **Mind Your Order:** Be mindful of the order in which you register your endpoints, especially if they have similar patterns.

- **Specificity Wins:** Define your routes as specifically as possible to avoid ambiguity.
- **Route Constraints:** Use route constraints to narrow down the valid values for parameters.
- **Catch-All with Caution:** Catch-all endpoints can be useful, but use them sparingly to avoid unintended matches.
- **Endpoint Metadata:** Explore the Endpoint object's metadata for insights into why a particular endpoint was selected.

Code

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapGet("/products/{id:int}", GetProductById); // Most specific
    endpoints.MapGet("/products", GetAllProducts);        // Less specific
    endpoints.MapGet("/{path?}", CatchAllHandler);         // Catch-all
});
```

In this example:

- /products/123 will match the first endpoint (GetProductById).
- /products will match the second endpoint (GetAllProducts).
- /anything-else will match the catch-all endpoint (CatchAllHandler).

Resolving Ambiguity

If the routing system cannot definitively determine the best match, you'll encounter an AmbiguousMatchException. This exception signals that you need to refine your route definitions or registration order to eliminate the conflict.

Static Files in ASP.NET Core

Static files are the assets that make up the visual presentation and functionality of your web application:

- **HTML Files:** The structure of your web pages.
- **CSS Stylesheets:** The styling and appearance of your content.
- **JavaScript Files:** The interactive elements and logic of your application.
- **Images:** Visual elements that enhance the user experience.

ASP.NET Core provides the UseStaticFiles() middleware component to efficiently serve these static files directly to the browser without requiring any server-side processing.

WebRoot: The Default Location

The WebRoot property in ASP.NET Core specifies the default directory from which static files are served. By default, this directory is named "wwwroot" and is located at the root of your project. However, you can customize this location if needed.

UseStaticFiles() Middleware: Enabling Static File Serving

- **Basic Usage:** Calling app.UseStaticFiles(); with no arguments will serve static files from the default WebRoot directory.
- **Customization:** You can customize the behavior of UseStaticFiles() by passing a StaticFileOptions object:
  - FileProvider: Specify a different file provider (e.g., PhysicalFileProvider) to serve files from a custom location.
  - RequestPath: Configure the base URL path for your static files (e.g., /static).
  - ContentTypeProvider: Customize how content types are determined for different file extensions.
  - OnPrepareResponse: Perform additional actions on the response before it's sent to the client.

Code

```
using Microsoft.Extensions.FileProviders;


// ...


var builder = WebApplication.CreateBuilder(new WebApplicationOptions()
{
   WebRootPath = "myroot"
});
var app = builder.Build();


// Serve from the specified WebRoot ("myroot" in this case)
app.UseStaticFiles();
```

```
// Serve from a custom directory ("mywebroot") located within the project's ContentRootPath
app.UseStaticFiles(new StaticFileOptions()
{
  FileProvider = new PhysicalFileProvider(
    Path.Combine(builder.Environment.ContentRootPath, "mywebroot")
  )
});
// ... (rest of your middleware and endpoints) ...
```

Explanation

1. **Custom WebRoot:** The WebRootPath property in WebApplicationOptions is set to "myroot", making "myroot" the default location for static files served by the first app.UseStaticFiles().
2. **Default Static Files:** The initial app.UseStaticFiles(); call serves files directly from the "myroot" directory. For instance, a request to /styles.css would look for a file named styles.css within "myroot".
3. **Custom Static Files Location:** The second app.UseStaticFiles call configures a PhysicalFileProvider to serve files from a custom location: "mywebroot". This directory is located within the application's ContentRootPath (the project's root folder).

Important Considerations

- **Security:** Always be cautious about the files you expose as static content. Avoid placing sensitive information in your WebRoot or custom directories.
- **Performance:** Consider using caching and compression techniques to optimize the delivery of static files.
- **Content Security Policy (CSP):** Implement a CSP to mitigate cross-site scripting (XSS) attacks that could exploit your static files.

By effectively managing your static files and utilizing the UseStaticFiles() middleware, you can enhance your ASP.NET Core application's performance and user experience.

KeyPoints to remember:

Routing

- **Purpose:** Matches incoming HTTP requests to specific endpoints (controllers, Razor Pages, minimal APIs) in your application.
- **Middleware:** UseRouting and UseEndpoints are essential middleware components for routing.
    - UseRouting: Analyzes the request URL and matches it to an endpoint.
    - UseEndpoints: Executes the matched endpoint's code.
- **Map* Methods:** Used to define endpoints for different HTTP methods (e.g., MapGet, MapPost, MapControllerRoute).

Endpoint Selection Order

- **Specificity:** More specific routes (with more parameters or constraints) take precedence over less specific ones.
- **Registration Order:** If multiple routes are equally specific, the one registered first wins.
- **HTTP Method:** Routes with an exact method match are preferred.
- **Route Constraints:** Routes with more restrictive constraints are favored.
- **Catch-All:** A fallback endpoint handles unmatched requests.

Route Parameters

- **Types:** Required ({id}), optional ({id?}), default value ({id=123}).
- **Access:** Parameter values are accessed through context.Request.RouteValues.

Route Constraints

- **Purpose:** Restrict the allowed values for route parameters.
- **Built-in:** int, bool, datetime, guid, regex, length, min, max, range, etc.
- **Custom:** Create classes implementing IRouteConstraint to define your own validation logic.

GetEndpoint()

- **Purpose:** Retrieves information about the matched endpoint.
- **Usage:** Call context.GetEndpoint() within middleware **after** UseRouting.
- **Information:** Access endpoint properties like DisplayName, route pattern, and metadata.

Static Files

- **WebRoot:** The default directory from which static files are served (usually "wwwroot").
- **UseStaticFiles():** Middleware for serving static files (HTML, CSS, JavaScript, images).
- **Customization:** Use StaticFileOptions to change the file provider, request path, or other settings.

Key Interview Tips

- **Explain the Flow:** Clearly articulate how a request flows through the routing middleware and how endpoints are selected.
- **Code Examples:** Be prepared to write code snippets demonstrating endpoint registration, parameter usage, and constraint application.
- **Troubleshooting:** Explain how you would diagnose and fix common routing issues (e.g., 404 errors, ambiguous matches).
- **Best Practices:** Discuss how to design clean, maintainable, and secure routes.