

Version

ASP.NET Core in .NET 9.0

Filter by title

- ASP.NET Core documentation
- Overview
- Get started
- What's new
- Tutorials
- Fundamentals
 - Overview
 - App start up
 - Dependency injection (services)
 - Native AOT
 - Middleware
 - Host
 - Configuration
 - Options
 - Environments (dev, stage, prod)
 - Logging and monitoring
 - HttpContext
 - Routing
 - Handle errors
 - Make HTTP requests
 - Static files
 - dotnet-scaffold telemetry
- Web apps
- APIs
- Real-time apps
- Remote Procedure Call apps
- Best practices
- Servers
- Test
- Debug
- Troubleshoot
- Code analysis
- Data access
- Host and deploy
- Security and Identity
- Performance
- Globalization and localization
- Advanced
- Migration and updates
- API reference

Download PDF

Learn / .NET / ASP.NET Core /

Feedback

Routing in ASP.NET Core

Article • 09/18/2024 • 43 contributors

In this article

- Routing basics
- Routing concepts
- Route templates
- Routing with special characters
- Show 12 more

By [Ryan Nowak](#), [Kirk Larkin](#), and [Rick Anderson](#)

Routing is responsible for matching incoming HTTP requests and dispatching those requests to the app's executable endpoints. [Endpoints](#) are the app's units of executable request-handling code. Endpoints are defined in the app and configured when the app starts. The endpoint matching process can extract values from the request's URL and provide those values for request processing. Using endpoint information from the app, routing is also able to generate URLs that map to endpoints.

Apps can configure routing using:

- Controllers
- Razor Pages
- SignalR
- gRPC Services
- Endpoint-enabled [middleware](#) such as [Health Checks](#).
- Delegates and lambdas registered with routing.

This article covers low-level details of ASP.NET Core routing. For information on configuring routing:

- For controllers, see [Routing to controller actions in ASP.NET Core](#).
- For Razor Pages conventions, see [Razor Pages route and app conventions in ASP.NET Core](#).
- For Blazor routing guidance, which adds to or supersedes the guidance in this article, see [ASP.NET Core Blazor routing and navigation](#).

Routing basics

The following code shows a basic example of routing:

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

app.MapGet("/", () => "Hello World!");

app.Run();
```

The preceding example includes a single endpoint using the [MapGet](#) method:

- When an HTTP `GET` request is sent to the root URL `/`:
 - The request delegate executes.
 - `Hello World!` is written to the HTTP response.
- If the request method is not `GET` or the root URL is not `/`, no route matches and an HTTP 404 is returned.

Routing uses a pair of middleware, registered by [UseRouting](#) and [UseEndpoints](#):

Additional resources

Events

Be one of the first to start using SQL database in Fabric

Dec 3, 11 PM - Dec 12, 11 PM

Get all your questions answered in live sessions starting December 3rd.
[Register now](#)

Training

Module
[Use pages, routing, and layouts to improve Blazor navigation - Training](#)

Learn how to optimize your app's navigation, use parameters from the URL, and create reusable layouts in a Blazor web app.

- `UseRouting` adds route matching to the middleware pipeline. This middleware looks at the set of endpoints defined in the app, and selects the **best match** based on the request.
- `UseEndpoints` adds endpoint execution to the middleware pipeline. It runs the delegate associated with the selected endpoint.

Apps typically don't need to call `UseRouting` or `UseEndpoints`. `WebApplicationBuilder` configures a middleware pipeline that wraps middleware added in `Program.cs` with `UseRouting` and `UseEndpoints`. However, apps can change the order in which `UseRouting` and `UseEndpoints` run by calling these methods explicitly. For example, the following code makes an explicit call to `UseRouting`:

```
app.Use(async (context, next) =>
{
    // ...
    await next(context);
});

app.UseRouting();

app.MapGet("/", () => "Hello World!");
```

In the preceding code:

- The call to `app.Use` registers a custom middleware that runs at the start of the pipeline.
- The call to `UseRouting` configures the route matching middleware to run *after* the custom middleware.
- The endpoint registered with `MapGet` runs at the end of the pipeline.

If the preceding example didn't include a call to `UseRouting`, the custom middleware would run *after* the route matching middleware.

Note: Routes added directly to the `WebApplication` execute at the **end** of the pipeline.

Endpoints

The `MapGet` method is used to define an **endpoint**. An endpoint is something that can be:

- Selected, by matching the URL and HTTP method.
- Executed, by running the delegate.

Endpoints that can be matched and executed by the app are configured in `UseEndpoints`. For example, `MapGet`, `MapPost`, and [similar methods](#) connect request delegates to the routing system. Additional methods can be used to connect ASP.NET Core framework features to the routing system:

- [MapRazorPages for Razor Pages](#)
- [MapControllers for controllers](#)
- [MapHub<THub> for SignalR](#)
- [MapGrpcService<TService> for gRPC](#)

The following example shows routing with a more sophisticated route template:

```
app.MapGet("/hello/{name:alpha}", (string name) => $"Hello {name}!");
```

The string `/hello/{name:alpha}` is a **route template**. A route template is used to configure how the endpoint is matched. In this case, the template matches:

- A URL like `/hello/Docs`
- Any URL path that begins with `/hello/` followed by a sequence of alphabetic characters. `:alpha` applies a route constraint that matches only alphabetic characters. [Route constraints](#) are explained later in this article.

The second segment of the URL path, `{name:alpha}`:

- Is bound to the `name` parameter.
- Is captured and stored in `HttpRequest.RouteValues`.

The following example shows routing with [health checks](#) and authorization:

```
app.UseAuthentication();
app.UseAuthorization();

app.MapHealthChecks("/healthz").RequireAuthorization();
app.MapGet("/", () => "Hello World!");
```

The preceding example demonstrates how:

- The authorization middleware can be used with routing.
- Endpoints can be used to configure authorization behavior.

The `MapHealthChecks` call adds a health check endpoint. Chaining `RequireAuthorization` on to this call attaches an authorization policy to the endpoint.

Calling `UseAuthentication` and `UseAuthorization` adds the authentication and authorization middleware. These middleware are placed between `UseRouting` and `UseEndpoints` so that they can:

- See which endpoint was selected by `UseRouting`.
- Apply an authorization policy before `UseEndpoints` dispatches to the endpoint.

Endpoint metadata

In the preceding example, there are two endpoints, but only the health check endpoint has an authorization policy attached. If the request matches the health check endpoint, `/healthz`, an authorization check is performed. This demonstrates that endpoints can have extra data attached to them. This extra data is called endpoint **metadata**:

- The metadata can be processed by routing-aware middleware.
- The metadata can be of any .NET type.

Routing concepts

The routing system builds on top of the middleware pipeline by adding the powerful **endpoint** concept. Endpoints represent units of the app's functionality that are distinct from each other in terms of routing, authorization, and any number of ASP.NET Core's systems.

ASP.NET Core endpoint definition

An ASP.NET Core endpoint is:

- Executable: Has a `RequestDelegate`.
- Extensible: Has a `Metadata` collection.
- Selectable: Optionally, has [routing information](#).
- Enumerable: The collection of endpoints can be listed by retrieving the `EndpointDataSource` from DI.

The following code shows how to retrieve and inspect the endpoint matching the current request:

