

# .net core & asp.net core

# Exception

An exception is when a member fails to complete the task it is supposed to perform as indicated by its name.

```
internal sealed class Account
{
    public static void Transfer(Account from, Account to, Decimal amount)
    {
        from -= amount;
        to += amount;
    }
}
```

# Exception handling

```
bool f = "Jeff".Substring(1, 1).ToUpper().EndsWith("E"); // true
```

In fact, there are many object-oriented constructs—constructors, getting/setting a property, adding/removing an event, calling an operator overload, calling a conversion operator—that have no way to return error codes, but these constructs must still be able to report an error. The mechanism provided by the Microsoft .NET Framework and all programming languages that support it is called exception handling.

0 references

```
private void SomeMethod() {  
    try {  
        // Put code requiring graceful recovery and/or cleanup operations here...  
    }  
    catch (InvalidOperationException) {  
        // Put code that recovers from an InvalidOperationException here...  
    }  
    catch (IOException) {  
        // Put code that recovers from an IOException here...  
    }  
    catch {  
        // Put code that recovers from any kind of exception other than those above here...  
        // When catching any exception, you usually re-throw the exception.  
        // I explain re-throwing later in this chapter.  
        throw;  
    }  
    finally  
    {  
        // Put code that cleans up any operations started within the try block here...  
        // The code in here ALWAYS executes, regardless of whether an exception is thrown.  
    }  
    // Code below the finally block executes if no exception is thrown within the try block  
    // or if a catch block catches the exception and doesn't throw or re-throw an exception.  
}
```

# try and catch block

A try **block** contains code that requires common cleanup operations, exception-recovery operations, or both.

A **catch** block contains code to execute in response to an exception. A **try** block can have zero or more catch blocks associated with it. If the code in a **try** block doesn't cause an exception to be thrown, the CLR will never execute the code contained within any of its catch blocks. The thread will simply skip over all of the catch blocks and execute the code in the **finally** block (if one exists). After the code in the **finally** block executes, execution continues with the statement following the **finally** block.

The parenthetical expression appearing after the catch keyword is called the **catch type**

0 references

```
private void SomeMethod() {  
    try {  
        // Put code requiring graceful recovery and/or cleanup operations here...  
    }  
    catch (InvalidOperationException) {  
        // Put code that recovers from an InvalidOperationException here...  
    }  
    catch (IOException) {  
        // Put code that recovers from an IOException here...  
    }  
    catch {  
        // Put code that recovers from any kind of exception other than those above here...  
        // When catching any exception, you usually re-throw the exception.  
        // I explain re-throwing later in this chapter.  
        throw;  
    }  
    finally  
    {  
        // Put code that cleans up any operations started within the try block here...  
        // The code in here ALWAYS executes, regardless of whether an exception is thrown.  
    }  
    // Code below the finally block executes if no exception is thrown within the try block  
    // or if a catch block catches the exception and doesn't throw or re-throw an exception.  
}
```

Catch type



# The *finally* Block

A *finally* block contains code that's guaranteed to execute.<sup>12</sup> Typically, the code in a *finally* block performs the cleanup operations required by actions taken in the *try* block. For example, if you open a file in a *try* block, you'd put the code to close the file in a *finally* block.

A *try* block doesn't require a *finally* block associated with it; sometimes the code in a *try* block just doesn't require any cleanup code. However, if you do have a *finally* block, it must appear after any and all *catch* blocks. A *try* block can have no more than one *finally* block associated with it

0 references

```
private void ReadData(String pathname)
{
    FileStream fs = null; try
    {
        fs = new FileStream(pathname, FileMode.Open);
        // Process the data in the file...
    }
    catch (IOException) {
        // Put code that recovers from an IOException here...
    }
    finally
    {
        // Make sure that the file gets closed.
        if (fs != null) fs.Close();
    }
}
```



