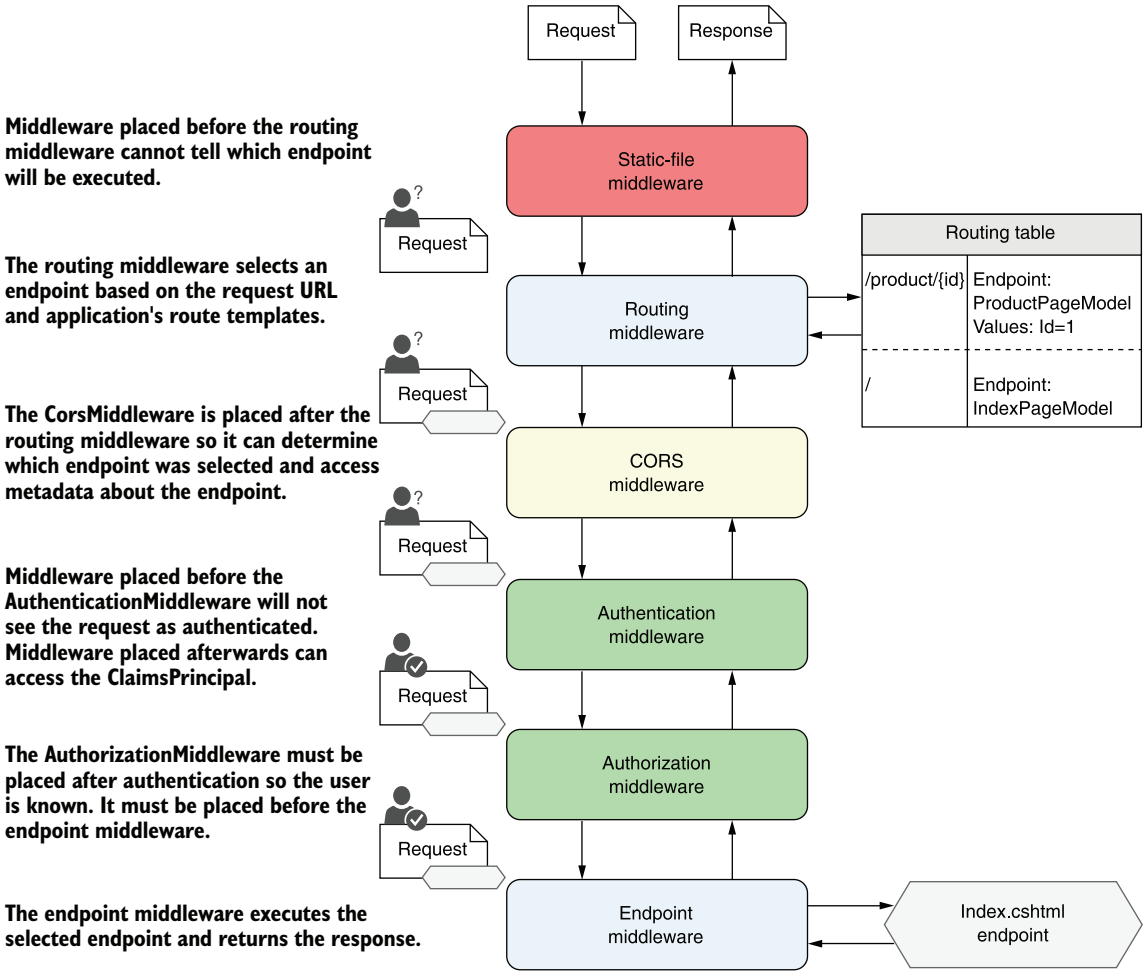
Request
Response
Middleware placed before the routing middleware cannot tell which endpoint will be executed.
Static-file middleware
Request
The routing middleware selects an endpoint based on the request URL and application's route templates.
Routing middleware
Routing table
/product/{id}
Endpoint: ProductPageModel Values: Id=1
/
Endpoint: IndexPageModel
Request
The CorsMiddleware is placed after the routing middleware so it can determine which endpoint was selected and access metadata about the endpoint.
CORS middleware
Request
Middleware placed before the AuthenticationMiddleware will not see the request as authenticated. Middleware placed afterwards can access the ClaimsPrincipal.
Authentication middleware
Request
The AuthorizationMiddleware must be placed after authentication so the user is known. It must be placed before the endpoint middleware.
Authorization middleware
Request
The endpoint middleware executes the selected endpoint and returns the response.
Endpoint middleware
Index.cshtml endpoint

DISCLAIMER:

ReadMe website is intended for academic and demonstration purposes only. We’re only showing a preview of the book to respect the author’s copyright. Thank you for your understanding!

- Group 4: The Classified

ASP.NET Core in Action, Second Edition

Many of the performance improvements made to Kestrel did not come from the ASP.NET team members themselves, but from contributors to the open source project on GitHub.[8] Developing in the open means you typically see fixes and features make their way to production faster than you would for the previous version of ASP.NET, which was dependent on .NET Framework and Windows and, as such, had long release cycles.