```
app.Use(async (context, next) =>
{
    var currentEndpoint = context.GetEndpoint();

    if (currentEndpoint is null)
    {
        await next(context);
        return;
    }

    Console.WriteLine($"Endpoint: {currentEndpoint.DisplayName}");

    if (currentEndpoint is RouteEndpoint routeEndpoint)
    {
        Console.WriteLine($" - Route Pattern: {routeEndpoint.RoutePattern}");
    }

    foreach (var endpointMetadata in currentEndpoint.Metadata)
    {
        Console.WriteLine($" - Metadata: {endpointMetadata}");
    }

    await next(context);
});
```

```
app.MapGet("/", () => "Inspect Endpoint.");
```

The endpoint, if selected, can be retrieved from the `HttpContext`. Its properties can be inspected. Endpoint objects are immutable and cannot be modified after creation. The most common type of endpoint is a `RouteEndpoint`. `RouteEndpoint` includes information that allows it to be selected by the routing system.

In the preceding code, `app.Use` configures an inline `middleware`.

The following code shows that, depending on where `app.Use` is called in the pipeline, there may not be an endpoint:

```
// Location 1: before routing runs, endpoint is always null here.
app.Use(async (context, next) =>
{
    Console.WriteLine($"1. Endpoint: {context.GetEndpoint()?.DisplayName}");
    await next(context);
});

app.UseRouting();

// Location 2: after routing runs, endpoint will be non-null if routing found a match
app.Use(async (context, next) =>
{
    Console.WriteLine($"2. Endpoint: {context.GetEndpoint()?.DisplayName}");
    await next(context);
});

// Location 3: runs when this endpoint matches
app.MapGet("/", (HttpContext context) =>
{
    Console.WriteLine($"3. Endpoint: {context.GetEndpoint()?.DisplayName}");
    return "Hello World!";
}).WithDisplayName("Hello");

app.UseEndpoints(_ => { });

// Location 4: runs after UseEndpoints - will only run if there was no match
app.Use(async (context, next) =>
{
    Console.WriteLine($"4. Endpoint: {context.GetEndpoint()?.DisplayName}");
    await next(context);
});
```

The preceding sample adds `Console.WriteLine` statements that display whether or not an endpoint has been selected. For clarity, the sample assigns a display name to the provided `/` endpoint.

The preceding sample also includes calls to `UseRouting` and `UseEndpoints` to control exactly when these middleware run within the pipeline.

Running this code with a URL of `/` displays:

```
1. Endpoint: (null)
2. Endpoint: Hello
3. Endpoint: Hello
```

Running this code with any other URL displays:

```
1. Endpoint: (null)
2. Endpoint: (null)
4. Endpoint: (null)
```

This output demonstrates that:

- The endpoint is always null before `UseRouting` is called.
- If a match is found, the endpoint is non-null between `UseRouting` and `UseEndpoints`.
- The `UseEndpoints` middleware is **terminal** when a match is found. `Terminal middleware` is defined later in this article.
- The middleware after `UseEndpoints` execute only when no match is found.

The `UseRouting` middleware uses the `SetEndpoint` method to attach the endpoint to the current context. It's possible to replace the `UseRouting` middleware with custom logic

and still get the benefits of using endpoints. Endpoints are a low-level primitive like middleware, and aren't coupled to the routing implementation. Most apps don't need to replace `UseRouting` with custom logic.

The `UseEndpoints` middleware is designed to be used in tandem with the `UseRouting` middleware. The core logic to execute an endpoint isn't complicated. Use `GetEndpoint` to retrieve the endpoint, and then invoke its `RequestDelegate` property.

The following code demonstrates how middleware can influence or react to routing:

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

app.Use(async (context, next) =>
{
    if (context.GetEndpoint()?.Metadata.GetMetadata<RequiresAuditAttribute>() != null)
    {
        Console.WriteLine($"ACCESS TO SENSITIVE DATA AT: {DateTime.UtcNow}");
    }

    await next(context);
});

app.MapGet("/", () => "Audit isn't required.");
app.MapGet("/sensitive", () => "Audit required for sensitive data.")
    .WithMetadata(new RequiresAuditAttribute());
```

```
public class RequiresAuditAttribute : Attribute { }
```

The preceding example demonstrates two important concepts:

- Middleware can run before `UseRouting` to modify the data that routing operates upon.
 - Usually middleware that appears before routing modifies some property of the request, such as `UseRewriter`, `UseHttpMethodOverride`, or `UsePathBase`.
- Middleware can run between `UseRouting` and `UseEndpoints` to process the results of routing before the endpoint is executed.
 - Middleware that runs between `UseRouting` and `UseEndpoints`:
 - Usually inspects metadata to understand the endpoints.
 - Often makes security decisions, as done by `UseAuthorization` and `UseCors`.
 - The combination of middleware and metadata allows configuring policies per-endpoint.

The preceding code shows an example of a custom middleware that supports per-endpoint policies. The middleware writes an *audit log* of access to sensitive data to the console. The middleware can be configured to *audit* an endpoint with the `RequiresAuditAttribute` metadata. This sample demonstrates an *opt-in* pattern where only endpoints that are marked as sensitive are audited. It's possible to define this logic in reverse, auditing everything that isn't marked as safe, for example. The endpoint metadata system is flexible. This logic could be designed in whatever way suits the use case.

The preceding sample code is intended to demonstrate the basic concepts of endpoints.

The sample is not intended for production use. A more complete version of an *audit log* middleware would:

- Log to a file or database.
- Include details such as the user, IP address, name of the sensitive endpoint, and more.

The audit policy metadata `RequiresAuditAttribute` is defined as an `Attribute` for easier use with class-based frameworks such as controllers and SignalR. When using *route to code*:

- Metadata is attached with a builder API.
- Class-based frameworks include all attributes on the corresponding method and class when creating endpoints.