Property	Access	Type	Description
Message	Read-only	String	Contains helpful text indicating why the exception was thrown. The message is typically written to a log when a thrown exception is unhandled.
Data	Read-only	IDictionary	A reference to a collection of key-value pairs. Usually, the code throwing the exception adds entries to this collection prior to throwing it;
Source	Read/write	String	Contains the name of the assembly that generated the exception.
StackTrace	Read-only	String	Contains the names and signatures of methods called that led up to the exception being thrown. This property is invaluable for debugging.
TargetSite	Read-only	MethodBase	Contains the method that threw the exception.
HelpLink	Read-only	String	Contains a URL (such as file:///C:/MyApp/Help.htm#MyExceptionHelp) to documentation that can help a user understand the exception
InnerException	Read-only	Exception	Indicates the previous exception if the current exception were raised while handling an exception

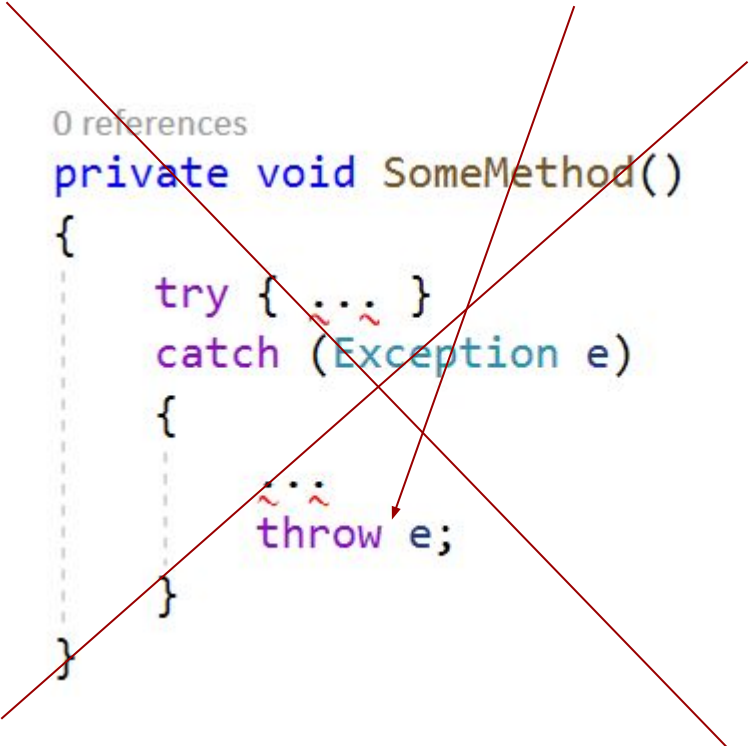
# Throw

Important When you throw an exception, the CLR resets the starting point for the exception; that is, the CLR remembers only the location where the most recent exception object was thrown.

CLR thinks this is where exception originated.

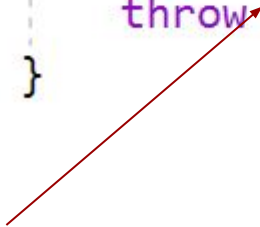
0 references

```
private void SomeMethod()  
{  
    try { ... }  
    catch (Exception e)  
    {  
        ...  
        throw e;  
    }  
}
```



0 references

```
private void SomeMethod()  
{  
    try { ... }  
    catch (Exception e)  
    {  
        ...  
        throw ;  
    }  
}
```



This has no effect on where the CLR thinks the exception

# Best practices

- **Use *finally* Blocks Liberally** . You should use *finally* blocks to clean up from any operation that successfully started before returning to your caller or allowing code following the *finally* block to execute
- **Don't *Catch* Everything** . When you catch an exception, you're stating that you expected this exception, you understand why it occurred, and you know how to deal with it
- **Recovering Gracefully from an Exception**
- **Backing Out of a Partially Completed Operation When an Unrecoverable Exception Occurs—Maintaining State**

# Unhandled Exceptions

When an exception is thrown, the CLR climbs up the call stack looking for catch blocks that match the type of the exception object being thrown. If no catch block matches the thrown exception type, an unhandled exception occurs. When the CLR detects that any thread in the process has had an unhandled exception, the CLR terminates the process. An unhandled exception identifies a situation that the application didn't anticipate and is considered to be a true bug in the application.

