Routing to controller actions in ASP.NET Core

03/25/2020 • 68 minutes to read • +15

In this article

By Ryan Nowak, Kirk Larkin, and Rick Anderson

ASP.NET Core controllers use the Routing middleware to match the URLs of incoming requests and map them to actions. Routes templates:

- Are defined in startup code or attributes.
- Describe how URL paths are matched to actions.
- Are used to generate URLs for links. The generated links are typically returned in responses.

Actions are either conventionally-routed or attribute-routed. Placing a route on the controller or action makes it attribute-routed. See Mixed routing for more information.

This document:

- Explains the interactions between MVC and routing:
  - How typical MVC apps make use of routing features.
  - Covers both:
    - Conventionally routing typically used with controllers and views.
    - *Attribute routing* used with REST APIs. If you're primarily interested in routing for REST APIs, jump to the Attribute routing for REST APIs section.
  - See Routing for advanced routing details.
- Refers to the default routing system added in ASP.NET Core 3.0, called endpoint routing. It's possible to use controllers with the previous version of routing for compatibility purposes. See the 2.2-3.0 migration guide for instructions. Refer to the 2.2 version of this document for reference material on the legacy routing system.

Set up conventional route

`Startup.Configure` typically has code similar to the following when using conventional routing:

C# Copy

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

Inside the call to UseEndpoints, MapControllerRoute is used to create a single route. The single route is named `default` route. Most apps with controllers and views use a route template similar to the `default` route. REST APIs should use attribute

routing.

The route template `"{controller=Home}/{action=Index}/{id?}"`:

- Matches a URL path like `/Products/Details/5`
- Extracts the route values `{ controller = Products, action = Details, id = 5 }` by tokenizing the path. The extraction of route values results in a match if the app has a controller named `ProductsController` and a `Details` action:

  C# Copy

  ```
  public class ProductsController : Controller
  {
      public IActionResult Details(int id)
      {
          return ControllerContext.MyDisplayRouteInfo(id);
      }
  }
  ```

  MyDisplayRouteInfo is provided by the Rick.Docs.Samples.RouteInfo NuGet package and displays route information.

- `/Products/Details/5` model binds the value of `id = 5` to set the `id` parameter to `5`. See Model Binding for more details.
- `{controller=Home}` defines `Home` as the default `controller`.
- `{action=Index}` defines `Index` as the default `action`.
- The `?` character in `{id?}` defines `id` as optional.
- Default and optional route parameters don't need to be present in the URL path for a match. See Route Template Reference for a detailed description of route template syntax.

- Matches the URL path `/`.
- Produces the route values `{ controller = Home, action = Index }`.

The values for `controller` and `action` make use of the default values. `id` doesn't produce a value since there's no corresponding segment in the URL path. `/` only matches if there exists a `HomeController` and `Index` action:

C# Copy

```
public class HomeController : Controller
{
  public IActionResult Index() { ... }
}
```

Using the preceding controller definition and route template, the `HomeController.Index` action is run for the following URL paths:

- `/Home/Index/17`
- `/Home/Index`
- `/Home`
- `/`

The URL path `/` uses the route template default `Home` controllers and `Index` action. The URL path `/Home` uses the route template default `Index` action.

The convenience method MapDefaultControllerRoute:

C# Copy

```
endpoints.MapDefaultControllerRoute();
```

Replaces:

C# Copy

```
endpoints.MapControllerRoute("default", "{controller=Home}/{action=Index}/{id?}");
```

Important

Routing is configured using the **UseRouting** and **UseEndpoints** middleware. To use controllers:

- Call **MapControllers** inside `UseEndpoints` to map **attribute routed** controllers.
- Call **MapControllerRoute** or **MapAreaControllerRoute**, to map both **conventionally routed** controllers and **attribute routed** controllers.

Conventional routing

Conventional routing is used with controllers and views. The `default` route:

C# Copy

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

is an example of a *conventional routing*. It's called *conventional routing* because it establishes a *convention* for URL paths:

- The first path segment, `{controller=Home}`, maps to the controller name.
- The second segment, `{action=Index}`, maps to the action name.

- The third segment, `{id?}` is used for an optional `id`. The `?` in `{id?}` makes it optional. `id` is used to map to a model entity.

Using this `default` route, the URL path:

- `/Products/List` maps to the `ProductsController.List` action.
- `/Blog/Article/17` maps to `BlogController.Article` and typically model binds the `id` parameter to 17.

This mapping:

- Is based on the controller and action names **only**.
- Isn't based on namespaces, source file locations, or method parameters.

Using conventional routing with the default route allows creating the app without having to come up with a new URL pattern for each action. For an app with CRUD style actions, having consistency for the URLs across controllers:

- Helps simplify the code.
- Makes the UI more predictable.

> ⚠ **Warning**
>
> The `id` in the preceding code is defined as optional by the route template. Actions can execute without the optional ID provided as part of the URL. Generally, when `id` is omitted from the URL:
>
> - `id` is set to `0` by model binding.
> - No entity is found in the database matching `id == 0`.
>
> **Attribute routing** provides fine-grained control to make the ID required for some actions and not for others. By convention, the documentation includes optional parameters like `id` when they're likely to appear in correct usage.

Most apps should choose a basic and descriptive routing scheme so that URLs are readable and meaningful. The default conventional route `{controller=Home}/{action=Index}/{id?}`:

- Supports a basic and descriptive routing scheme.
- Is a useful starting point for UI-based apps.
- Is the only route template needed for many web UI apps. For larger web UI apps, another route using Areas is frequently all that's needed.

MapControllerRoute and MapAreaRoute :

- Automatically assign an **order** value to their endpoints based on the order they are invoked.

Endpoint routing in ASP.NET Core 3.0 and later:

- Doesn't have a concept of routes.
- Doesn't provide ordering guarantees for the execution of extensibility, all endpoints are processed at once.

Enable Logging to see how the built-in routing implementations, such as Route, match requests.

Attribute routing is explained later in this document.

Multiple conventional routes

Multiple conventional routes can be added inside `UseEndpoints` by adding more calls to MapControllerRoute and MapAreaControllerRoute. Doing so allows defining multiple conventions, or to adding conventional routes that are dedicated to a specific action, such as:

C# Copy

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(name: "blog",
                 pattern: "blog/{*article}",
```

```
                    defaults: new { controller = "Blog", action = "Article" });
    endpoints.MapControllerRoute(name: "default",
                    pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

The `blog` route in the preceding code is a **dedicated conventional route**. It's called a dedicated conventional route because:

- It uses conventional routing.
- It's dedicated to a specific action.