In contrast, .NET 5.0, and hence ASP.NET Core, is designed to be released in small increments. Major versions will be released on a predictable cadence, with a new version every year, and a new Long Term Support (LTS) version released every two years.[9] In addition, bug fixes and minor updates can be released as and when they're needed. Additional functionality is provided as NuGet packages, independent of the underlying .NET 5.0 platform.

> **NOTE** NuGet is a package manager for .NET that enables importing libraries into your projects. It's equivalent to Ruby Gems, npm for JavaScript, or Maven for Java.

To enable this approach to releases, ASP.NET Core is highly modular, with as little coupling to other features as possible. This modularity lends itself to a pay-for-play approach to dependencies, where you start with a bare-bones application and only add the additional libraries you require, as opposed to the kitchen-sink approach of previous ASP.NET applications. Even MVC is an optional package! But don't worry, this approach doesn't mean that ASP.NET Core is lacking in features; it means you need to opt in to them. Some of the key infrastructure improvements include

- Middleware "pipeline" for defining your application's behavior
- Built-in support for dependency injection
- Combined UI (MVC) and API (Web API) infrastructure
- Highly extensible configuration system
- Scalable for cloud platforms by default using asynchronous programming

Each of these features was possible in the previous version of ASP.NET but required a fair amount of additional work to set up. With ASP.NET Core, they're all there, ready, and waiting to be connected!

Microsoft fully supports ASP.NET Core, so if you have a new system you want to build, there's no significant reason not to use it. The largest obstacle you're likely to come across is wanting to use programming models that are no longer supported in ASP.NET Core, such as Web Forms or WCF server, as I'll discuss in the next section.

Hopefully, this section has whetted your appetite with some of the many reasons to use ASP.NET Core for building new applications. But if you're an existing ASP.NET

---

[8] The Kestrel HTTP server GitHub project can be found in the ASP.NET Core repository at https://github.com/dotnet/aspnetcore.

[9] The release schedule for .NET 5.0 and beyond: https://devblogs.microsoft.com/dotnet/introducing-net-5/.

As you saw in chapter 2, you define the middleware pipeline in code as part of your initial application configuration in `Startup`. You can tailor the middleware pipeline specifically to your needs—simple apps may need only a short pipeline, whereas large apps with a variety of features may use much more middleware. Middleware is the fundamental source of behavior in your application. Ultimately, the middleware pipeline is responsible for responding to any HTTP requests it receives.

Requests are passed to the middleware pipeline as `HttpContext` objects. As you saw in chapter 2, the ASP.NET Core web server builds an `HttpContext` object from an incoming request, which passes up and down the middleware pipeline. When you're using existing middleware to build a pipeline, this is a detail you'll rarely have to deal with. But, as you'll see in the final section of this chapter, its presence behind the scenes provides a route to exerting extra control over your middleware pipeline.

You can also think of your middleware pipeline as being a series of concentric components, similar to a traditional matryoshka (Russian) doll, as shown in figure 3.3. A request progresses "through" the pipeline by heading deeper into the stack of middleware until a response is returned. The response then returns through the middleware, passing through them in the reverse order to the request.

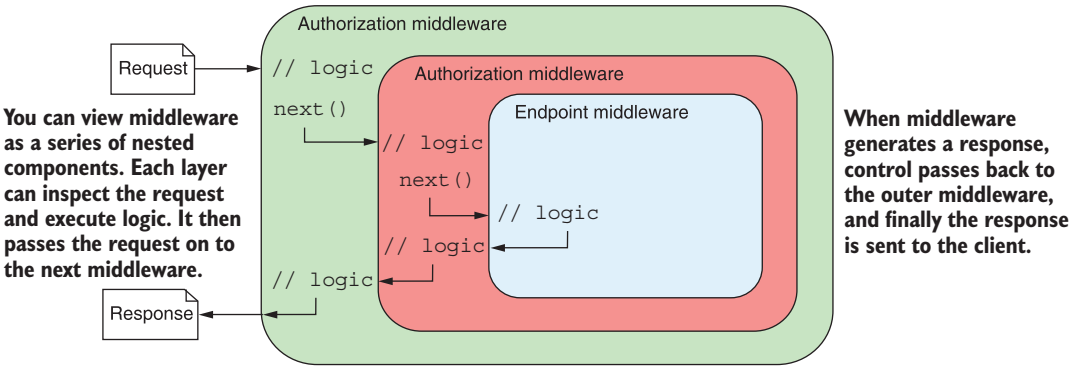

Figure 3.3 You can also think of middleware as being a series of nested components, where a request is sent deeper into the middleware, and the response resurfaces out of it. Each middleware can execute logic before passing the response on to the next middleware and can execute logic after the response has been created, on the way back out of the stack.

Middleware vs. HTTP modules and HTTP handlers

In the previous version of ASP.NET, the concept of a middleware pipeline isn't used. Instead, you have HTTP modules and HTTP handlers.

