



Routing

.NET CORE

console app:

Program.Main

everything else you control

method1

method2

methodA

class library

ideally, you can think about & write this code without worrying about exactly what's in the main method

from this library's point of view... it just gets called, it doesn't control exactly when its code runs.

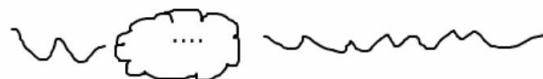
in the view, we can use Razor syntax

- @-expressions, @{} blocks
- loops like @foreach
- HTML helpers like @Html.ActionLink
- tag helpers like <a asp-for="...">

ASP.NET Core

Program.Main, Program.CreateHostBuilder
Startup.ConfigureServices, Startup.Configure

} initial startup code,
before any HTTP requests



Controller1

Controller2

action methods

views

ASP.NET Core's own code
creates your controllers, based on
routing

1. user sends HTTP request (e.g. from browser)
2. routing decides which controller and which action method can handle this request. (controlled from Startup.Configure, with MapRoute calls)
3. filters run, model binding happens, action method is called.
4. action method interacted with the model (e.g. contacts DB through a repository, calls business logic methods on things)
5. action method chooses a view to render, and optionally passes it data in a model/viewmodel object, and/or ViewData/ViewBag/TempData.
6. other filters run, then the view is rendered (aka the result is executed) - the view's Razor stuff is run, we get real HTML out and send it to the user.
7. re-runs on every request

time



requests each with their own steps 1-7

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

- defined in startup code or attributes,
- describe how URL paths are matched to actions, and
- are used to generate URLs for links.

Actions are either conventionally routed or attribute routed.

[HTTPS://DOCS.MICROSOFT.COM/EN-US/ASPNET/CORE/MVC/CONTROLLERS/ROUTING?VIEW=ASPNETCORE-3.1](https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/routing?view=aspnetcore-3.1)

Controllers

<https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/actions?view=aspnetcore-3.1>

A **controller** is a class used to define and group a set of **actions**. **Controllers** logically group similar **actions** together. This allows routing, caching, and authorization, to be applied collectively.

Within the **Model-View-Controller** pattern, a **controller** is responsible for the initial processing of a request and instantiation of a **model**. Business decisions should be performed within the **model**.

To be a **controller**, at least one of these conditions is true:

- The class name is suffixed with **Controller**.
- The class inherits from a class whose name is suffixed with **Controller**.
- The **[Controller]** attribute is applied to the class.

Controller classes reside in the project's root-level **Controllers** folder and inherit from **Microsoft.AspNetCore.Mvc.Controller**.

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using MvcProjectStarter.Models;

namespace MvcProjectStarter.Controllers
{
    public class HomeController : Controller
    {
        private readonly ILogger<HomeController>

        public HomeController(ILogger<HomeContro
        {
```


Action Methods

<https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/actions?view=aspnetcore-3.1#defining-actions>

- An **action method** is a method in a **controller** which handles requests.
- All public methods in a controller (except those with the **[NonAction]** attribute) are **actions**.
- Parameters on **actions** are **bound** to **request** data and are validated using **Model Binding**.
- **Model validation** occurs for everything that's **Model-Bound**.
- The **ModelState.IsValid** method indicates whether **Model Binding** and **validation** succeeded.
- **Action methods** should contain logic for mapping a request to a business concern.
- Business concerns should typically be represented as **services** that the **controller** accesses through **dependency injection**.
- **Actions** can return anything, but usually return an **ActionResult** or **Task<ActionResult>** (for async methods).

```
namespace MvcProjectStarter.Controllers
{
    public class SongsController : Controller
    {
        private readonly MvcSongContext _context;

        public SongsController(MvcSongContext context)
        {
            _context = context;
        }

        // GET: Songs
        public async Task<ActionResult> Index()
        {
            return View(await _context.Song.ToListAsync());
        }

        // GET: Songs/Details/5
        public async Task<ActionResult> Details(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            var song = await _context.Song
                .FirstOrDefaultAsync(m => m.id == id);
            if (song == null)
            {
                return NotFound();
            }