Because `controller` and `action` don't appear in the route template `"blog/{*article}"` as parameters:

- They can only have the default values `{ controller = "Blog", action = "Article" }`.
- This route always maps to the action `BlogController.Article`.

`/Blog`, `/Blog/Article`, and `/Blog/{any-string}` are the only URL paths that match the blog route.

The preceding example:

- `blog` route has a higher priority for matches than the `default` route because it is added first.
- Is an example of Slug style routing where it's typical to have an article name as part of the URL.

> ⚠ **Warning**
>
> In ASP.NET Core 3.0 and later, routing doesn't:
>
> - Define a concept called a *route*. `UseRouting` adds route matching to the middleware pipeline. The `UseRouting` middleware looks at the set of endpoints defined in the app, and selects the best endpoint match based on the request.
> - Provide guarantees about the execution order of extensibility like **IRouteConstraint** or **IActionConstraint**.
>
> See **Routing** for reference material on routing.

Conventional routing order

Conventional routing only matches a combination of action and controller that are defined by the app. This is intended to simplify cases where conventional routes overlap. Adding routes using MapControllerRoute, MapDefaultControllerRoute, and MapAreaControllerRoute automatically assign an order value to their endpoints based on the order they are invoked. Matches from a route that appears earlier have a higher priority. Conventional routing is order-dependent. In general, routes with areas should be placed earlier as they're more specific than routes without an area. Dedicated conventional routes with catch-all route parameters like `{*article}` can make a route too greedy, meaning that it matches URLs that you intended to be matched by other routes. Put the greedy routes later in the route table to prevent greedy matches.

Resolving ambiguous actions

When two endpoints match through routing, routing must do one of the following:

- Choose the best candidate.
- Throw an exception.

For example:

C# Copy

```
public class Products33Controller : Controller
{
    public IActionResult Edit(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }

    [HttpPost]
    public IActionResult Edit(int id, Product product)
    {
```

```
            return ControllerContext.MyDisplayRouteInfo(id, product.name);
        }
    }
}
```

The preceding controller defines two actions that match:

- The URL path `/Products33/Edit/17`
- Route data `{ controller = Products33, action = Edit, id = 17 }`.

This is a typical pattern for MVC controllers:

- `Edit(int)` displays a form to edit a product.
- `Edit(int, Product)` processes the posted form.

To resolve the correct route:

- `Edit(int, Product)` is selected when the request is an HTTP `POST`.
- `Edit(int)` is selected when the HTTP verb is anything else. `Edit(int)` is generally called via `GET`.

The HttpPostAttribute, `[HttpPost]`, is provided to routing so that it can choose based on the HTTP method of the request. The `HttpPostAttribute` makes `Edit(int, Product)` a better match than `Edit(int)`.

It's important to understand the role of attributes like `HttpPostAttribute`. Similar attributes are defined for other HTTP verbs. In conventional routing, it's common for actions to use the same action name when they're part of a show form, submit form workflow. For example, see Examine the two Edit action methods.

If routing can't choose a best candidate, an AmbiguousMatchException is thrown, listing the multiple matched endpoints.

Conventional route names

The strings `"blog"` and `"default"` in the following examples are conventional route names:

C# Copy

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(name: "blog",
                pattern: "blog/{*article}",
                defaults: new { controller = "Blog", action = "Article" });
    endpoints.MapControllerRoute(name: "default",
                pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

The route names give the route a logical name. The named route can be used for URL generation. Using a named route simplifies URL creation when the ordering of routes could make URL generation complicated. Route names must be unique application wide.

Route names:

- Have no impact on URL matching or handling of requests.
- Are used only for URL generation.

The route name concept is represented in routing as IEndpointNameMetadata. The terms **route name** and **endpoint name**:

- Are interchangeable.
- Which one is used in documentation and code depends on the API being described.

Attribute routing for REST APIs

REST APIs should use attribute routing to model the app's functionality as a set of resources where operations are represented by HTTP verbs.

Attribute routing uses a set of attributes to map actions directly to route templates. The following `StartUp.Configure` code is typical for a REST API and is used in the next sample:

C# Copy

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseHttpsRedirection();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

In the preceding code, MapControllers is called inside `UseEndpoints` to map attribute routed controllers.

In the following example:

- The preceding `Configure` method is used.

- `HomeController` matches a set of URLs similar to what the default conventional route `{controller=Home}/{action=Index}/{id?}` matches.

C# Copy

```
public class HomeController : Controller
{
    [Route("")]
    [Route("Home")]
    [Route("Home/Index")]
    [Route("Home/Index/{id?}")]
    public IActionResult Index(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }

    [Route("Home/About")]
    [Route("Home/About/{id?}")]
    public IActionResult About(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

The `HomeController.Index` action is run for any of the URL paths `/`, `/Home`, `/Home/Index`, or `/Home/Index/3`.

This example highlights a key programming difference between attribute routing and conventional routing. Attribute routing requires more input to specify a route. The conventional default route handles routes more succinctly. However, attribute routing allows and requires precise control of which route templates apply to each action.

With attribute routing, the controller and action names play no part in which action is matched, unless token replacement is used. The following example matches the same URLs as the previous example:

C# Copy

```
public class MyDemoController : Controller
{
    [Route("")]
    [Route("Home")]
    [Route("Home/Index")]
    [Route("Home/Index/{id?}")]
    public IActionResult MyIndex(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }

    [Route("Home/About")]
    [Route("Home/About/{id?}")]
    public IActionResult MyAbout(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

The following code uses token replacement for `action` and `controller`:

C# Copy

```
public class HomeController : Controller
{
    [Route("")]
    [Route("Home")]
    [Route("[controller]/[action]")]
    public IActionResult Index()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    [Route("[controller]/[action]")]
    public IActionResult About()
    {
        return ControllerContext.MyDisplayRouteInfo();
```

```
    }
}
```

The following code applies `[Route("[controller]/[action]")]` to the controller:

C# Copy

```
[Route("[controller]/[action]")]
public class HomeController : Controller
{
    [Route("~/")]
    [Route("/Home")]
    [Route("~/Home/Index")]
    public IActionResult Index()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    public IActionResult About()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

In the preceding code, the `Index` method templates must prepend `/` or `~/` to the route templates. Route templates applied to an action that begin with `/` or `~/` don't get combined with route templates applied to the controller.

See Route template precedence for information on route template selection.

Reserved routing names

The following keywords are reserved route parameter names when using Controllers or Razor Pages:

- `action`
- `area`
- `controller`
- `handler`
- `page`

Using `page` as a route parameter with attribute routing is a common error. Doing that results in inconsistent and confusing behavior with URL generation.

C# Copy

```
public class MyDemo2Controller : Controller
{
    [Route("/articles/{page}")]
    public IActionResult ListArticles(int page)
    {
        return ControllerContext.MyDisplayRouteInfo(page);
    }
}
```

The special parameter names are used by the URL generation to determine if a URL generation operation refers to a Razor Page or to a Controller.

HTTP verb templates

ASP.NET Core has the following HTTP verb templates:

- [HttpGet]
- [HttpPost]
- [HttpPut]
- [HttpDelete]

- [HttpHead]
- [HttpPatch]

Route templates

ASP.NET Core has the following route templates:

- All the HTTP verb templates are route templates.
- [Route]

Attribute routing with Http verb attributes

Consider the following controller:

C# Copy

```
[Route("api/[controller]")]
[ApiController]
public class Test2Controller : ControllerBase
{
    [HttpGet]   // GET /api/test2
    public IActionResult ListProducts()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    [HttpGet("{id}")]   // GET /api/test2/xyz
    public IActionResult GetProduct(string id)
    {
       return ControllerContext.MyDisplayRouteInfo(id);
    }

    [HttpGet("int/{id:int}")] // GET /api/test2/int/3
    public IActionResult GetIntProduct(int id)
```

```
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }

    [HttpGet("int2/{id}")]  // GET /api/test2/int2/3
    public IActionResult GetInt2Product(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

In the preceding code:

- Each action contains the `[HttpGet]` attribute, which constrains matching to HTTP GET requests only.
- The `GetProduct` action includes the `"{id}"` template, therefore `id` is appended to the `"api/[controller]"` template on the controller. The methods template is `"api/[controller]/"{id}""`. Therefore this action only matches GET requests for the form `/api/test2/xyz`,`/api/test2/123`,`/api/test2/{any string}`, etc.

  C# Copy

  ```
  [HttpGet("{id}")]   // GET /api/test2/xyz
  public IActionResult GetProduct(string id)
  {
     return ControllerContext.MyDisplayRouteInfo(id);
  }
  ```

- The `GetIntProduct` action contains the `"int/{id:int}")` template. The `:int` portion of the template constrains the `id` route values to strings that can be converted to an integer. A GET request to `/api/test2/int/abc`:
  - Doesn't match this action.
  - Returns a 404 Not Found error.

  C# Copy

```
[HttpGet("int/{id:int}")] // GET /api/test2/int/3
public IActionResult GetIntProduct(int id)
{
    return ControllerContext.MyDisplayRouteInfo(id);
}
```

- The `GetInt2Product` action contains `{id}` in the template, but doesn't constrain `id` to values that can be converted to an integer. A GET request to `/api/test2/int2/abc`:
  - Matches this route.
  - Model binding fails to convert `abc` to an integer. The `id` parameter of the method is integer.
  - Returns a 400 Bad Request because model binding failed to convert `abc` to an integer.

C# Copy

```
[HttpGet("int2/{id}")]  // GET /api/test2/int2/3
public IActionResult GetInt2Product(int id)
{
    return ControllerContext.MyDisplayRouteInfo(id);
}
```

Attribute routing can use HttpMethodAttribute attributes such as HttpPostAttribute, HttpPutAttribute, and HttpDeleteAttribute. All of the HTTP verb attributes accept a route template. The following example shows two actions that match the same route template:

C# Copy

```
[ApiController]
public class MyProductsController : ControllerBase
{
    [HttpGet("/products3")]
    public IActionResult ListProducts()
    {
        return ControllerContext.MyDisplayRouteInfo();
```

```
    }

    [HttpPost("/products3")]
    public IActionResult CreateProduct(MyProduct myProduct)
    {
        return ControllerContext.MyDisplayRouteInfo(myProduct.Name);
    }
}
```

Using the URL path `/products3`:

- The `MyProductsController.ListProducts` action runs when the HTTP verb is `GET`.
- The `MyProductsController.CreateProduct` action runs when the HTTP verb is `POST`.

When building a REST API, it's rare that you'll need to use `[Route(...)]` on an action method because the action accepts all HTTP methods. It's better to use the more specific HTTP verb attribute to be precise about what your API supports. Clients of REST APIs are expected to know what paths and HTTP verbs map to specific logical operations.

REST APIs should use attribute routing to model the app's functionality as a set of resources where operations are represented by HTTP verbs. This means that many operations, for example, GET and POST on the same logical resource use the same URL. Attribute routing provides a level of control that's needed to carefully design an API's public endpoint layout.

Since an attribute route applies to a specific action, it's easy to make parameters required as part of the route template definition. In the following example, `id` is required as part of the URL path:

C# Copy

```
[ApiController]
public class Products2ApiController : ControllerBase
{
    [HttpGet("/products2/{id}", Name = "Products_List")]
    public IActionResult GetProduct(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
```

```
    }
}
```

The `Products2ApiController.GetProduct(int)` action:

- Is run with URL path like `/products2/3`
- Isn't run with the URL path `/products2`.

The [Consumes] attribute allows an action to limit the supported request content types. For more information, see Define supported request content types with the Consumes attribute.

See Routing for a full description of route templates and related options.

For more information on `[ApiController]`, see ApiController attribute.