The best practices for metadata types are to define them either as interfaces or attributes. Interfaces and attributes allow code reuse. The metadata system is flexible and doesn't impose any limitations.

Compare terminal middleware with routing

The following example demonstrates both terminal middleware and routing:

```
// Approach 1: Terminal Middleware.
app.Use(async (context, next) =>
{
    if (context.Request.Path == "/")
    {
        await context.Response.WriteAsync("Terminal Middleware.");
        return;
    }

    await next(context);
});

app.UseRouting();

// Approach 2: Routing.
app.MapGet("/Routing", () => "Routing.");
```

The style of middleware shown with [Approach 1](#) is **terminal middleware**. It's called terminal middleware because it does a matching operation:

- The matching operation in the preceding sample is `Path == "/"` for the middleware and `Path == "/Routing"` for routing.
- When a match is successful, it executes some functionality and returns, rather than invoking the `next` middleware.

It's called terminal middleware because it terminates the search, executes some functionality, and then returns.

The following list compares terminal middleware with routing:

- Both approaches allow terminating the processing pipeline:
 - Middleware terminates the pipeline by returning rather than invoking `next`.
 - Endpoints are always terminal.
- Terminal middleware allows positioning the middleware at an arbitrary place in the pipeline:
 - Endpoints execute at the position of [UseEndpoints](#).
- Terminal middleware allows arbitrary code to determine when the middleware matches:
 - Custom route matching code can be verbose and difficult to write correctly.
 - Routing provides straightforward solutions for typical apps. Most apps don't require custom route matching code.
- Endpoints interface with middleware such as `UseAuthorization` and `UseCors`.
 - Using a terminal middleware with `UseAuthorization` or `UseCors` requires manual interfacing with the authorization system.

An [endpoint](#) defines both:

- A delegate to process requests.
- A collection of arbitrary metadata. The metadata is used to implement cross-cutting concerns based on policies and configuration attached to each endpoint.

Terminal middleware can be an effective tool, but can require:

- A significant amount of coding and testing.
- Manual integration with other systems to achieve the desired level of flexibility.

Consider integrating with routing before writing a terminal middleware.

Existing terminal middleware that integrates with `Map` or `MapWhen` can usually be turned into a routing aware endpoint. [MapHealthChecks](#) [↗] demonstrates the pattern for routerware:

- Write an extension method on `IEndpointRouteBuilder`.
- Create a nested middleware pipeline using `CreateApplicationBuilder`.
- Attach the middleware to the new pipeline. In this case, [UseHealthChecks](#).
- [Build](#) the middleware pipeline into a `RequestDelegate`.
- Call `Map` and provide the new middleware pipeline.
- Return the builder object provided by `Map` from the extension method.

The following code shows use of [MapHealthChecks](#):

```
app.UseAuthentication();
app.UseAuthorization();

app.MapHealthChecks("/healthz").RequireAuthorization();
```

The preceding sample shows why returning the builder object is important. Returning the builder object allows the app developer to configure policies such as authorization for the endpoint. In this example, the health checks middleware has no direct integration with the authorization system.

The metadata system was created in response to the problems encountered by extensibility authors using terminal middleware. It's problematic for each middleware to implement its own integration with the authorization system.

URL matching

- Is the process by which routing matches an incoming request to an [endpoint](#).
- Is based on data in the URL path and headers.
- Can be extended to consider any data in the request.

When a routing middleware executes, it sets an [Endpoint](#) and route values to a [request feature](#) on the [HttpContext](#) from the current request:

- Calling [HttpContext.GetEndpoint](#) gets the endpoint.
- [HttpRequest.RouteValues](#) gets the collection of route values.

[Middleware](#) that runs after the routing middleware can inspect the endpoint and take action. For example, an authorization middleware can interrogate the endpoint's metadata collection for an authorization policy. After all of the middleware in the request processing pipeline is executed, the selected endpoint's delegate is invoked.

The routing system in endpoint routing is responsible for all dispatching decisions. Because the middleware applies policies based on the selected endpoint, it's important that:

- Any decision that can affect dispatching or the application of security policies is made inside the routing system.

⚠ Warning

For backward-compatibility, when a Controller or Razor Pages endpoint delegate is executed, the properties of [RouteContext.RouteData](#) are set to appropriate values based on the request processing performed thus far.

The [RouteContext](#) type will be marked obsolete in a future release:

- Migrate [RouteData.Values](#) to [HttpRequest.RouteValues](#).
- Migrate [RouteData.DataTokens](#) to retrieve [IDataTokensMetadata](#) from the endpoint metadata.

URL matching operates in a configurable set of phases. In each phase, the output is a set of matches. The set of matches can be narrowed down further by the next phase. The routing implementation does not guarantee a processing order for matching endpoints. **All** possible matches are processed at once. The URL matching phases occur in the following order. ASP.NET Core:

1. Processes the URL path against the set of endpoints and their route templates, collecting **all** of the matches.
2. Takes the preceding list and removes matches that fail with route constraints applied.
3. Takes the preceding list and removes matches that fail the set of [MatcherPolicy](#) instances.
4. Uses the [EndpointSelector](#) to make a final decision from the preceding list.

The list of endpoints is prioritized according to:

- The [RouteEndpoint.Order](#)
- The [route template precedence](#)

All matching endpoints are processed in each phase until the [EndpointSelector](#) is reached. The [EndpointSelector](#) is the final phase. It chooses the highest priority endpoint from

the matches as the best match. If there are other matches with the same priority as the best match, an ambiguous match exception is thrown.

The route precedence is computed based on a **more specific** route template being given a higher priority. For example, consider the templates `/hello` and `/message`:

- Both match the URL path `/hello`.
- `/hello` is more specific and therefore higher priority.

In general, route precedence does a good job of choosing the best match for the kinds of URL schemes used in practice. Use [Order](#) only when necessary to avoid an ambiguity.

Due to the kinds of extensibility provided by routing, it isn't possible for the routing system to compute ahead of time the ambiguous routes. Consider an example such as the route templates `/message:alpha` and `/message:int`:

- The `alpha` constraint matches only alphabetic characters.
- The `int` constraint matches only numbers.
- These templates have the same route precedence, but there's no single URL they both match.
- If the routing system reported an ambiguity error at startup, it would block this valid use case.

Warning

The order of operations inside [UseEndpoints](#) doesn't influence the behavior of routing, with one exception. [MapControllerRoute](#) and [MapAreaRoute](#) automatically assign an order value to their endpoints based on the order they are invoked. This simulates long-time behavior of controllers without the routing system providing the same guarantees as older routing implementations.