# The **System.Exception** Class

The CLR allows an instance of any type to be thrown for an exception—from an `Int32` to a `String` and beyond. However, Microsoft decided against forcing all programming languages to throw and catch exceptions of any type, so they defined the `System.Exception` type and decreed that all CLS-compliant programming languages must be able to throw and catch exceptions whose type is derived from this type. Exception types that are derived from `System.Exception` are said to be CLS-compliant. C# and many other language compilers allow your code to throw only CLS-compliant exceptions.

# ASP.NET CORE ROUTING

# Routing basics

Routing is responsible for mapping request URIs to endpoints and dispatching incoming requests to those endpoints. Routes are defined in the app and configured when the app starts. A route can optionally extract values from the URL contained in the request, and these values can then be used for request processing. Using route information from the app, routing is also able to generate URLs that map to endpoints.

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.



# The routing system characteristics

- Route template syntax is used to define routes with tokenized route parameters.
- Conventional-style and attribute-style endpoint configuration is permitted.
- IRouteConstraint is used to determine whether a URL parameter contains a valid value for a given endpoint constraint.
- App models, such as MVC/Razor Pages, register all of their endpoints, which have a predictable implementation of routing scenarios.
- The routing implementation makes routing decisions wherever desired in the middleware pipeline.
- Middleware that appears after a Routing Middleware can inspect the result of the Routing Middleware's endpoint decision for a given request URI.
- It's possible to enumerate all of the endpoints in the app anywhere in the middleware pipeline.
- An app can use routing to generate URLs (for example, for redirection or links) based on endpoint information and thus avoid hard-coded URLs, which helps maintainability.
- URL generation is based on addresses, which support arbitrary extensibility:
  - The Link Generator API (LinkGenerator) can be resolved anywhere using dependency injection (DI) to generate URLs.
  - Where the Link Generator API isn't available via DI, IUrlHelper offers methods to build URLs.

# Url matching

URL matching is the process by which routing dispatches an incoming request to an *endpoint*. This process is based on data in the URL path but can be extended to consider any data in the request. The ability to dispatch requests to separate handlers is key to scaling the size and complexity of an app.

The routing system in endpoint routing is responsible for all dispatching decisions. Since 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.

# Url matching

When the endpoint delegate is executed, the properties of [RouteContext.RouteData](#) are set to appropriate values based on the request processing performed thus far.

[RouteData.Values](#) is a dictionary of *route values* produced from the route. These values are usually determined by tokenizing the URL and can be used to accept user input or to make further dispatching decisions inside the app.

[RouteData.DataTokens](#) is a property bag of additional data related to the matched route. [DataTokens](#) are provided to support associating state data with each route so that the app can make decisions based on which route matched. These values are developer-defined and do not affect the behavior of routing in any way. Additionally, values stashed in [RouteData.DataTokens](#) can be of any type, in contrast to [RouteData.Values](#), which must be convertible to and from strings.

[RouteData.Routers](#) is a list of the routes that took part in successfully matching the request. Routes can be nested inside of one another. The [Routers](#) property reflects the path through the logical tree of routes that resulted in a match. Generally, the first item in [Routers](#) is the route collection and should be used for URL generation. The last item in [Routers](#) is the route handler that matched.

# Url generation

1 reference

```
public class HomeController: Controller
```

```
{
```

```
    private readonly LinkGenerator _linkGenerator;
```

0 references | 0 exceptions

```
    public HomeController(LinkGenerator linkGenerator)
```

```
    {
```

```
        _linkGenerator = linkGenerator;
```

```
    }
```

0 references | 0 requests | 0 exceptions

```
    public IActionResult Index()
```

```
    {
```

```
        var url = _linkGenerator.GetPathByAction("Privacy", "Home");
```

```
        return Redirect(url);
```

```
    }
```

```
}
```

# Url generation

In views you can use

```
@Html.ActionLink("link text", "index", "hello", new { id = "123" }, null)
```

Generates `<a href="/hello/index/123">link text</a>`

# MapRoute

The following code example is an example of a [MapRoute](#) call used by a typical ASP.NET Core MVC route definition:

```
routes.MapRoute(  
    name: "default",  
    template: "{controller=Home}/{action=Index}/{id?}");
```

This template matches a URL path and extracts the route values. For example, the path `/Products/Details/17` generates the following route values: `{ controller = Products, action = Details, id = 17 }`

# Route parameters

Route values are determined by splitting the URL path into segments and matching each segment with the *route parameter* name in the route template. Route parameters are named. The parameters defined by enclosing the parameter name in braces { ... }.