An *HTTP handler* is a process that runs in response to a request and generates the response. For example, the ASP.NET page handler runs in response to requests for

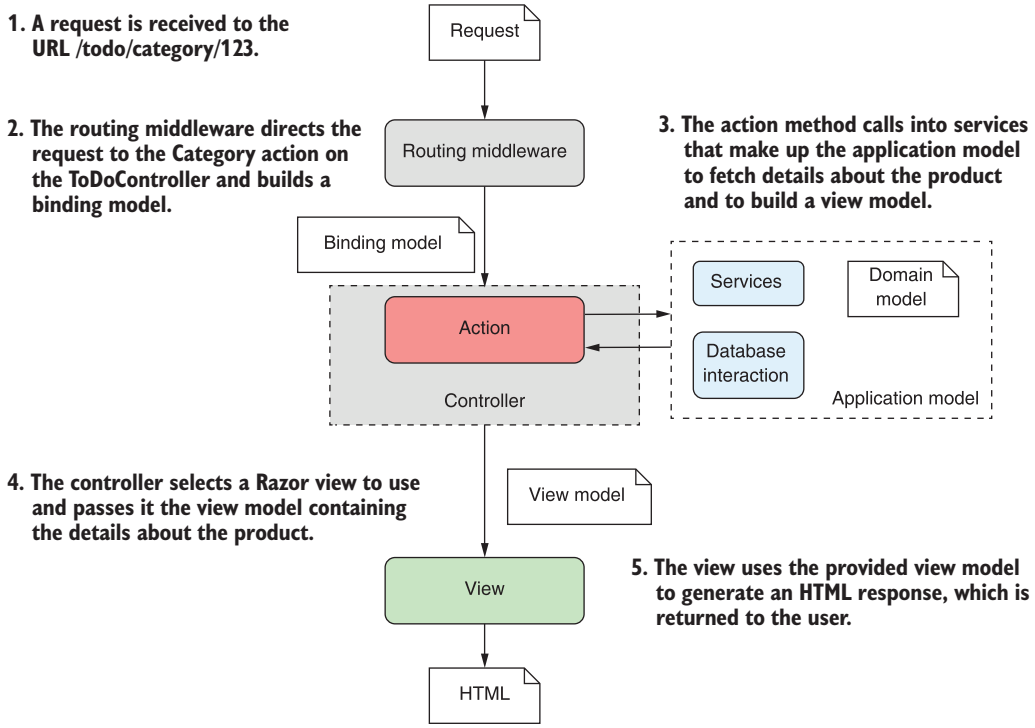

Figure 4.10 A complete MVC controller request for a category. The MVC controller pattern is almost identical to that of Razor Pages, shown in figure 4.6. The controller is equivalent to a Razor Page, and the action is equivalent to a page handler.

Listing 4.2 An MVC controller for viewing all to-do items in a given category

```
public class ToDoController : Controller
{
    private readonly ToDoService _service;
    public ToDoController(ToDoService service)
    {
        _service = service;
    }

    public ActionResult Category(string id)
    {
        var items = _service.GetItemsForCategory(id);
        var viewModel = new CategoryViewModel(items);

        return View(viewModel);
    }
```

The ToDoService is provided in the controller constructor using dependency injection.

The Category action method takes a parameter, id.

The action method calls out to the ToDoService to retrieve data and build a view model.

The view model is a simple C# class, defined elsewhere in your application.

Returns a ViewResult indicating the Razor view should be rendered, passing in the view model

- Don't use route constraints as general input validators. Use them to disambiguate between two similar routes.
- Use a catch-all parameter to capture the remainder of a URL into a route value.
- You can use the routing infrastructure to generate internal URLs for your application.
- The `IUrlHelper` can be used to generate URLs as a string based on an action name or Razor Page.
- You can use the `RedirectToAction` and `RedirectToPage` methods to generate URLs while also generating a redirect response.
- The `LinkGenerator` can be used to generate URLs from other services in your application, where you don't have access to an `HttpContext` object.
- When a Razor Page is executed, a single page handler is invoked based on the HTTP verb of the request and the value of the `handler` route value.
- If there is no page handler for a request, an implicit page handler is used that renders the Razor view.
- You can control the routing conventions used by ASP.NET Core by configuring the `RouteOptions` object, such as to force all URLs to be lowercase, or to always append a trailing slash.
- You can add additional routing conventions for Razor Pages by calling `AddRazorPagesOptions()` after `AddRazorPages()` in Startup.cs. These conventions can control how route parameters are displayed or can add additional route templates for specific Razor Pages.
- Where possible, avoid customizing the route templates for a Razor Page and rely on the conventions instead.

Sometimes you'll want to execute some C#, but you don't need to output the values. We used this technique when we were setting values in `ViewData`:

```
@{
    ViewData["Title"] = "Home Page";
}
```

This example demonstrates a *Razor code block*, which is normal C# code, identified by the `@{}` structure. Nothing is written to the HTML output here; it's all compiled as though you'd written it in any other normal C# file.