Route name

The following code defines a route name of `Products_List`:

C# Copy

```
[ApiController]
public class Products2ApiController : ControllerBase
{
    [HttpGet("/products2/{id}", Name = "Products_List")]
    public IActionResult GetProduct(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

Route names can be used to generate a URL based on a specific route. Route names:

- Have no impact on the URL matching behavior of routing.
- Are only used for URL generation.

Route names must be unique application-wide.

Contrast the preceding code with the conventional default route, which defines the `id` parameter as optional (`{id?}`). The ability to precisely specify APIs has advantages, such as allowing `/products` and `/products/5` to be dispatched to different actions.

Combining attribute routes

To make attribute routing less repetitive, route attributes on the controller are combined with route attributes on the individual actions. Any route templates defined on the controller are prepended to route templates on the actions. Placing a route attribute on the controller makes **all** actions in the controller use attribute routing.

C# Copy

```
[ApiController]
[Route("products")]
public class ProductsApiController : ControllerBase
{
    [HttpGet]
    public IActionResult ListProducts()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    [HttpGet("{id}")]
    public IActionResult GetProduct(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

In the preceding example:

- The URL path `/products` can match `ProductsApi.ListProducts`
- The URL path `/products/5` can match `ProductsApi.GetProduct(int)`.

Both of these actions only match HTTP `GET` because they're marked with the `[HttpGet]` attribute.

Route templates applied to an action that begin with `/` or `~/` don't get combined with route templates applied to the controller. The following example matches a set of URL paths similar to the default route.

C# Copy

```
[Route("Home")]
public class HomeController : Controller
{
    [Route("")]
    [Route("Index")]
    [Route("/")]
    public IActionResult Index()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    [Route("About")]
    public IActionResult About()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

The following table explains the `[Route]` attributes in the preceding code:

Attribute	Combines with `[Route("Home")]`	Defines route template

Attribute	Combines with `[Route("Home")]`	Defines route template
`[Route("")]`	Yes	`"Home"`
`[Route("Index")]`	Yes	`"Home/Index"`
`[Route("/")]`	**No**	`""`
`[Route("About")]`	Yes	`"Home/About"`

Attribute route order

Routing builds a tree and matches all endpoints simultaneously:

- The route entries behave as if placed in an ideal ordering.
- The most specific routes have a chance to execute before the more general routes.

For example, an attribute route like `blog/search/{topic}` is more specific than an attribute route like `blog/{*article}`. The `blog/search/{topic}` route has higher priority, by default, because it's more specific. Using conventional routing, the developer is responsible for placing routes in the desired order.

Attribute routes can configure an order using the Order property. All of the framework provided route attributes include `Order` . Routes are processed according to an ascending sort of the `Order` property. The default order is `0`. Setting a route using `Order = -1` runs before routes that don't set an order. Setting a route using `Order = 1` runs after default route ordering.

Avoid depending on `Order`. If an app's URL-space requires explicit order values to route correctly, then it's likely confusing to clients as well. In general, attribute routing selects the correct route with URL matching. If the default order used for URL generation isn't working, using a route name as an override is usually simpler than applying the `Order` property.

Consider the following two controllers which both define the route matching `/home`:

C# Copy

```
public class HomeController : Controller
{
    [Route("")]
    [Route("Home")]
    [Route("Home/Index")]
    [Route("Home/Index/{id?}")]
    public IActionResult Index(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }

    [Route("Home/About")]
    [Route("Home/About/{id?}")]
    public IActionResult About(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

C# Copy

```
public class MyDemoController : Controller
{
    [Route("")]
    [Route("Home")]
    [Route("Home/Index")]
    [Route("Home/Index/{id?}")]
    public IActionResult MyIndex(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
```

```
    [Route("Home/About")]
    [Route("Home/About/{id?}")]
    public IActionResult MyAbout(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

Requesting `/home` with the preceding code throws an exception similar to the following:

text Copy

```
AmbiguousMatchException: The request matched multiple endpoints. Matches:

 WebMvcRouting.Controllers.HomeController.Index
 WebMvcRouting.Controllers.MyDemoController.MyIndex
```

Adding `Order` to one of the route attributes resolves the ambiguity:

C# Copy

```
[Route("")]
[Route("Home", Order = 2)]
[Route("Home/MyIndex")]
public IActionResult MyIndex()
{
    return ControllerContext.MyDisplayRouteInfo();
}
```

With the preceding code, `/home` runs the `HomeController.Index` endpoint. To get to the `MyDemoController.MyIndex`, request `/home/MyIndex`. **Note**:

- The preceding code is an example or poor routing design. It was used to illustrate the `Order` property.

- The `Order` property only resolves the ambiguity, that template cannot be matched. It would be better to remove the `[Route("Home")]` template.

See Razor Pages route and app conventions: Route order for information on route order with Razor Pages.

In some cases, an HTTP 500 error is returned with ambiguous routes. Use logging to see which endpoints caused the `AmbiguousMatchException`.

Token replacement in route templates [controller], [action], [area]

For convenience, attribute routes support token replacement for reserved route parameters by enclosing a token in one of the following:

- Square brackets: `[]`
- Curly braces: `{}`

The tokens `[action]`, `[area]`, and `[controller]` are replaced with the values of the action name, area name, and controller name from the action where the route is defined:

C# Copy

```
[Route("[controller]/[action]")]
public class Products0Controller : Controller
{
    [HttpGet]
    public IActionResult List()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
```

```
    [HttpGet("{id}")]
    public IActionResult Edit(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

In the preceding code:

C# Copy

```
[HttpGet]
public IActionResult List()
{
    return ControllerContext.MyDisplayRouteInfo();
}
```

- Matches `/Products0/List`

C# Copy

```
[HttpGet("{id}")]
public IActionResult Edit(int id)
{
    return ControllerContext.MyDisplayRouteInfo(id);
}
```

- Matches `/Products0/Edit/{id}`

Token replacement occurs as the last step of building the attribute routes. The preceding example behaves the same as the following code:

C# Copy

```
public class Products20Controller : Controller
{
    [HttpGet("[controller]/[action]")]  // Matches '/Products20/List'
    public IActionResult List()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    [HttpGet("[controller]/[action]/{id}")]   // Matches '/Products20/Edit/{id}'
    public IActionResult Edit(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

If you are reading this in a language other than English, let us know in this GitHub discussion issue if you'd like to see the code comments in your native language.

Attribute routes can also be combined with inheritance. This is powerful combined with token replacement. Token replacement also applies to route names defined by attribute routes. `[Route("[controller]/[action]", Name="[controller]_[action]")]` generates a unique route name for each action:

C# Copy

```
[ApiController]
[Route("api/[controller]/[action]", Name = "[controller]_[action]")]
public abstract class MyBase2Controller : ControllerBase
{
}

public class Products11Controller : MyBase2Controller
{
    [HttpGet]                       // /api/products11/
    public IActionResult List()
    {
        return ControllerContext.MyDisplayRouteInfo();
```

```
    }

    [HttpGet("{id}")]             //    /api/products11/edit/3
    public IActionResult Edit(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

Token replacement also applies to route names defined by attribute routes. `[Route("[controller]/[action]", Name="[controller]_[action]")]` generates a unique route name for each action.