The preceding template could also match the URL path / and produce the values { controller = Home, action = Index }. This occurs because the {controller} and {action} route parameters have default values and the id route parameter is optional. An equals sign (=) followed by a value after the route parameter name defines a default value for the parameter. A question mark (?) after the route parameter name defines an optional parameter.

Route parameters with a default value *always* produce a route value when the route matches. Optional parameters don't produce a route value if there was no corresponding URL path segment

```
routes.MapRoute(  
    name: "default",  
    template: "{controller=Home}/{action=Index}/{id:int}");
```

```
routes.MapRoute(  
    name: "default_route",  
    template: "{controller}/{action}/{id?}",  
    defaults: new { controller = "Home", action = "Index" });
```

```
routes.MapRoute(  
    name: "default_route",  
    template: "{controller=Home}/{action=Index}/{id?}");
```



The preceding template matches a URL path like `/Blog/All-About-Routing/Introduction` and extracts the values `og`,

```
{  
    template: "Blog/{**article}",  
    controller = Blog, controller = "Blog", action = "ReadArticle" });  
    action = ReadArticle,  
    article = All-About-Routing/Introduction  
}.
```

The default route values for `controller` and `action` are produced by the route even though there are no corresponding route parameters in the template. Default values can be specified in the route template. The `article` route parameter is defined as a *catch-all* by the appearance of a double asterisk (`**`) before the route parameter name. Catch-all route parameters capture the remainder of the URL path and can also match the empty string.

# Reserved routing

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

# 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 yes/no decision about whether or not 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.

constraint	Example	Example Matches	Notes
int	{id:int}	123456789, -123456789	Matches any integer
bool	{active:bool}	true, FALSE	Matches true or false (case-insensitive)
datetime	{dob:datetime}	2016-12-31, 2016-12-31 7:32pm	Matches a valid DateTime value (in the invariant culture - see warning)
decimal	{price:decimal}	49.99, -1,000.01	Matches a valid decimal value (in the invariant culture - see warning)
double	{weight:double}	1.234, -1,001.01e8	Matches a valid double value (in the invariant culture - see warning)
float	{weight:float}	1.234, -1,001.01e8	Matches a valid float value (in the invariant culture - see warning)
guid	{id:guid}	CD2C1638-1638-72D5-1638-DEADBEEF1638, {CD2C1638-1638-72D5-1638-DEADBEEF1638}	Matches a valid Guid value

<code>long</code>	<code>{ticks:long}</code>	123456789, -123456789	Matches a valid <code>long</code> value
<code>minlength(value)</code>	<code>{username:minlength(4)}</code>	Rick	String must be at least 4 characters
<code>maxlength(value)</code>	<code>{filename:maxlength(8)}</code>	Richard	String must be no more than 8 characters
<code>length(length)</code>	<code>{filename:length(12)}</code>	somefile.txt	String must be exactly 12 characters long
<code>length(min,max)</code>	<code>{filename:length(8,16)}</code>	somefile.txt	String must be at least 8 and no more than 16 characters long
<code>min(value)</code>	<code>{age:min(18)}</code>	19	Integer value must be at least 18
<code>max(value)</code>	<code>{age:max(120)}</code>	91	Integer value must be no more than 120
<code>range(min,max)</code>	<code>{age:range(18,120)}</code>	91	Integer value must be at least 18 but no more than 120
<code>alpha</code>	<code>{name:alpha}</code>	Rick	String must consist of one or more alphabetical characters (a-z, case-insensitive)
<code>regex(expression)</code>	<code>{ssn:regex(^\\d{{3}}-\\d{{2}}-\\d{{4}}\$)}</code>	123-45-6789	String must match the regular expression (see tips about defining a regular expression)
<code>required</code>	<code>{name:required}</code>	Rick	Used to enforce that a non-parameter value is present during URL generation

# Route constraints

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 GetById(int id) { }
```

# Regular expressions

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

# Custom routes constraints

In addition to the built-in route constraints, custom route constraints can be created by implementing the [IRouteConstraint](#) interface. The [IRouteConstraint](#) interface contains a single method, `Match`, which returns `true` if the constraint is satisfied and `false` otherwise.