> **TIP** When you execute code within code blocks, it must be valid C#, so you need to add semicolons. Conversely, when you're writing values directly to the response using Razor expressions, you don't need them. If your output HTML breaks unexpectedly, keep an eye out for missing or rogue extra semicolons.

Razor expressions are one of the most common ways of writing data from your `PageModel` to the HTML output. You'll see the other approach, using Tag Helpers, in the next chapter. Razor's capabilities extend far further than this, however, as you'll see in the next section, where you'll learn how to include traditional C# structures in your templates.

7.3.2 Adding loops and conditionals to Razor templates

One of the biggest advantages of using Razor templates over static HTML is the ability to dynamically generate the output. Being able to write values from your `PageModel` to the HTML using Razor expressions is a key part of that, but another common use is loops and conditionals. With these, you can hide sections of the UI, or produce HTML for every item in a list, for example.

Loops and conditionals include constructs such as `if` and `for` loops. Using them in Razor templates is almost identical to C#, but you need to prefix their usage with the `@` symbol. In case you're not getting the hang of Razor yet, when in doubt, throw in another `@`!

One of the big advantages of Razor in the context of ASP.NET Core is that it uses languages you're already familiar with: C# and HTML. There's no need to learn a whole new set of primitives for some other templating language: it's the same `if`, `foreach`, and `while` constructs you already know. And when you don't need them, you're writing raw HTML, so you can see exactly what the user will be getting in their browser.

In listing 7.6, I've applied a number of these different techniques in the template for displaying a to-do item. The `PageModel` has a `bool IsComplete` property, as well as a `List<string>` property called `Tasks`, which contains any outstanding tasks.

Listing 7.6 Razor template for rendering a `ToDoItemViewModel`

```
@page
@model ToDoItemModel
```

The @model directive indicates the type of PageModel in Model.

The `ValidationSummary` enum controls which values are displayed, and it has three possible values:

- `None`—Don't display a summary. (I don't know why you'd use this.)
- `ModelOnly`—Only display errors that are *not* associated with a property.
- `All`—Display errors either associated with a property or with the model.

The Validation Summary Tag Helper is particularly useful if you have errors associated with your page that aren't specific to a single property. These can be added to the model state by using a blank key, as shown in listing 8.8. In this example, the property validation passed, but we provide additional model-level validation to check that we aren't trying to convert a currency to itself.

Listing 8.8 Adding model-level validation errors to the `ModelState`

```
public class ConvertModel : PageModel
{
    [BindProperty]
    public InputModel Input { get; set; }

    [HttpPost]
    public IActionResult OnPost()
    {
        if(Input.CurrencyFrom == Input.CurrencyTo)
        {
            ModelState.AddModelError(
                string.Empty,
                "Cannot convert currency to itself");
        }
        if (!ModelState.IsValid)
        {
            return Page();
        }

        //store the valid values somewhere etc
        return RedirectToPage("Checkout");
    }
}
```

Can't convert currency to itself

Adds model-level error, not tied to a specific property, by using empty key

If there are any property-level or model-level errors, display them.

Without the Validation Summary Tag Helper, the model-level error would still be added if the user used the same currency twice, and the form would be redisplayed. Unfortunately, there would have been no visual cue to the user indicating why the form did not submit—obviously that's a problem! By adding the Validation Summary Tag Helper, the model-level errors are shown to the user so they can correct the problem, as shown in figure 8.10.

NOTE For simplicity, I added the validation check to the page handler. A better approach might be to create a custom validation attribute to achieve this instead. That way, your handler stays lean and sticks to the single responsibility principle. You'll see how to achieve this in chapter 20.

10.1.1 Understanding the benefits of dependency injection

When you first started programming, the chances are you didn't immediately use a DI framework. That's not surprising, or even a bad thing; DI adds a certain amount of extra wiring that's often not warranted in simple applications or when you're getting started. But when things start to get more complex, DI comes into its own as a great tool to help keep that complexity in check.

Let's consider a simple example, written without any sort of DI. Imagine a user has registered on your web app and you want to send them an email. This listing shows how you might approach this initially in an API controller.

> **NOTE** I'm using an API controller for this example, but I could just as easily have used a Razor Page. Razor Pages and API controllers both use constructor dependency injection, as you'll see in section 10.2.

Listing 10.1 Sending an email without DI when there are no dependencies

```
public class UserController : ControllerBase
{
    [HttpPost("register")]
    public IActionResult RegisterUser(string username)
    {
        var emailSender = new EmailSender();
        emailSender.SendEmail(username);
        return Ok();
    }
}
```

The action method is called when a new user is created.

Creates a new instance of EmailSender

Uses the new instance to send the email

In this example, the `RegisterUser` action on `UserController` executes when a new user registers on your app. This creates a new instance of an `EmailSender` class and calls `SendEmail()` to send the email. The `EmailSender` class is the one that does the sending of the email. For the purposes of this example, you can imagine it looks something like this:

```
public class EmailSender
{
    public void SendEmail(string username)
    {
        Console.WriteLine($"Email sent to {username}!");
    }
}
```

`Console.Writeline` stands in here for the real process of sending the email.