Endpoint routing in ASP.NET Core:

- Doesn't have the concept of routes.
- Doesn't provide ordering guarantees. All endpoints are processed at once.

Route template precedence and endpoint selection order

[Route template precedence](#) is a system that assigns each route template a value based on how specific it is. Route template precedence:

- Avoids the need to adjust the order of endpoints in common cases.
- Attempts to match the common-sense expectations of routing behavior.

For example, consider templates `/Products/List` and `/Products/{id}`. It would be reasonable to assume that `/Products/List` is a better match than `/Products/{id}` for the URL path `/Products/List`. This works because the literal segment `/List` is considered to have better precedence than the parameter segment `/id`.

The details of how precedence works are coupled to how route templates are defined:

- Templates with more segments are considered more specific.
- A segment with literal text is considered more specific than a parameter segment.
- A parameter segment with a constraint is considered more specific than one without.
- A complex segment is considered as specific as a parameter segment with a constraint.
- Catch-all parameters are the least specific. See **catch-all** in the [Route templates](#) section for important information on catch-all routes.

URL generation concepts

URL generation:

- Is the process by which routing can create a URL path based on a set of route values.
- Allows for a logical separation between endpoints and the URLs that access them.

Endpoint routing includes the [LinkGenerator](#) API. `LinkGenerator` is a singleton service available from [DI](#). The `LinkGenerator` API can be used outside of the context of an executing request. [Mvc.UrlHelper](#) and scenarios that rely on [IUrlHelper](#), such as [Tag](#)

[Helpers](#), [HTML Helpers](#), and [Action Results](#), use the `LinkGenerator` API internally to provide link generating capabilities.

The link generator is backed by the concept of an **address** and **address schemes**. An address scheme is a way of determining the endpoints that should be considered for link generation. For example, the route name and route values scenarios many users are familiar with from controllers and Razor Pages are implemented as an address scheme.

The link generator can link to controllers and Razor Pages via the following extension methods:

- [GetPathByAction](#)
- [GetUriByAction](#)
- [GetPathByPage](#)
- [GetUriByPage](#)

Overloads of these methods accept arguments that include the `HttpContext`. These methods are functionally equivalent to [Url.Action](#) and [Url.Page](#), but offer additional flexibility and options.

The `GetPath*` methods are most similar to `Url.Action` and `Url.Page`, in that they generate a URI containing an absolute path. The `GetUri*` methods always generate an absolute URI containing a scheme and host. The methods that accept an `HttpContext` generate a URI in the context of the executing request. The [ambient](#) route values, URL base path, scheme, and host from the executing request are used unless overridden.


`LinkGenerator` is called with an address. Generating a URI occurs in two steps:

1. An address is bound to a list of endpoints that match the address.
2. Each endpoint's [RoutePattern](#) is evaluated until a route pattern that matches the supplied values is found. The resulting output is combined with the other URI parts supplied to the link generator and returned.

The methods provided by `LinkGenerator` support standard link generation capabilities for any type of address. The most convenient way to use the link generator is through extension methods that perform operations for a specific address type:

 [Expand table](#)

Extension Method	Description
GetPathByAddress	Generates a URI with an absolute path based on the provided values.
GetUriByAddress	Generates an absolute URI based on the provided values.

 **Warning**

Pay attention to the following implications of calling `LinkGenerator` methods:

- Use `GetUri*` extension methods with caution in an app configuration that doesn't validate the `Host` header of incoming requests. If the `Host` header of incoming requests isn't validated, untrusted request input can be sent back to the client in URIs in a view or page. We recommend that all production apps configure their server to validate the `Host` header against known valid values.
- Use `LinkGenerator` with caution in middleware in combination with `Map` or `MapWhen`. `Map*` changes the base path of the executing request, which affects the output of link generation. All of the `LinkGenerator` APIs allow specifying a base path. Specify an empty base path to undo the `Map*` affect on link generation.

Middleware example

In the following example, a middleware uses the `LinkGenerator` API to create a link to an action method that lists store products. Using the link generator by injecting it into a class and calling `GenerateLink` is available to any class in an app:

```
public class ProductsMiddleware
{
    private readonly LinkGenerator _linkGenerator;

    public ProductsMiddleware(RequestDelegate next, LinkGenerator linkGen
```

```

        _linkGenerator = linkGenerator;

        public async Task InvokeAsync(HttpContext httpContext)
        {
            httpContext.Response.ContentType = MediaTypeNames.Text.Plain;

            var productsPath = _linkGenerator.GetPathByAction("Products", "Sto

            await httpContext.Response.WriteAsync(
                $"Go to {productsPath} to see our products.");
        }
    }
}

```

Route templates

Tokens within `{ }` define route parameters that are bound if the route is matched. More than one route parameter can be defined in a route segment, but route parameters must be separated by a literal value. For example:

```
{controller=Home}{action=Index}
```

isn't a valid route, because there's no literal value between `{controller}` and `{action}`. Route parameters must have a name and may have additional attributes specified.

Literal text other than route parameters (for example, `{id}`) and the path separator `/` must match the text in the URL. Text matching is case-insensitive and based on the decoded representation of the URL's path. To match a literal route parameter delimiter `{` or `}`, escape the delimiter by repeating the character. For example `{{` or `}}`.

Asterisk `*` or double asterisk `**`:

- Can be used as a prefix to a route parameter to bind to the rest of the URI.
- Are called a **catch-all** parameters. For example, `blog/{**slug}`:
 - Matches any URI that starts with `blog/` and has any value following it.
 - The value following `blog/` is assigned to the `slug` route value.

Catch-all parameters can also match the empty string.

The catch-all parameter escapes the appropriate characters when the route is used to generate a URL, including path separator `/` characters. For example, the route `foo/{*path}` with route values `{ path = "my/path" }` generates `foo/my%2Fpath`. Note the escaped forward slash. To round-trip path separator characters, use the `**` route parameter prefix. The route `foo/{**path}` with `{ path = "my/path" }` generates `foo/my/path`.

URL patterns that attempt to capture a file name with an optional file extension have additional considerations. For example, consider the template `files/{filename}.{ext?}`. When values for both `filename` and `ext` exist, both values are populated. If only a value for `filename` exists in the URL, the route matches because the trailing `.` is optional. The following URLs match this route:

- `/files/myFile.txt`
- `/files/myFile`

Route parameters may have **default values** designated by specifying the default value after the parameter name separated by an equals sign (`=`). For example, `{controller=Home}` defines `Home` as the default value for `controller`. The default value is used if no value is present in the URL for the parameter. Route parameters are made optional by appending a question mark (`?`) to the end of the parameter name. For example, `id?`. The difference between optional values and default route parameters is:

- A route parameter with a default value always produces a value.
- An optional parameter has a value only when a value is provided by the request URL.