To match the literal token replacement delimiter `[` or `]`, escape it by repeating the character (`[[` or `]]`).

Use a parameter transformer to customize token replacement

Token replacement can be customized using a parameter transformer. A parameter transformer implements IOutboundParameterTransformer and transforms the value of parameters. For example, a custom `SlugifyParameterTransformer` parameter transformer changes the `SubscriptionManagement` route value to `subscription-management`:

C# Copy

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

        return Regex.Replace(value.ToString(),
                             "([a-z])([A-Z])",
                             "$1-$2",
                             RegexOptions.CultureInvariant,
```

```
                                TimeSpan.FromMilliseconds(100)).ToLowerInvariant();
    }
}
```

The RouteTokenTransformerConvention is an application model convention that:

- Applies a parameter transformer to all attribute routes in an application.
- Customizes the attribute route token values as they are replaced.

C# Copy

```
public class SubscriptionManagementController : Controller
{
    [HttpGet("[controller]/[action]")]
    public IActionResult ListAll()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

The preceding `ListAll` method matches `/subscription-management/list-all`.

The `RouteTokenTransformerConvention` is registered as an option in `ConfigureServices`.

C# Copy

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews(options =>
    {
        options.Conventions.Add(new RouteTokenTransformerConvention(
                                    new SlugifyParameterTransformer()));
    });
}
```

See MDN web docs on Slug for the definition of Slug.

> ⚠ **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.

Multiple attribute routes

Attribute routing supports defining multiple routes that reach the same action. The most common usage of this is to mimic the behavior of the default conventional route as shown in the following example:

C# Copy

```
[Route("[controller]")]
public class Products13Controller : Controller
{
    [Route("")]     // Matches 'Products13'
    [Route("Index")] // Matches 'Products13/Index'
    public IActionResult Index()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
```

Putting multiple route attributes on the controller means that each one combines with each of the route attributes on the action methods:

C# Copy

```
[Route("Store")]
[Route("[controller]")]
public class Products6Controller : Controller
{
    [HttpPost("Buy")]       // Matches 'Products6/Buy' and 'Store/Buy'
    [HttpPost("Checkout")]  // Matches 'Products6/Checkout' and 'Store/Checkout'
    public IActionResult Buy()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

All the HTTP verb route constraints implement `IActionConstraint`.

When multiple route attributes that implement IActionConstraint are placed on an action:

- Each action constraint combines with the route template applied to the controller.

C# Copy

```
[Route("api/[controller]")]
public class Products7Controller : ControllerBase
{
    [HttpPut("Buy")]        // Matches PUT 'api/Products7/Buy'
    [HttpPost("Checkout")]  // Matches POST 'api/Products7/Checkout'
    public IActionResult Buy()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

Using multiple routes on actions might seem useful and powerful, it's better to keep your app's URL space basic and well defined. Use multiple routes on actions **only** where needed, for example, to support existing clients.

Specifying attribute route optional parameters, default values, and constraints

Attribute routes support the same inline syntax as conventional routes to specify optional parameters, default values, and constraints.

C# Copy

```
public class Products14Controller : Controller
{
    [HttpPost("product14/{id:int}")]
    public IActionResult ShowProduct(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

In the preceding code, `[HttpPost("product/{id:int}")]` applies a route constraint. The `ProductsController.ShowProduct` action is matched only by URL paths like `/product/3`. The route template portion `{id:int}` constrains that segment to only integers.

See Route Template Reference for a detailed description of route template syntax.

Custom route attributes using IRouteTemplateProvider

All of the route attributes implement IRouteTemplateProvider. The ASP.NET Core runtime:

- Looks for attributes on controller classes and action methods when the app starts.
- Uses the attributes that implement `IRouteTemplateProvider` to build the initial set of routes.

Implement `IRouteTemplateProvider` to define custom route attributes. Each `IRouteTemplateProvider` allows you to define a single route with a custom route template, order, and name:

C# Copy

```
public class MyApiControllerAttribute : Attribute, IRouteTemplateProvider
{
    public string Template => "api/[controller]";
    public int? Order => 2;
    public string Name { get; set; }
}

[MyApiController]
[ApiController]
public class MyTestApiController : ControllerBase
{
    // GET /api/MyTestApi
    [HttpGet]
    public IActionResult Get()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

The preceding `Get` method returns `Order = 2, Template = api/MyTestApi`.

Use application model to customize attribute routes

The application model:

- Is an object model created at startup.
- Contains all of the metadata used by ASP.NET Core to route and execute the actions in an app.

The application model includes all of the data gathered from route attributes. The data from route attributes is provided by the `IRouteTemplateProvider` implementation. Conventions:

- Can be written to modify the application model to customize how routing behaves.
- Are read at app startup.

This section shows a basic example of customizing routing using application model. The following code makes routes roughly line up with the folder structure of the project.

C# Copy

```
public class NamespaceRoutingConvention : Attribute, IControllerModelConvention
{
    private readonly string _baseNamespace;

    public NamespaceRoutingConvention(string baseNamespace)
    {
        _baseNamespace = baseNamespace;
    }

    public void Apply(ControllerModel controller)
    {
        var hasRouteAttributes = controller.Selectors.Any(selector =>
                                                 selector.AttributeRouteModel != null);
        if (hasRouteAttributes)
        {
            return;
        }

        var namespc = controller.ControllerType.Namespace;
        if (namespc == null)
            return;
        var template = new StringBuilder();
        template.Append(namespc, _baseNamespace.Length + 1,
                        namespc.Length - _baseNamespace.Length - 1);
        template.Replace('.', '/');
```

```
        template.Append("/[controller]/[action]/{id?}");

        foreach (var selector in controller.Selectors)
        {
            selector.AttributeRouteModel = new AttributeRouteModel()
            {
                Template = template.ToString()
            };
        }
    }
}
```

The following code prevents the `namespace` convention from being applied to controllers that are attribute routed:

C# Copy