> **NOTE** Although I'm using sending email as a simple example, in practice you might want to move this code out of your Razor Page and controller classes entirely. This type of asynchronous task is well suited to using message queues and a background process. For more details, see http://mng.bz/pVWR.

```
    {
        config.Sources.Clear();
        config.AddJsonFile("appsettings.json", optional: true);
    }
```

Clears the providers configured by default in CreateDefaultBuilder

Adds a JSON configuration provider, providing the filename of the configuration file

TIP In listing 11.5 I extracted the configuration to a static helper method, `AddAppConfiguration`, but you can also provide this inline as a lambda method.

The `HostBuilder` creates a `ConfigurationBuilder` instance before invoking the `ConfigureAppConfiguration` method. All you need to do is add the configuration providers for your application.

In this example, you've added a single JSON configuration provider by calling the `AddJsonFile` extension method and providing a filename. You've also set the value of `optional` to `true`. This tells the configuration provider to skip over files it can't find at runtime, instead of throwing `FileNotFoundException`. Note that the extension method just registers the provider at this point; it doesn't try to load the file yet.

And that's it! The `HostBuilder` instance takes care of calling `Build()`, which generates `IConfiguration`, which represents your configuration object. This is then registered with the DI container, so you can inject it into your classes. You'd commonly inject this into the constructor of your `Startup` class, so you can use it in the `Configure` and `ConfigureServices` methods:

```
public class Startup
{
    public Startup(IConfiguration config)
    {
        Configuration = config;
    }
    public IConfiguration Configuration { get; }
}
```

NOTE The `ConfigurationBuilder` creates an `IConfigurationRoot` instance, which implements `IConfiguration`. This is registered as an `IConfiguration` in the DI container, *not* an `IConfigurationRoot`. `IConfiguration` is one of the few things that you can inject into the `Startup` constructor.

At this point, at the end of the `Startup` constructor, you have a fully loaded configuration object. But what can you do with it? `IConfiguration` stores configuration as a set of key-value `string` pairs. You can access any value by its key, using standard dictionary syntax. For example, you could use

```
var zoomLevel = Configuration["MapSettings:DefaultZoomLevel"];
```

to retrieve the configured zoom level for your application. Note that I used a colon (`:`) to designate a separate section. Similarly, to retrieve the `latitude` key, you could use

```
Configuration["MapSettings:DefaultLocation:Latitude"];
```

APIs from the underlying business logic. This will often make your business logic easier to test and more reusable.

Our database doesn't have any data in it yet, so we'd better start by letting you create a recipe.

12.4.1 Creating a record

In this section you're going to build the functionality to let users create a recipe in the app. This will primarily consist of a form that the user can use to enter all the details of the recipe using Razor Tag Helpers, which you learned about in chapters 7 and 8. This form is posted to the Create.cshtml Razor Page, which uses model binding and validation attributes to confirm the request is valid, as you saw in chapter 6.

If the request is valid, the page handler calls `RecipeService` to create the new `Recipe` object in the database. As EF Core is the topic of this chapter, I'm going to focus on this service alone, but you can always check out the source code for this book if you want to see how everything fits together.

The business logic for creating a recipe in this application is simple—there *is* no logic! Map the *command* binding model provided in the Create.cshtml Razor Page to a `Recipe` entity and its `Ingredients`, add the `Recipe` object to `AppDbContext`, and save that in the database, as shown in figure 12.11.