Route parameters may have constraints that must match the route value bound from the URL. Adding `:` and constraint name after the route parameter name specifies an inline constraint on a route parameter. If the constraint requires arguments, they're enclosed in parentheses `(...)` after the constraint name. Multiple *inline constraints* can be specified by appending another `:` and constraint name.

The constraint name and arguments are passed to the [InlineConstraintResolver](#) service to create an instance of [IRouteConstraint](#) to use in URL processing. For example, the route template `blog/{article:minlength(10)}` specifies a `minlength` constraint with the

argument `10`. For more information on route constraints and a list of the constraints provided by the framework, see the [Route constraints](#) section.

Route parameters may also have parameter transformers. Parameter transformers transform a parameter's value when generating links and matching actions and pages to URLs. Like constraints, parameter transformers can be added inline to a route parameter by adding a `:` and transformer name after the route parameter name. For example, the route template `blog/{article:slugify}` specifies a `slugify` transformer. For more information on parameter transformers, see the [Parameter transformers](#) section.

The following table demonstrates example route templates and their behavior:

 [Expand table](#)

Route Template	Example Matching URI	The request URI...
hello	/hello	Only matches the single path <code>/hello</code> .
{Page=Home}	/	Matches and sets <code>Page</code> to <code>Home</code> .
{Page=Home}	/Contact	Matches and sets <code>Page</code> to <code>Contact</code> .
{controller}/{action}/{id?}	/Products/List	Maps to the <code>Products</code> controller and <code>List</code> action.
{controller}/{action}/{id?}	/Products/Details/123	Maps to the <code>Products</code> controller and <code>Details</code> action with <code>id</code> set to <code>123</code> .
{controller=Home}/{action=Index}/{id?}	/	Maps to the <code>Home</code> controller and <code>Index</code> method. <code>id</code> is ignored.
{controller=Home}/{action=Index}/{id?}	/Products	Maps to the <code>Products</code> controller and <code>Index</code> method. <code>id</code> is ignored.


Using a template is generally the simplest approach to routing. Constraints and defaults can also be specified outside the route template.

Complex segments


Complex segments are processed by matching up literal delimiters from right to left in a [non-greedy](#) way. For example, `[Route("/a{b}c{d}")]` is a complex segment. Complex segments work in a particular way that must be understood to use them successfully. The example in this section demonstrates why complex segments only really work well when the delimiter text doesn't appear inside the parameter values. Using a [regex](#) and then manually extracting the values is needed for more complex cases.

Warning

When using [System.Text.RegularExpressions](#) to process untrusted input, pass a timeout. A malicious user can provide input to [RegexExpressions](#) causing a [Denial-of-Service attack](#). ASP.NET Core framework APIs that use [RegexExpressions](#) pass a timeout.

This is a summary of the steps that routing performs with the template `/a{b}c{d}` and the URL path `/abcd`. The  is used to help visualize how the algorithm works:

- The first literal, right to left, is `c`. So `/abcd` is searched from right and finds `/ab|c|d`.
- Everything to the right (`d`) is now matched to the route parameter `{d}`.
- The next literal, right to left, is `a`. So `/ab|c|d` is searched starting where we left off, then `a` is found `/|a|b|c|d`.
- The value to the right (`b`) is now matched to the route parameter `{b}`.
- There is no remaining text and no remaining route template, so this is a match.

Here's an example of a negative case using the same template `/a{b}c{d}` and the URL path `/aabcd`. The  is used to help visualize how the algorithm works. This case isn't a

match, which is explained by the same algorithm:

- The first literal, right to left, is `c`. So `/aabcd` is searched from right and finds `/aab|c|d`.
- Everything to the right (`d`) is now matched to the route parameter `{d}`.
- The next literal, right to left, is `a`. So `/aab|c|d` is searched starting where we left off, then `a` is found `/a|a|b|c|d`.
- The value to the right (`b`) is now matched to the route parameter `{b}`.
- At this point there is remaining text `a`, but the algorithm has run out of route template to parse, so this is not a match.

Since the matching algorithm is **non-greedy**:

- It matches the smallest amount of text possible in each step.
- Any case where the delimiter value appears inside the parameter values results in not matching.

Regular expressions provide much more control over their matching behavior.

Greedy matching, also known as *maximal matching* attempts to find the longest possible match in the input text that satisfies the **regex** pattern. Non-greedy matching, also known as *lazy matching*, seeks the shortest possible match in the input text that satisfies the regex pattern.

Routing with special characters

Routing with special characters can lead to unexpected results. For example, consider a controller with the following action method:

```
[HttpGet("{id?}/name")]
public async Task<ActionResult<string>> GetName(string id)
{
    var todoItem = await _context.TODOItems.FindAsync(id);

    if (todoItem == null || todoItem.Name == null)
    {
        return NotFound();
    }

    return todoItem.Name;
}
```

When `string id` contains the following encoded values, unexpected results might occur:


 Expand table

ASCII	Encoded
<code>/</code>	<code>%2F</code>
<code> </code>	<code>+</code>

Route parameters are not always URL decoded. This problem may be addressed in the future. For more information, see [this GitHub issue](#);

Route constraints

Route constraints execute when a match has occurred to the incoming URL and the URL path is tokenized into route values. Route constraints generally inspect the route value associated via the route template and make a true or false decision about whether the value is acceptable. Some route constraints use data outside the route value to consider whether the request can be routed. For example, the `HttpMethodRouteConstraint` can accept or reject a request based on its HTTP verb. Constraints are used in routing requests and link generation.

 **Warning**

Don't use constraints for input validation. If constraints are used for input validation, invalid input results in a `404` Not Found response. Invalid input should produce a `400` Bad Request with an appropriate error message. Route constraints are used to disambiguate similar routes, not to validate the inputs for a particular route.