```
services.AddRouting(options =>
{
    options.ConstraintMap.Add("customName", typeof(MyCustomConstraint));
});
```

```
[HttpGet("{id:customName}")]
public ActionResult<string> Get(string id)
```



# Conventional routing

The `default` route:

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

is an example of a *conventional routing*. We call this style *conventional routing* because it establishes a *convention* for URL paths:

- the first path segment maps to the *controller* name
- the second maps to the *action* name.
- the third segment is used for an optional *id* used to map to a model entity

# Multiple routes

```
app.UseMvc(routes =>
{
    routes.MapRoute("blog", "blog/{*article}",
        defaults: new { controller = "Blog", action = "Article" });
    routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
});
```

The `blog` route here is a *dedicated conventional route*, meaning that it uses the conventional routing system, but is dedicated to a specific action. Since `controller` and `action` don't appear in the route template as parameters, they can only have the default values, and thus this route will always map to the action `BlogController.Article`.

# Fallback

As part of request processing, MVC will verify that the route values can be used to find a controller and action in your application. If the route values don't match an action then the route isn't considered a match, and the next route will be tried. This is called ***fallback***, and it's intended to simplify cases where conventional routes overlap.

# Disambiguating actions

When two actions match through routing, MVC must disambiguate to choose the 'best' candidate or else throw an exception

```
public class ProductsController : Controller
{
    public IActionResult Edit(int id) { ... }

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

The `HttpPostAttribute ( [HttpPost] )` is an implementation of `IActionConstraint` that will only allow the action to be selected when the HTTP verb is `POST`. The presence of an `IActionConstraint` makes the `Edit(int, Product)` a 'better' match than `Edit(int)`, so `Edit(int, Product)` will be tried first.

If multiple routes match, and MVC can't find a 'best' route, it will throw an `AmbiguousActionException`.

# Route names

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

```
app.UseMvc(routes =>
{
    routes.MapRoute("blog", "blog/{*article}",
        defaults: new { controller = "Blog", action = "Article" });
    routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
});
```

Route names must be unique application-wide.

Route names have no impact on URL matching or handling of requests; they're used only for URL generation

# Attribute routing

The `HomeController.Index()` action will be executed for any of the URL paths `/`, `/Home`, or `/Home/Index`.

```
public class HomeController : Controller
{
    [Route("")]
    [Route("Home")]
    [Route("Home/Index")]
    public IActionResult Index()
    {
        return View();
    }
    [Route("Home/About")]
    public IActionResult About()
    {
        return View();
    }
    [Route("Home/Contact")]
    public IActionResult Contact()
    {
        return View();
    }
}
```

# Attribute routing with Http[Verb] attributes

Attribute routing can also make use of the Http[Verb] attributes such as `HttpPostAttribute`. All of these attributes can accept a route template

```
[HttpGet("/products")]
public IActionResult ListProducts()
{
    // ...
}

[HttpPost("/products")]
public IActionResult CreateProduct(...)
{
    // ...
}
```



```
public class ProductsApiController : Controller
{
    [HttpGet("/products/{id}", Name = "Products_List")]
    public IActionResult GetProduct(int id) { ... }
}
```

The `ProductsApi.GetProduct(int)` action will be executed for a URL path like `/products/3` but not for a URL path like `/products`. See [Routing](#) for a full description of route templates and related options.

# Combining routes

```
[Route("products")]
public class ProductsApiController : Controller
{
    [HttpGet]
    public IActionResult ListProducts() { ... }

    [HttpGet("{id}")]
    public ActionResult GetProduct(int id) { ... }
}
```

In this example the URL path `/products` can match `ProductsApi.ListProducts` and the URL path `/products/5` can match `ProductsApi.GetProduct(int)`. Both of these actions only match HTTP `GET` because they're decorated with the `HttpGetAttribute`.