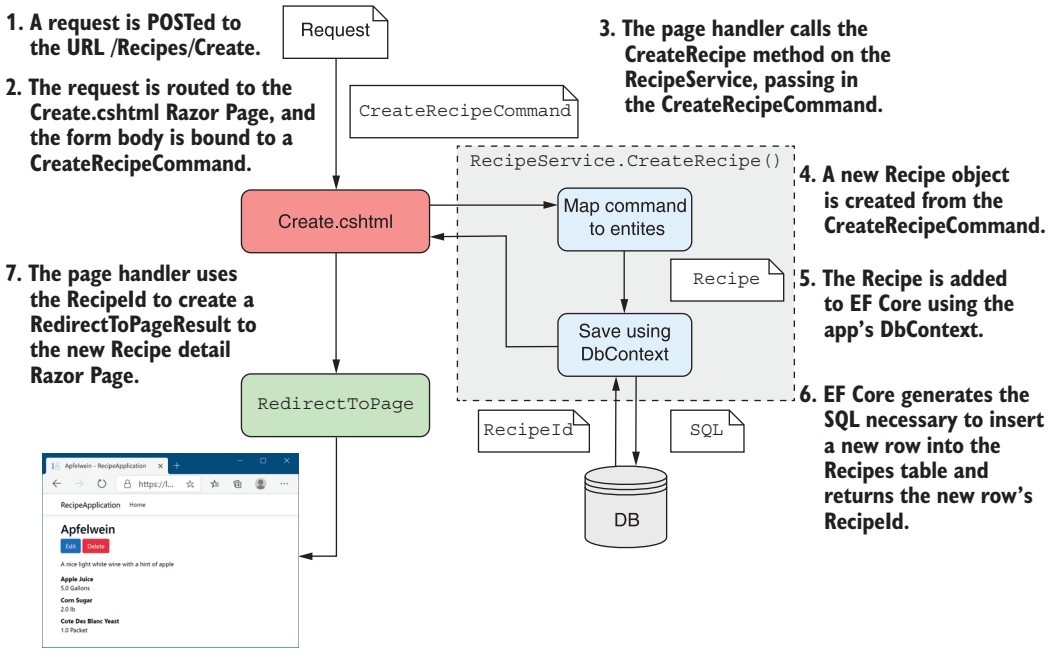

Figure 12.11 Calling the Create.cshtml Razor Page and creating a new entity. A `Recipe` is created from the `CreateRecipeCommand` binding model and is added to the `DbContext`. EF Core generates the SQL to add a new row to the Recipes table in the database.

you can ensure that other filters, such as result filters that define the output format, run as usual, even when your action filters short-circuit.

The last thing I'd like to talk about in this chapter is how to use DI with your filters. You saw in chapter 10 that DI is integral to ASP.NET Core, and in the next section you'll see how to design your filters so that the framework can inject service dependencies into them for you.

13.4 Using dependency injection with filter attributes

In this section you'll learn how to inject services into your filters so you can take advantage of the simplicity of DI in your filters. You'll learn to use two helper filters to achieve this, `TypeFilterAttribute` and `ServiceFilterAttribute`, and you'll see how they can be used to simplify the action filter you defined in section 13.2.3.

The previous version of ASP.NET used filters, but they suffered from one problem in particular: it was hard to use services from them. This was a fundamental issue with implementing them as attributes that you decorate your actions with. C# attributes don't let you pass dependencies into their constructors (other than constant values), and they're created as singletons, so there's only a single instance for the lifetime of your app.

In ASP.NET Core, this limitation is still there in general, in that filters are typically created as attributes that you add to your controller classes, action methods, and Razor Pages. What happens if you need to access a transient or scoped service from inside the singleton attribute?

Listing 13.13 showed one way of doing this, using a pseudo-service locator pattern to reach into the DI container and pluck out `RecipeService` at runtime. This works but is generally frowned upon as a pattern, in favor of proper DI. How can you add DI to your filters?

The key is to split the filter into two. Instead of creating a class that's both an attribute and a filter, create a filter class that contains the functionality and an attribute that tells the framework when and where to use the filter.

Let's apply this to the action filter from listing 13.13. Previously I derived from `ActionFilterAttribute` and obtained an instance of `RecipeService` from the `context` passed to the method. In the following listing, I show two classes, `EnsureRecipeExistsFilter` and `EnsureRecipeExistsAttribute`. The filter class is responsible for the functionality and takes in `RecipeService` as a constructor dependency.

Listing 13.18 Using DI in a filter by not deriving from `Attribute`

```
public class EnsureRecipeExistsFilter : IActionFilter
{
    private readonly RecipeService _service;
    public EnsureRecipeExistsFilter(RecipeService service)
    {
        _service = service;
    }
```

Doesn't derive from an Attribute class

RecipeService is injected into the constructor.

Listing 15.1 Applying [Authorize] to an action

```
public class RecipeApiController : ControllerBase
{
    public IActionResult List()
    {
        return Ok();
    }

    [Authorize]
    public IActionResult View()
    {
        return Ok();
    }
}
```

This action can be executed by anyone, even when not logged in.

Applies [Authorize] to individual actions, whole controllers, or Razor Pages

This action can only be executed by authenticated users.

Applying the [Authorize] attribute to an endpoint attaches metadata to it, indicating only authenticated users may access the endpoint. As you saw in figure 15.2, this metadata is made available to the AuthorizationMiddleware when an endpoint is selected by the RoutingMiddleware.

You can apply the [Authorize] attribute at the action scope, controller scope, Razor Page scope, or globally, as you saw in chapter 13. Any action or Razor Page that has the [Authorize] attribute applied in this way can be executed only by an authenticated user. Unauthenticated users will be redirected to the login page.

TIP There are several different ways to apply the [Authorize] attribute globally. You can read about the different options, and when to choose which option, on my blog: http://mng.bz/opQp.

Sometimes, especially when you apply the [Authorize] attribute globally, you might need to poke holes in this authorization requirement. If you apply the [Authorize] attribute globally, then any unauthenticated request will be redirected to the login page for your app. But if the [Authorize] attribute is global, then when the login page tries to load, you'll be unauthenticated and redirected to the login page again. And now you're stuck in an infinite redirect loop.

To get around this, you can designate specific endpoints to ignore the [Authorize] attribute by applying the [AllowAnonymous] attribute to an action or Razor Page, as shown next. This allows unauthenticated users to execute the action, so you can avoid the redirect loop that would otherwise result.