The following table demonstrates example route constraints and their expected behavior:

[Expand table](#)

constraint	Example	Example Matches	Notes
int	{id:int}	123456789, -123456789	Matches any integer
bool	{active:bool}	true, FALSE	Matches <code>true</code> or <code>false</code> . Case-insensitive
datetime	{dob:datetime}	2016-12-31, 2016-12-31 7:32pm	Matches a valid <code>DateTime</code> value in the invariant culture. See preceding warning.
decimal	{price:decimal}	49.99, -1,000.01	Matches a valid <code>decimal</code> value in the invariant culture. See preceding warning.
double	{weight:double}	1.234, -1,001.01e8	Matches a valid <code>double</code> value in the invariant culture. See preceding warning.
float	{weight:float}	1.234, -1,001.01e8	Matches a valid <code>float</code> value in the invariant culture. See preceding warning.
guid	{id:guid}	CD2C1638-1638- 72D5-1638- DEADBEEF1638	Matches a valid <code>Guid</code> value
long	{ticks:long}	123456789, -123456789	Matches a valid <code>long</code> value
minlength(value)	{username:minlength(4)}	Rick	String must be at least 4 characters
maxlength(value)	{filename:maxlength(8)}	MyFile	String must be no more than 8 characters
length(length)	{filename:length(12)}	somefile.txt	String must be exactly 12 characters long
length(min,max)	{filename:length(8,16)}	somefile.txt	String must be at least 8 and no more than 16 characters long
min(value)	{age:min(18)}	19	Integer value must be at least 18
max(value)	{age:max(120)}	91	Integer value must be no more than 120
range(min,max)	{age:range(18,120)}	91	Integer value must be at least 18 but no more than 120
alpha	{name:alpha}	Rick	String must consist of one or more alphabetical characters, <code>a-z</code> and case-insensitive.
regex(expression)	{ssn:regex(@"\d{{3}}-\d{{2}}-\d{{4}}\$")}	123-45-6789	String must match the regular expression. See tips about defining a regular expression.
required	{name:required}	Rick	Used to enforce that a non-parameter value is present during URL generation

Warning

When using `System.Text.RegularExpressions` to process untrusted input, pass a timeout. A malicious user can provide input to `Regex` causing a

Denial-of-Service attack [↗]. ASP.NET Core framework APIs that use `RegularExpressions` pass a timeout.

Multiple, colon delimited constraints can be applied to a single parameter. For example, the following constraint restricts a parameter to an integer value of 1 or greater:

```
[Route("users/{id:int:min(1)}")]
public User GetUserById(int id) { }
```

Warning

Route constraints that verify the URL and are converted to a CLR type always use the invariant culture. For example, conversion to the CLR type `int` or `DateTime`. These constraints assume that the URL is not localizable. The framework-provided route constraints don't modify the values stored in route values. All route values parsed from the URL are stored as strings. For example, the `float` constraint attempts to convert the route value to a float, but the converted value is used only to verify it can be converted to a float.

Regular expressions in constraints

Warning

When using `System.Text.RegularExpressions` to process untrusted input, pass a timeout. A malicious user can provide input to `RegularExpressions` causing a **Denial-of-Service attack** [↗]. ASP.NET Core framework APIs that use `RegularExpressions` pass a timeout.

Regular expressions can be specified as inline constraints using the `regex(...)` route constraint. Methods in the `MapControllerRoute` family also accept an object literal of constraints. If that form is used, string values are interpreted as regular expressions.

The following code uses an inline regex constraint:

```
app.MapGet("{message:regex(^\\d{3}-\\d{2}-\\d{4}$})",
    () => "Inline Regex Constraint Matched");
```

The following code uses an object literal to specify a regex constraint:

```
app.MapControllerRoute(
    name: "people",
    pattern: "people/{ssn}",
    constraints: new { ssn = "^\\d{3}-\\d{2}-\\d{4}$", },
    defaults: new { controller = "People", action = "List" });
```

The ASP.NET Core framework adds `RegexOptions.IgnoreCase` | `RegexOptions.Compiled` | `RegexOptions.CultureInvariant` to the regular expression constructor. See `RegexOptions` for a description of these members.

Regular expressions use delimiters and tokens similar to those used by routing and the C# language. Regular expression tokens must be escaped. To use the regular expression `^\\d{3}-\\d{2}-\\d{4}$` in an inline constraint, use one of the following:

- Replace `\` characters provided in the string as `\\` characters in the C# source file in order to escape the `\` string escape character.
- [Verbatim string literals](#).

To escape routing parameter delimiter characters `{`, `}`, `[`, `]`, double the characters in the expression, for example, `{{`, `}}`, `[[`, `]]`. The following table shows a regular expression and its escaped version:

[Expand table](#)

Regular expression	Escaped regular expression
<code>^\\d{3}-\\d{2}-\\d{4}\$</code>	<code>^\\d{3}-\\d{2}-\\d{4}\$</code>

<code>^[a-z]{2}\$</code>	<code>^[a-z]{2}\$</code>
--------------------------	--------------------------

Regular expressions used in routing often start with the `^` character and match the starting position of the string. The expressions often end with the `$` character and match the end of the string. The `^` and `$` characters ensure that the regular expression matches the entire route parameter value. Without the `^` and `$` characters, the regular expression matches any substring within the string, which is often undesirable. The following table provides examples and explains why they match or fail to match:

[Expand table](#)

Expression	String	Match	Comment
<code>[a-z]{2}</code>	hello	Yes	Substring matches
<code>[a-z]{2}</code>	123abc456	Yes	Substring matches
<code>[a-z]{2}</code>	mz	Yes	Matches expression
<code>[a-z]{2}</code>	MZ	Yes	Not case sensitive
<code>^[a-z]{2}\$</code>	hello	No	See <code>^</code> and <code>\$</code> above
<code>^[a-z]{2}\$</code>	123abc456	No	See <code>^</code> and <code>\$</code> above

For more information on regular expression syntax, see [.NET Framework Regular Expressions](#).

To constrain a parameter to a known set of possible values, use a regular expression. For example, `{action:regex(^{list|get|create}$)}` only matches the `action` route value to `list`, `get`, or `create`. If passed into the constraints dictionary, the string `^(list|get|create)$` is equivalent. Constraints that are passed in the constraints dictionary that don't match one of the known constraints are also treated as regular expressions. Constraints that are passed within a template that don't match one of the known constraints are not treated as regular expressions.

Custom route constraints

Custom route constraints can be created by implementing the `IRouteConstraint` interface. The `IRouteConstraint` interface contains `Match`, which returns `true` if the constraint is satisfied and `false` otherwise.

Custom route constraints are rarely needed. Before implementing a custom route constraint, consider alternatives, such as model binding.

The ASP.NET Core [Constraints](#) folder provides good examples of creating constraints. For example, [GuidIdRouteConstraint](#).