```
public void Apply(ControllerModel controller)
{
    var hasRouteAttributes = controller.Selectors.Any(selector =>
                                            selector.AttributeRouteModel != null);
    if (hasRouteAttributes)
    {
        return;
    }
```

For example, the following controller doesn't use `NamespaceRoutingConvention`:

C# Copy

```
[Route("[controller]/[action]/{id?}")]
public class ManagersController : Controller
{
    // /managers/index
    public IActionResult Index()
    {
```

```
        var template = ControllerContext.ActionDescriptor.AttributeRouteInfo?.Template;
        return Content($"Index- template:{template}");
    }

    public IActionResult List(int? id)
    {
        var path = Request.Path.Value;
        return Content($"List- Path:{path}");
    }
}
```

The `NamespaceRoutingConvention.Apply` method:

- Does nothing if the controller is attribute routed.
- Sets the controllers template based on the `namespace`, with the base `namespace` removed.

The `NamespaceRoutingConvention` can be applied in `Startup.ConfigureServices`:

C# Copy

```
namespace My.Application
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
```

```
            services.AddControllersWithViews(options =>
            {
                options.Conventions.Add(
                    new NamespaceRoutingConvention(typeof(Startup).Namespace));
            });
        }
        // Remaining code ommitted for brevity.
```

For example, consider the following controller:

C# Copy

```
using Microsoft.AspNetCore.Mvc;

namespace My.Application.Admin.Controllers
{
    public class UsersController : Controller
    {
        // GET /admin/controllers/users/index
        public IActionResult Index()
        {
            var fullname = typeof(UsersController).FullName;
            var template =
                ControllerContext.ActionDescriptor.AttributeRouteInfo?.Template;
            var path = Request.Path.Value;

            return Content($"Path: {path} fullname: {fullname}  template:{template}");
        }

        public IActionResult List(int? id)
        {
            var path = Request.Path.Value;
            return Content($"Path: {path} ID:{id}");
        }
    }
}
```

In the preceding code:

- The base `namespace` is `My.Application`.
- The full name of the preceding controller is `My.Application.Admin.Controllers.UsersController`.
- The `NamespaceRoutingConvention` sets the controllers template to `Admin/Controllers/Users/[action]/{id?`.

The `NamespaceRoutingConvention` can also be applied as an attribute on a controller:

C# Copy

```
[NamespaceRoutingConvention("My.Application")]
public class TestController : Controller
{
    // /admin/controllers/test/index
    public IActionResult Index()
    {
        var template = ControllerContext.ActionDescriptor.AttributeRouteInfo?.Template;
        var actionname = ControllerContext.ActionDescriptor.ActionName;
        return Content($"Action- {actionname} template:{template}");
    }

    public IActionResult List(int? id)
    {
        var path = Request.Path.Value;
        return Content($"List- Path:{path}");
    }
}
```

Mixed routing: Attribute routing vs conventional routing

ASP.NET Core apps can mix the use of conventional routing and attribute routing. It's typical to use conventional routes for controllers serving HTML pages for browsers, and attribute routing for controllers serving REST APIs.

Actions are either conventionally routed or attribute routed. Placing a route on the controller or the action makes it attribute routed. Actions that define attribute routes cannot be reached through the conventional routes and vice-versa. **Any** route attribute on the controller makes **all** actions in the controller attribute routed.

Attribute routing and conventional routing use the same routing engine.

URL Generation and ambient values

Apps can use routing URL generation features to generate URL links to actions. Generating URLs eliminates hardcoding URLs, making code more robust and maintainable. This section focuses on the URL generation features provided by MVC and only cover basics of how URL generation works. See Routing for a detailed description of URL generation.

The IUrlHelper interface is the underlying element of infrastructure between MVC and routing for URL generation. An instance of `IUrlHelper` is available through the `Url` property in controllers, views, and view components.

In the following example, the `IUrlHelper` interface is used through the `Controller.Url` property to generate a URL to another action.

C# Copy

```
public class UrlGenerationController : Controller
{
    public IActionResult Source()
    {
        // Generates /UrlGeneration/Destination
        var url = Url.Action("Destination");
        return ControllerContext.MyDisplayRouteInfo("", $" URL = {url}");
    }

    public IActionResult Destination()
    {
        return ControllerContext.MyDisplayRouteInfo();
```

```
    }
}
```

If the app is using the default conventional route, the value of the `url` variable is the URL path string `/UrlGeneration/Destination`. This URL path is created by routing by combining:

- The route values from the current request, which are called **ambient values**.
- The values passed to `Url.Action` and substituting those values into the route template:

text Copy

```text
ambient values: { controller = "UrlGeneration", action = "Source" }
values passed to Url.Action: { controller = "UrlGeneration", action = "Destination" }
route template: {controller}/{action}/{id?}

result: /UrlGeneration/Destination
```

Each route parameter in the route template has its value substituted by matching names with the values and ambient values. A route parameter that doesn't have a value can:

- Use a default value if it has one.
- Be skipped if it's optional. For example, the `id` from the route template `{controller}/{action}/{id?}`.

URL generation fails if any required route parameter doesn't have a corresponding value. If URL generation fails for a route, the next route is tried until all routes have been tried or a match is found.

The preceding example of `Url.Action` assumes conventional routing. URL generation works similarly with attribute routing, though the concepts are different. With conventional routing:

- The route values are used to expand a template.
- The route values for `controller` and `action` usually appear in that template. This works because the URLs matched by routing adhere to a convention.

The following example uses attribute routing:

C# Copy

```
public class UrlGenerationAttrController : Controller
{
    [HttpGet("custom")]
    public IActionResult Source()
    {
        var url = Url.Action("Destination");
        return ControllerContext.MyDisplayRouteInfo("", $" URL = {url}");
    }