Listing 15.2 Applying [AllowAnonymous] to allow unauthenticated access

```
[Authorize]
public class AccountController : ControllerBase
{
    public IActionResult ManageAccount()
    {
        return Ok();
    }
```

Applied at the controller scope, so the user must be authenticated for all actions on the controller.

Only authenticated users may execute ManageAccount.

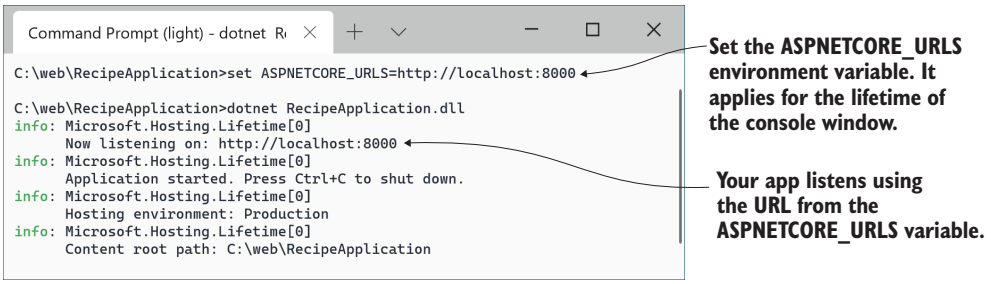

Figure 16.10 Change the `ASPNETCORE_URLS` environment variable to change the URL used by ASP.NET Core apps.

You can instruct an app to listen on multiple URLs by separating them with a semicolon, or you can listen to a specific port without specifying the localhost hostname. If you set the `ASPNETCORE_URLS` environment variable to

```
http://localhost:5001;http://*:5002
```

your ASP.NET Core apps will listen for requests sent to the following:

- http://localhost:5001—This address is only accessible on your local computer, so it will not accept requests from the wider internet.
- http://*:5002—Any URL on port 5002. External requests from the internet can access the app on port 5002, using any URL that maps to your computer.

Note that you *can't* specify a different hostname, like tastyrecipes.com, for example. ASP.NET Core will listen to all requests on a given port. The exception is the localhost hostname, which only allows requests that came from your own computer.

> **NOTE** If you find the `ASPNETCORE_URLS` variable isn't working properly, ensure you don't have a launchSettings.json file in the directory. When present, the values in this file take precedence. By default, launchSettings.json isn't included in the `publish` output, so this generally won't be an issue in production.

Setting the URL of an app using a single environment variable works great for some scenarios, most notably when you're running a single application in a virtual machine, or within a Docker container.[6]

If you're not using Docker containers, the chances are you're hosting multiple apps side-by-side on the same machine. A single environment variable is no good for setting URLs in this case, as it would change the URL of every app.

[6] ASP.NET Core is well-suited to running in containers, but working with containers is a separate book in its own right. For details on hosting and publishing apps using Docker, see Microsoft's "Host ASP.NET Core in Docker containers" documentation: http://mng.bz/e5GV.

The scope state can be any object at all: an `int`, a `string`, or a `Dictionary`, for example. It's up to each logging provider implementation to decide how to handle the state you provide in the `BeginScope` call, but typically it will be serialized using `ToString()`.

> **TIP** The most common use for scopes I've found is to attach additional key-value pairs to logs. To achieve this behavior in Seq and Serilog, you need to pass `Dictionary<string, object>` as the state object.[6]

When the log messages inside the scope block are written, the scope state is captured and written as part of the log, as shown in figure 17.15. The `Dictionary<>` of key-value pairs is added directly to the log message (`CustomValue1`), and the remaining state values are added to the `Scope` property. You will likely find the dictionary approach the more useful of the two, as the added properties are more easily filtered on, as you saw in figure 17.14.

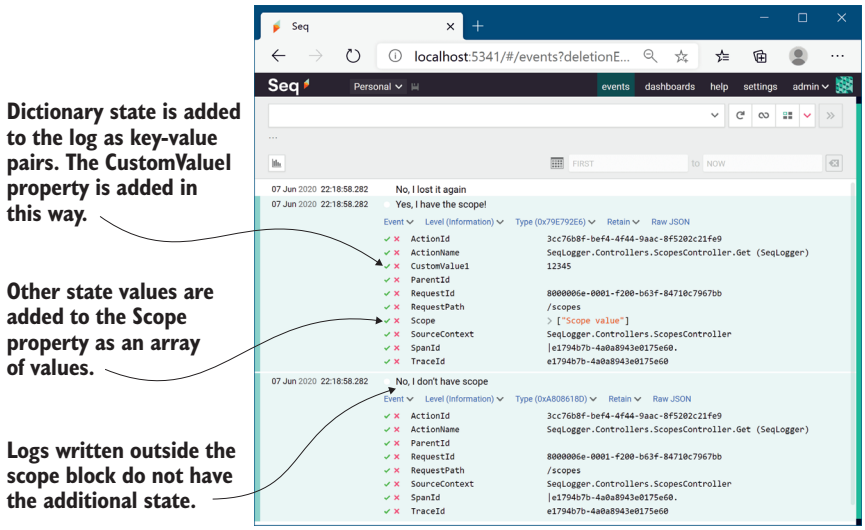

Figure 17.15 Adding properties to logs using scopes. Scope state added using the dictionary approach is added as structured logging properties, but other state is added to the `Scope` property. Adding properties makes it easier to associate related logs with one another.