To use a custom `IRouteConstraint`, the route constraint type must be registered with the app's `ConstraintMap` in the service container. A `ConstraintMap` is a dictionary that maps route constraint keys to `IRouteConstraint` implementations that validate those constraints. An app's `ConstraintMap` can be updated in `Program.cs` either as part of an `AddRouting` call or by configuring `RouteOptions` directly with `builder.Services.Configure<RouteOptions>`. For example:

```
builder.Services.AddRouting(options =>
    options.ConstraintMap.Add("noZeroes", typeof(NoZeroesRouteConstraint))
```

The preceding constraint is applied in the following code:

```
[ApiController]
[Route("api/[controller]")]
public class NoZeroesController : ControllerBase
{
    [HttpGet("{id:noZeroes}")]
    public IActionResult Get(string id) =>
        Content(id);
}
```

The implementation of `NoZeroesRouteConstraint` prevents `0` being used in a route parameter:

```
public class NoZeroesRouteConstraint : IRouteConstraint
{
    private static readonly Regex _regex = new(
        @"^[1-9]*$",
        RegexOptions.CultureInvariant | RegexOptions.IgnoreCase,
        TimeSpan.FromMilliseconds(100));

    public bool Match(
        HttpContext? httpContext, IRouter? route, string routeKey,
        RouteValueDictionary values, RouteDirection routeDirection)
    {
        if (!values.TryGetValue(routeKey, out var routeValue))
        {
            return false;
        }

        var routeValueString = Convert.ToString(routeValue, CultureInfo.InvariantCulture);

        if (routeValueString is null)
        {
            return false;
        }

        return _regex.IsMatch(routeValueString);
    }
}
```

⚠ Warning

When using [System.Text.RegularExpressions](#) to process untrusted input, pass a timeout. A malicious user can provide input to [RegexExpressions](#) causing a [Denial-of-Service attack](#) [↗]. ASP.NET Core framework APIs that use [RegexExpressions](#) pass a timeout.

The preceding code:

- Prevents 0 in the {id} segment of the route.
- Is shown to provide a basic example of implementing a custom constraint. It should not be used in a production app.

The following code is a better approach to preventing an `id` containing a 0 from being processed:

```
[HttpGet("{id}")]
public IActionResult Get(string id)
{
    if (id.Contains('0'))
    {
        return StatusCode(StatusCodes.Status406NotAcceptable);
    }

    return Content(id);
}
```

The preceding code has the following advantages over the `NoZeroesRouteConstraint` approach:

- It doesn't require a custom constraint.
- It returns a more descriptive error when the route parameter includes 0.

Parameter transformers

Parameter transformers:

- Execute when generating a link using [LinkGenerator](#).
- Implement [Microsoft.AspNetCore.Routing.IOutboundParameterTransformer](#).
- Are configured using [ConstraintMap](#).
- Take the parameter's route value and transform it to a new string value.
- Result in using the transformed value in the generated link.

For example, a custom `slugify` parameter transformer in route pattern `blog\{article:slugify}` with `Url.Action(new { article = "MyTestArticle" })` generates `blog\my-test-article`.

Consider the following `IOutboundParameterTransformer` implementation:


```
public class SlugifyParameterTransformer : IOutboundParameterTransformer
{
    public string? TransformOutbound(object? value)
    {
        if (value is null)
        {
            return null;
        }

        return Regex.Replace(
            value.ToString(),
            "([a-z])([A-Z])",
            "$1-$2",
            RegexOptions.CultureInvariant,
            TimeSpan.FromMilliseconds(100))
            .ToLowerInvariant();
    }
}
```

To use a parameter transformer in a route pattern, configure it using [ConstraintMap](#) in `Program.cs`:

```
builder.Services.AddRouting(options =>
    options.ConstraintMap["slugify"] = typeof(SlugifyParameterTransformer
```

The ASP.NET Core framework uses parameter transformers to transform the URI where an endpoint resolves. For example, parameter transformers transform the route values used to match an `area`, `controller`, `action`, and `page`:

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller:slugify=Home}/{action:slugify=Index}/{id?}");
```

With the preceding route template, the action `SubscriptionManagementController.GetAll` is matched with the URI `/subscription-management/get-all`. A parameter transformer doesn't change the route values used to generate a link. For example, `Url.Action("GetAll", "SubscriptionManagement")` outputs `/subscription-management/get-all`.

ASP.NET Core provides API conventions for using parameter transformers with generated routes:

- The [Microsoft.AspNetCore.Mvc.ApplicationModels.RouteTokenTransformerConvention](#) MVC convention applies a specified parameter transformer to all attribute routes in the app. The parameter transformer transforms attribute route tokens as they are replaced. For more information, see [Use a parameter transformer to customize token replacement](#).
- Razor Pages uses the [PageRouteTransformerConvention](#) API convention. This convention applies a specified parameter transformer to all automatically discovered Razor Pages. The parameter transformer transforms the folder and file name segments of Razor Pages routes. For more information, see [Use a parameter transformer to customize page routes](#).

URL generation reference

This section contains a reference for the algorithm implemented by URL generation. In practice, most complex examples of URL generation use controllers or Razor Pages. See [routing in controllers](#) for additional information.

The URL generation process begins with a call to [LinkGenerator.GetPathByAddress](#) or a similar method. The method is provided with an address, a set of route values, and optionally information about the current request from `HttpContext`.

The first step is to use the address to resolve a set of candidate endpoints using an [IEndpointAddressScheme<TAddress>](#) that matches the address's type.

Once the set of candidates is found by the address scheme, the endpoints are ordered and processed iteratively until a URL generation operation succeeds. URL generation does **not** check for ambiguities, the first result returned is the final result.

Troubleshooting URL generation with logging

The first step in troubleshooting URL generation is setting the logging level of `Microsoft.AspNetCore.Routing` to `TRACE`. `LinkGenerator` logs many details about its processing which can be useful to troubleshoot problems.

See [URL generation reference](#) for details on URL generation.

Addresses

Addresses are the concept in URL generation used to bind a call into the link generator to a set of candidate endpoints.

Addresses are an extensible concept that come with two implementations by default:

- Using *endpoint name* (`string`) as the address:
 - Provides similar functionality to MVC's route name.
 - Uses the `EndpointNameMetadata` metadata type.
 - Resolves the provided string against the metadata of all registered endpoints.
 - Throws an exception on startup if multiple endpoints use the same name.
 - Recommended for general-purpose use outside of controllers and Razor Pages.
- Using *route values* (`RouteValuesAddress`) as the address:
 - Provides similar functionality to controllers and Razor Pages legacy URL generation.
 - Very complex to extend and debug.
 - Provides the implementation used by `UrlHelper`, Tag Helpers, HTML Helpers, Action Results, etc.

The role of the address scheme is to make the association between the address and matching endpoints by arbitrary criteria:

- The endpoint name scheme performs a basic dictionary lookup.
- The route values scheme has a complex best subset of set algorithm.

Ambient values and explicit values

From the current request, routing accesses the route values of the current request `HttpContext.Request.RouteValues`. The values associated with the current request are referred to as the **ambient values**. For the purpose of clarity, the documentation refers to the route values passed in to methods as **explicit values**.

The following example shows ambient values and explicit values. It provides ambient values from the current request and explicit values:

```
public class WidgetController : ControllerBase
{
    private readonly LinkGenerator _linkGenerator;

    public WidgetController(LinkGenerator linkGenerator) =>
        _linkGenerator = linkGenerator;

    public IActionResult Index()
    {
        var indexPath = _linkGenerator.GetPathByAction(
            HttpContext, values: new { id = 17 }!);

        return Content(indexPath);
    }

    // ...
}
```

The preceding code:

- Returns `/Widget/Index/17`
- Gets `LinkGenerator` via DI.

The following code provides only explicit values and no ambient values:

```
var subscribePath = _linkGenerator.GetPathByAction(
    "Subscribe", "Home", new { id = 17 }!);
```

The preceding method returns `/Home/Subscribe/17`

The following code in the `WidgetController` returns `/Widget/Subscribe/17`:

```
var subscribePath = _linkGenerator.GetPathByAction(
    HttpContext, "Subscribe", null, new { id = 17 });
```

The following code provides the controller from ambient values in the current request and explicit values:

```
public class GadgetController : ControllerBase
{
    public IActionResult Index() =>
        Content(Url.Action("Edit", new { id = 17 }!));
}
```

In the preceding code:

- `/Gadget/Edit/17` is returned.
- `Url` gets the `UrlHelper`.
- `Action` generates a URL with an absolute path for an action method. The URL contains the specified `action` name and `route` values.

The following code provides ambient values from the current request and explicit values:

```
public class IndexModel : PageModel
{
    public void OnGet()
    {
        var editUrl = Url.Page("./Edit", new { id = 17 });

        // ...
    }
}
```

The preceding code sets `url` to `/Edit/17` when the Edit Razor Page contains the following page directive:

```
@page "{id:int}"
```

If the Edit page doesn't contain the `"{id:int}"` route template, `url` is `/Edit?id=17`.

The behavior of MVC's `UrlHelper` adds a layer of complexity in addition to the rules described here:

- `UrlHelper` always provides the route values from the current request as ambient values.
- `UrlHelper.Action` always copies the current `action` and `controller` route values as explicit values unless overridden by the developer.
- `UrlHelper.Page` always copies the current `page` route value as an explicit value unless overridden.
- `UrlHelper.Page` always overrides the current `handler` route value with `null` as an explicit values unless overridden.

Users are often surprised by the behavioral details of ambient values, because MVC doesn't seem to follow its own rules. For historical and compatibility reasons, certain route values such as `action`, `controller`, `page`, and `handler` have their own special-case behavior.

The equivalent functionality provided by `LinkGenerator.GetPathByAction` and `LinkGenerator.GetPathByPage` duplicates these anomalies of `UrlHelper` for compatibility.

URL generation process

Once the set of candidate endpoints are found, the URL generation algorithm:

- Processes the endpoints iteratively.
- Returns the first successful result.

The first step in this process is called **route value invalidation**. Route value invalidation is the process by which routing decides which route values from the ambient values should be used and which should be ignored. Each ambient value is considered and either combined with the explicit values, or ignored.

The best way to think about the role of ambient values is that they attempt to save application developers typing, in some common cases. Traditionally, the scenarios where ambient values are helpful are related to MVC:

- When linking to another action in the same controller, the controller name doesn't need to be specified.
- When linking to another controller in the same area, the area name doesn't need to be specified.
- When linking to the same action method, route values don't need to be specified.
- When linking to another part of the app, you don't want to carry over route values that have no meaning in that part of the app.

Calls to `LinkGenerator` or `UrlHelper` that return `null` are usually caused by not understanding route value invalidation. Troubleshoot route value invalidation by explicitly specifying more of the route values to see if that solves the problem.

Route value invalidation works on the assumption that the app's URL scheme is hierarchical, with a hierarchy formed from left-to-right. Consider the basic controller route template `{controller}/{action}/{id?}` to get an intuitive sense of how this works in practice. A **change** to a value **invalidates** all of the route values that appear to the right. This reflects the assumption about hierarchy. If the app has an ambient value for `id`, and the operation specifies a different value for the `controller`:

- `id` won't be reused because `{controller}` is to the left of `{id?}`.

Some examples demonstrating this principle:

- If the explicit values contain a value for `id`, the ambient value for `id` is ignored. The ambient values for `controller` and `action` can be used.
- If the explicit values contain a value for `action`, any ambient value for `action` is ignored. The ambient values for `controller` can be used. If the explicit value for `action` is different from the ambient value for `action`, the `id` value won't be used. If the explicit value for `action` is the same as the ambient value for `action`, the `id` value can be used.
- If the explicit values contain a value for `controller`, any ambient value for `controller` is ignored. If the explicit value for `controller` is different from the ambient value for `controller`, the `action` and `id` values won't be used. If the explicit value for `controller` is the same as the ambient value for `controller`, the `action` and `id` values can be used.

This process is further complicated by the existence of attribute routes and dedicated conventional routes. Controller conventional routes such as `{controller}/{action}/{id?}` specify a hierarchy using route parameters. For [dedicated conventional routes](#) and [attribute routes](#) to controllers and Razor Pages:

- There is a hierarchy of route values.
- They don't appear in the template.

For these cases, URL generation defines the **required values** concept. Endpoints created by controllers and Razor Pages have required values specified that allow route value invalidation to work.

The route value invalidation algorithm in detail:

- The required value names are combined with the route parameters, then processed from left-to-right.
- For each parameter, the ambient value and explicit value are compared:
 - If the ambient value and explicit value are the same, the process continues.
 - If the ambient value is present and the explicit value isn't, the ambient value is used when generating the URL.
 - If the ambient value isn't present and the explicit value is, reject the ambient value and all subsequent ambient values.
 - If the ambient value and the explicit value are present, and the two values are different, reject the ambient value and all subsequent ambient values.

At this point, the URL generation operation is ready to evaluate route constraints. The set of accepted values is combined with the parameter default values, which is provided to constraints. If the constraints all pass, the operation continues.