    [HttpGet("custom/url/to/destination")]
    public IActionResult Destination()
    {
       return ControllerContext.MyDisplayRouteInfo();
    }
}
```

The `Source` action in the preceding code generates `custom/url/to/destination`.

LinkGenerator was added in ASP.NET Core 3.0 as an alternative to `IUrlHelper`. `LinkGenerator` offers similar but more flexible functionality. Each method on `IUrlHelper` has a corresponding family of methods on `LinkGenerator` as well.

Generating URLs by action name

Url.Action, LinkGenerator.GetPathByAction, and all related overloads all are designed to generate the target endpoint by specifying a controller name and action name.

When using `Url.Action`, the current route values for `controller` and `action` are provided by the runtime:

- The value of `controller` and `action` are part of both ambient values and values. The method `Url.Action` always uses the current values of `action` and `controller` and generates a URL path that routes to the current action.

Routing attempts to use the values in ambient values to fill in information that wasn't provided when generating a URL. Consider a route like `{a}/{b}/{c}/{d}` with ambient values `{ a = Alice, b = Bob, c = Carol, d = David }`:

- Routing has enough information to generate a URL without any additional values.
- Routing has enough information because all route parameters have a value.

If the value `{ d = Donovan }` is added:

- The value `{ d = David }` is ignored.
- The generated URL path is `Alice/Bob/Carol/Donovan`.

Warning: URL paths are hierarchical. In the preceding example, if the value `{ c = Cheryl }` is added:

- Both of the values `{ c = Carol, d = David }` are ignored.
- There is no longer a value for `d` and URL generation fails.
- The desired values of `c` and `d` must be specified to generate a URL.

You might expect to hit this problem with the default route `{controller}/{action}/{id?}`. This problem is rare in practice because `Url.Action` always explicitly specifies a `controller` and `action` value.

Several overloads of Url.Action take a route values object to provide values for route parameters other than `controller` and `action`. The route values object is frequently used with `id`. For example, `Url.Action("Buy", "Products", new { id = 17 })`. The route values object:

- By convention is usually an object of anonymous type.
- Can be an `IDictionary<>` or a POCO).

Any additional route values that don't match route parameters are put in the query string.

C# Copy

```
public IActionResult Index()
{
    var url = Url.Action("Buy", "Products", new { id = 17, color = "red" });
    return Content(url);
}
```

The preceding code generates `/Products/Buy/17?color=red`.

The following code generates an absolute URL:

C# Copy

```
public IActionResult Index2()
{
    var url = Url.Action("Buy", "Products", new { id = 17 }, protocol: Request.Scheme);
    // Returns https://localhost:5001/Products/Buy/17
    return Content(url);
}
```

To create an absolute URL, use one of the following:

- An overload that accepts a `protocol`. For example, the preceding code.
- LinkGenerator.GetUriByAction, which generates absolute URIs by default.

Generate URLs by route

The preceding code demonstrated generating a URL by passing in the controller and action name. `IUrlHelper` also provides the Url.RouteUrl family of methods. These methods are similar to Url.Action, but they don't copy the current values of `action` and `controller` to the route values. The most common usage of `Url.RouteUrl`:

- Specifies a route name to generate the URL.
- Generally doesn't specify a controller or action name.

C# Copy

```
public class UrlGeneration2Controller : Controller
{
    [HttpGet("")]
    public IActionResult Source()
    {
        var url = Url.RouteUrl("Destination_Route");
        return ControllerContext.MyDisplayRouteInfo("", $" URL = {url}");
    }

    [HttpGet("custom/url/to/destination2", Name = "Destination_Route")]
    public IActionResult Destination()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
```

The following Razor file generates an HTML link to the `Destination_Route`:

CSHTML Copy

```
<h1>Test Links</h1>

<ul>
    <li><a href="@Url.RouteUrl("Destination_Route")">Test Destination_Route</a></li>
</ul>
```

Generate URLs in HTML and Razor

IHtmlHelper provides the HtmlHelper methods Html.BeginForm and Html.ActionLink to generate `<form>` and `<a>` elements respectively. These methods use the Url.Action method to generate a URL and they accept similar arguments. The `Url.RouteUrl` companions for `HtmlHelper` are `Html.BeginRouteForm` and `Html.RouteLink` which have similar functionality.

TagHelpers generate URLs through the `form` TagHelper and the `<a>` TagHelper. Both of these use `IUrlHelper` for their implementation. See Tag Helpers in forms for more information.

Inside views, the `IUrlHelper` is available through the `Url` property for any ad-hoc URL generation not covered by the above.

URL generation in Action Results

The preceding examples showed using `IUrlHelper` in a controller. The most common usage in a controller is to generate a URL as part of an action result.

The ControllerBase and Controller base classes provide convenience methods for action results that reference another action. One typical usage is to redirect after accepting user input:

C# Copy

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Edit(int id, Customer customer)
{
    if (ModelState.IsValid)
    {
        // Update DB with new details.
        ViewData["Message"] = $"Successful edit of customer {id}";
        return RedirectToAction("Index");
    }
    return View(customer);
}
```

The action results factory methods such as RedirectToAction and CreatedAtAction follow a similar pattern to the methods on `IUrlHelper`.

Special case for dedicated conventional routes

Conventional routing can use a special kind of route definition called a dedicated conventional route. In the following example, the route named `blog` is a dedicated conventional route:

C# Copy

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(name: "blog",
                pattern: "blog/{*article}",
                defaults: new { controller = "Blog", action = "Article" });
    endpoints.MapControllerRoute(name: "default",
                pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

Using the preceding route definitions, `Url.Action("Index", "Home")` generates the URL path `/` using the `default` route, but why? You might guess the route values `{ controller = Home, action = Index }` would be enough to generate a URL using `blog`, and the result would be `/blog?action=Index&controller=Home`.

Dedicated conventional routes rely on a special behavior of default values that don't have a corresponding route parameter that prevents the route from being too greedy with URL generation. In this case the default values are `{ controller = Blog, action = Article }`, and neither `controller` nor `action` appears as a route parameter. When routing performs URL generation, the values provided must match the default values. URL generation using `blog` fails because the values `{ controller = Home, action = Index }` don't match `{ controller = Blog, action = Article }`. Routing then falls back to try `default`, which succeeds.

Areas

Areas are an MVC feature used to organize related functionality into a group as a separate:

- Routing namespace for controller actions.
- Folder structure for views.

Using areas allows an app to have multiple controllers with the same name, as long as they have different areas. Using areas creates a hierarchy for the purpose of routing by adding another route parameter, `area` to `controller` and `action`. This section discusses how routing interacts with areas. See Areas for details about how areas are used with views.