            return View(song);
        }
    }
}
```

Model Binding

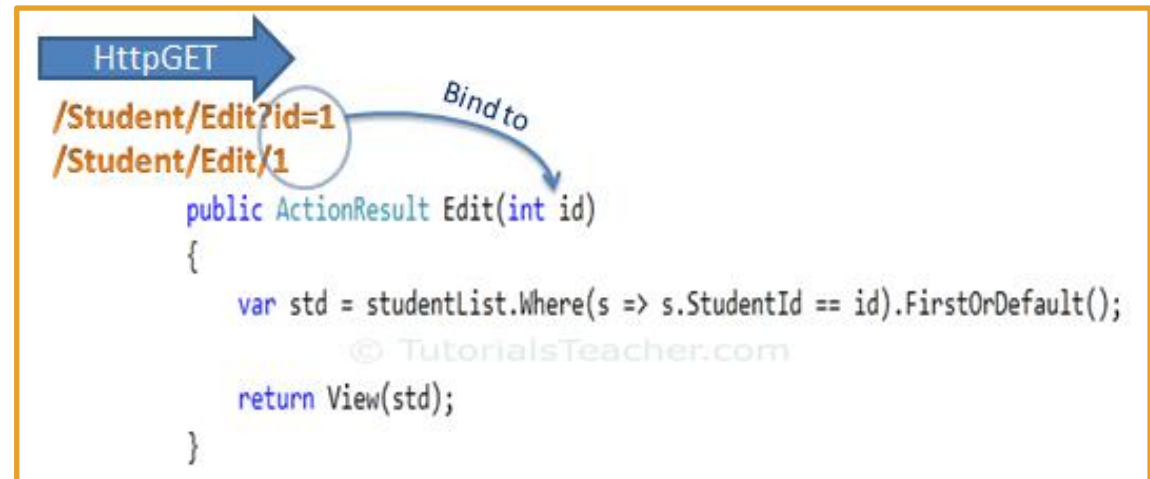
<https://docs.microsoft.com/en-us/aspnet/core/mvc/models/model-binding?view=aspnetcore-3.1>

Controllers and **Action Methods** work with data that comes from HTTP requests. (Ex. **POST**ed form fields provide values for the properties of the model.)

Writing code to retrieve each of these values and convert them from strings to .NET **types** would be tedious and error-prone. **Model Binding** automates this process.

The **Model Binding** system:

- Retrieves data from various sources such as **route data**, **form fields**, and **query strings**.
- Provides the data to **controllers** in **Action Method** parameters and public properties.
- Converts string data to .NET types.
- Updates properties of complex types.



Model Binding

<https://docs.microsoft.com/en-us/aspnet/core/mvc/models/model-binding?view=aspnetcore-3.1>

Model Binding goes through the following steps after the routing system selects the **action** method:

1. Finds the first parameter of **GetByID (id)**.
2. Looks through the HTTP request and finds id = "2" in route data.
3. Converts the string "2" into integer 2.
4. Finds the next parameter of **GetByID (dogsOnly)**.
5. Finds "DogsOnly=true" in the query string. Name matching is not case-sensitive.
6. Converts the string **"true"** to a boolean true.

Suppose you have the following action method:

C#

```
[HttpGet("{id}")]  
public ActionResult<Pet> GetById(int id, bool dogsOnly)
```

And the app receives a request with this URL:

```
http://contoso.com/api/pets/2?DogsOnly=true
```

Different Controller Helper (Action) Methods

<https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/actions?view=aspnetcore-3.1#controller-helper-methods>

The ***Controller*** provides access to three categories of helper methods, resulting in

<u>empty response body</u>	<u>a non-empty response body with a predefined content type</u>	<u>a non-empty response body formatted in a content type negotiated with the client</u>
HTTP Status Code – Ex. BadRequest, NotFound, and Ok	View which uses a model to render HTML. (EX. Return View(customer);	This category is better known as Content Negotiation . Content negotiation applies whenever an <i>action</i> returns an ActionResult type or something other than an IActionResult. (Ex. BadRequest, CreatedAtRoute, and Ok)
Redirect - returns a redirect to an action or destination (Redirect, LocalRedirect, RedirectToAction, or RedirectToRoute).	Formatted Response - JSON or a similar data exchange format to represent an object, (Ex. Json(customer);)	

Conventional Routing

<https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/routing?view=aspnetcore-3.1#cr>

Startup.Configure typically has code similar to the following when using conventional routing. Inside the call to ***UseEndpoints***, ***MapControllerRoute*** is used to create a single route. The single route is named the ***default*** route.

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

Conventional Routing

<https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/routing?view=aspnetcore-3.1#set-up-conventional-route>

<https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/routing?view=aspnetcore-3.1#multiple-conventional-routes>

The route template (in Startup.cs)

`"{controller=Home}/{action=Index}/{id?}"`
matches a URL path like
`/Products/Details/5`.

The route template *tokenizes* (extracts) the route values:

- controller = Products,
- action = Details,
- id = 5

This results in a match if the app has a **controller** named ProductsController and an **action** called Details. The *id* value is optional due to the `?`.

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

```
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?}");
});
```

Attribute Routing – REST API's

<https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/routing?view=aspnetcore-3.1#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 sets of **attributes** on each **controller action** to map **actions** directly to route templates. The following **Startup.Configure** code is typical for a *REST API*.

MapControllers() is called inside **UseEndpoints()** to map attribute routed controllers.

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

Attribute Routing – REST API's

<https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/routing?view=aspnetcore-3.1#attribute-routing-for-rest-apis>

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

Conventional Routing handles routes more succinctly, but *Attribute Routing* allows (and requires) precise control over which route templates apply to each *action*.

With *attribute routing*, the *controller* name and *action* names play no role in which *action* is matched.

```
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);
    }
}
```

Attribute Routing - HTTP Verb Templates

<https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/routing?view=aspnetcore-3.1#http-verb-templates>

ASP.NET Core has these *HTTP verb* templates:

[HttpGet], [HttpPost], [HttpPut], [HttpDelete],
[HttpHead], [HttpPatch].

The `GetProduct()` action method includes the `"{id}"` template, therefore id is appended to the `"api/[controller]"` template on the *controller*, so `GetProduct()` template is `"api/test2/{id}"`.

Therefore, this *action* only matches *GET* requests of the form:

- `/api/test2/123`, `/api/test2/{any string}`.

```
[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);
    }
}
```