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

```
[Route("Home")]
public class HomeController : Controller
{
    [Route("")] // Combines to define the route template "Home"
    [Route("Index")] // Combines to define the route template "Home/Index"
    [Route("/")] // Doesn't combine, defines the route template ""
    public IActionResult Index()
    {
        ViewData["Message"] = "Home index";
        var url = Url.Action("Index", "Home");
        ViewData["Message"] = "Home index" + "var url = Url.Action; = " + url;
        return View();
    }

    [Route("About")] // Combines to define the route template "Home/About"
    public IActionResult About()
    {
        return View();
    }
}
```

# Ordering attribute routes

In contrast to conventional routes which execute in a defined order, attribute routing builds a tree and matches all routes simultaneously. This behaves as-if the route entries were placed in an ideal ordering; the most specific routes have a chance to execute before the more general routes.

For example, a route like `blog/search/{topic}` is more specific than a route like `blog/{*article}`. Logically speaking the `blog/search/{topic}` route 'runs' first, by default, because that's the only sensible ordering. Using conventional routing, the developer is responsible for placing routes in the desired order

# Token replacement in route templates ([controller], [action], [area])

For convenience, attribute routes support *token replacement* by enclosing a token in square-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

```
[Route("[controller]/[action]")]
public class ProductsController : Controller
{
    [HttpGet] // Matches '/Products/List'
    public IActionResult List() {
        // ...
    }

    [HttpGet("{id}")] // Matches '/Products/Edit/{id}'
    public IActionResult Edit(int id) {
        // ...
    }
}
```

# Combine with inheritance

```
[Route("api/[controller]")]
public abstract class MyBaseController : Controller { ... }

public class ProductsController : MyBaseController
{
    [HttpGet] // Matches '/api/Products'
    public IActionResult List() { ... }

    [HttpPut("{id}")] // Matches '/api/Products/{id}'
    public IActionResult Edit(int id) { ... }
}
```

# Multiple routes

```
[Route("Store")]
[Route("[controller]")]
public class ProductsController : Controller
{
    [HttpPost("Buy")]      // Matches 'Products/Buy' and 'Store/Buy'
    [HttpPost("Checkout")] // Matches 'Products/Checkout' and 'Store/Checkout'
    public IActionResult Buy()
}
```

# Optional parameters, default values, and constraints

```
[HttpPost("product/{id:int}")]  
public IActionResult ShowProduct(int id)  
{  
    // ...  
}
```



# Mixed routing: Attribute routing vs conventional routing

MVC applications 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.

# URL generation

MVC applications can use routing's URL generation features to generate URL links to actions. Generating URLs eliminates hardcoding URLs, making your code more robust and maintainable. This section focuses on the URL generation features provided by MVC and will only cover basics of how URL generation works. See [Routing](#) for a detailed description of URL generation.

The `IUrlHelper` interface is the underlying piece of infrastructure between MVC and routing for URL generation. You'll find an instance of `IUrlHelper` available through the `Url` property in controllers, views, and view components.

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

```
using Microsoft.AspNetCore.Mvc;
```

```
public class UrlGenerationController : Controller
```

```
{
```

```
    [HttpGet("")]
```

```
    public IActionResult Source()
```

```
    {
```

```
        var url = Url.Action("Destination"); // Generates /custom/url/to/destination
```

```
        return Content($"Go check out {url}, it's really great.");
```

```
    }
```

```
    [HttpGet("custom/url/to/destination")]
```

```
    public IActionResult Destination() {
```

```
        return View();
```

```
    }
```

```
}
```

# Areas

[Areas](#) are an MVC feature used to organize related functionality into a group as a separate routing-namespace (for controller actions) and folder structure (for views). Using areas allows an application 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 will discuss 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`:

# Areas

```
app.UseMvc(routes =>
{
    routes.MapAreaRoute("blog_route", "Blog",
        "Manage/{controller}/{action}/{id?}");
    routes.MapRoute("default_route", "{controller}/{action}/{id?}");
});
```

When matching a URL path like `/Manage/Users/AddUser`, the first route will produce the route values `{ area = Blog, controller = Users, action = AddUser }`

# Area attribute

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace1
{
    [Area("Blog")]
    public class UsersController : Controller
    {
        public IActionResult AddUser()
        {
            return View();
        }
    }
}
```