The following example configures MVC to use the default conventional route and an `area` route for an `area` named `Blog`:

C# Copy

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapAreaControllerRoute("blog_route", "Blog",
        "Manage/{controller}/{action}/{id?}");
    endpoints.MapControllerRoute("default_route", "{controller}/{action}/{id?}");
});
```

In the preceding code, MapAreaControllerRoute is called to create the `"blog_route"`. The second parameter, `"Blog"`, is the area name.

When matching a URL path like `/Manage/Users/AddUser`, the `"blog_route"` route generates the route values `{ area = Blog, controller = Users, action = AddUser }`. The `area` route value is produced by a default value for `area`. The route created by `MapAreaControllerRoute` is equivalent to the following:

C# Copy

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute("blog_route", "Manage/{controller}/{action}/{id?}",
        defaults: new { area = "Blog" }, constraints: new { area = "Blog" });
    endpoints.MapControllerRoute("default_route", "{controller}/{action}/{id?}");
});
```

`MapAreaControllerRoute` creates a route using both a default value and constraint for `area` using the provided area name, in this case `Blog`. The default value ensures that the route always produces `{ area = Blog, ... }`, the constraint requires the value `{ area = Blog, ... }` for URL generation.

Conventional routing is order-dependent. In general, routes with areas should be placed earlier as they're more specific than routes without an area.

Using the preceding example, the route values `{ area = Blog, controller = Users, action = AddUser }` match the following action:

C# Copy

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace1
{
    [Area("Blog")]
    public class UsersController : Controller
    {
        // GET /manage/users/adduser
        public IActionResult AddUser()
        {
            var area = ControllerContext.ActionDescriptor.RouteValues["area"];
            var actionName = ControllerContext.ActionDescriptor.ActionName;
            var controllerName = ControllerContext.ActionDescriptor.ControllerName;

            return Content($"area name:{area}" +
```

```
                $" controller:{controllerName}  action name: {actionName}");
        }
    }
}
```

The [Area] attribute is what denotes a controller as part of an area. This controller is in the `Blog` area. Controllers without an `[Area]` attribute are not members of any area, and do **not** match when the `area` route value is provided by routing. In the following example, only the first controller listed can match the route values `{ area = Blog, controller = Users, action = AddUser }`.

C# Copy

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace1
{
    [Area("Blog")]
    public class UsersController : Controller
    {
        // GET /manage/users/adduser
        public IActionResult AddUser()
        {
            var area = ControllerContext.ActionDescriptor.RouteValues["area"];
            var actionName = ControllerContext.ActionDescriptor.ActionName;
            var controllerName = ControllerContext.ActionDescriptor.ControllerName;

            return Content($"area name:{area}" +
                $" controller:{controllerName}  action name: {actionName}");
        }
    }
}
```

C# Copy

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace2
{
    // Matches { area = Zebra, controller = Users, action = AddUser }
    [Area("Zebra")]
    public class UsersController : Controller
    {
        // GET /zebra/users/adduser
        public IActionResult AddUser()
        {
            var area = ControllerContext.ActionDescriptor.RouteValues["area"];
            var actionName = ControllerContext.ActionDescriptor.ActionName;
            var controllerName = ControllerContext.ActionDescriptor.ControllerName;

            return Content($"area name:{area}" +
                $" controller:{controllerName}  action name: {actionName}");
        }
    }
}
```

C# Copy

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace3
{
    // Matches { area = string.Empty, controller = Users, action = AddUser }
    // Matches { area = null, controller = Users, action = AddUser }
    // Matches { controller = Users, action = AddUser }
    public class UsersController : Controller
    {
        // GET /users/adduser
        public IActionResult AddUser()
        {
            var area = ControllerContext.ActionDescriptor.RouteValues["area"];
```

```
            var actionName = ControllerContext.ActionDescriptor.ActionName;
            var controllerName = ControllerContext.ActionDescriptor.ControllerName;

            return Content($"area name:{area}" +
                $" controller:{controllerName}  action name: {actionName}");
        }
    }
}
```

The namespace of each controller is shown here for completeness. If the preceding controllers uses the same namespace, a compiler error would be generated. Class namespaces have no effect on MVC's routing.

The first two controllers are members of areas, and only match when their respective area name is provided by the `area` route value. The third controller isn't a member of any area, and can only match when no value for `area` is provided by routing.

In terms of matching *no value*, the absence of the `area` value is the same as if the value for `area` were null or the empty string.

When executing an action inside an area, the route value for `area` is available as an ambient value for routing to use for URL generation. This means that by default areas act *sticky* for URL generation as demonstrated by the following sample.

C# Copy

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapAreaControllerRoute(name: "duck_route",
                                     areaName: "Duck",
                                     pattern: "Manage/{controller}/{action}/{id?}");
    endpoints.MapControllerRoute(name: "default",
                                 pattern: "Manage/{controller=Home}/{action=Index}/{id?}");
});
```

C# Copy

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace4
{
    [Area("Duck")]
    public class UsersController : Controller
    {
        // GET /Manage/users/GenerateURLInArea
        public IActionResult GenerateURLInArea()
        {
            // Uses the 'ambient' value of area.
            var url = Url.Action("Index", "Home");
            // Returns /Manage/Home/Index
            return Content(url);
        }

        // GET /Manage/users/GenerateURLOutsideOfArea
        public IActionResult GenerateURLOutsideOfArea()
        {
            // Uses the empty value for area.
            var url = Url.Action("Index", "Home", new { area = "" });
            // Returns /Manage
            return Content(url);
        }
    }
}
```

The following code generates a URL to `/Zebra/Users/AddUser`:

C# Copy

```
public class HomeController : Controller
{
    public IActionResult About()
    {
```

```
        var url = Url.Action("AddUser", "Users", new { Area = "Zebra" });
        return Content($"URL: {url}");
    }
```

Action definition

Public methods on a controller, except those with the NonAction attribute, are actions.

Sample code

- MyDisplayRouteInfo is provided by the Rick.Docs.Samples.RouteInfo NuGet package and displays route information.
- View or download sample code (how to download)

Debug diagnostics

For detailed routing diagnostic output, set `Logging:LogLevel:Microsoft` to `Debug`. In the development environment, set the log level in *appsettings.Development.json*:

JSON Copy

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Debug",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
}
```