That brings us to the end of this chapter on logging. Whether you use the built-in logging providers or opt to use a third-party provider like Serilog or NLog, ASP.NET Core makes it easy to get detailed logs not only for your app code, but for the libraries

[6] Nicholas Blumhardt, the creator of Serilog and Seq, has examples and the reasoning for this on his blog in the "The semantics of ILogger.BeginScope()" article: http://mng.bz/GxDD.

TIP In section 19.2 you'll see how to create endpoints that use the endpoint routing system.

One situation where `Map` can be useful is when you want to have two "independent" sub-applications but don't want the hassle of multiple deployments. You can use `Map` to keep these pipelines separate, with separate routing and endpoints inside each branch of the pipeline. Just be aware that these branches will both share the same configuration and DI container, so they're only independent from the middleware pipeline's point of view.[1]

The final point you should be aware of when using the `Map` extension is that it modifies the effective `Path` seen by middleware on the branch. When it matches a URL prefix, the `Map` extension cuts off the matched segment from the path, as shown in figure 19.2. The removed segments are stored on a property of `HttpContext` called `PathBase`, so they're still accessible if you need them.

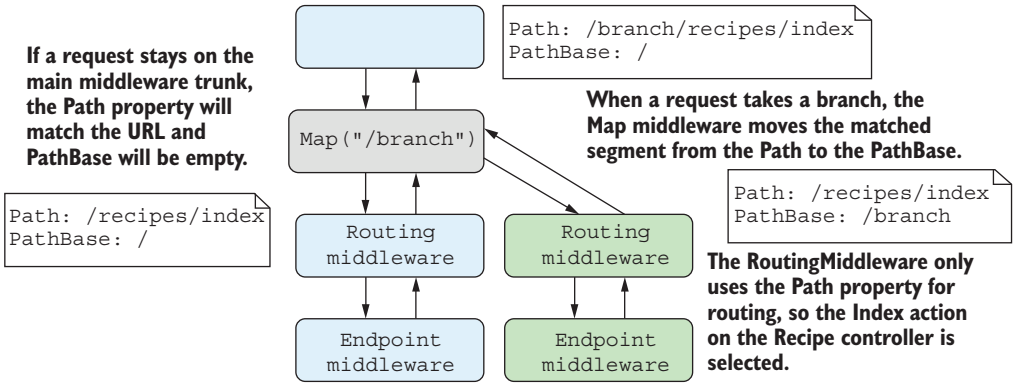

Figure 19.2 When the `Map` extension diverts a request to a branch, it removes the matched segment from the `Path` property and adds it to the `PathBase` property.

NOTE ASP.NET Core's link generator (used in Razor, for example, as discussed in chapter 5) uses `PathBase` to ensure it generates URLs that include the `PathBase` as a prefix.

You've seen the `Run` extension, which always returns a response, and the `Map` extension which creates a branch in the pipeline. The next extension we'll look at is the general-purpose `Use` extension.

[1] Achieving truly independent branches in the same application requires a lot more effort. See Filip W's blog post, "Running multiple independent ASP.NET Core pipelines side by side in the same application," for guidance: http://mng.bz/vzA4.

```
            .Length(3)
            .Must(value => _allowedValues.Contains(value))
            .WithMessage("Not a valid currency code");

        RuleFor(x => x.Quantity)
            .NotNull()
            .InclusiveBetween(1, 1000);
    }
}
```

Thanks to strong typing, the rules available depend on the property being validated.

Your first impression of this code might be that it's quite verbose compared to listing 20.7 but remember that listing 20.7 used a custom validation attribute, `[CurrencyCode]`. The validation in listing 20.11 doesn't require anything else—the logic implemented by the `[CurrencyCode]` attribute is right there in the validator, making it easy to reason about. The `Must()` method can be used to perform arbitrarily complex validations, without having the additional layers of indirection required by custom `DataAnnotations` attributes.

On top of that, you'll notice that you can only define validation rules that make sense for the property being validated. Previously, there was nothing to stop us applying the `[CurrencyCode]` attribute to the `Quantity` property; that's just not possible with FluentValidation.

Of course, just because you *can* write the custom `[CurrencyCode]` logic in-line doesn't necessarily mean you have to. If a rule is used in multiple parts of your application, it may make sense to extract it into a helper class. The following listing shows how you could extract the currency code logic into an extension method, which can be used in multiple validators.

Listing 20.12 An extension method for currency validation

```
public static class ValidationExtensions
{
    public static IRuleBuilderOptions<T, string>
        MustBeCurrencyCode<T>(
            this IRuleBuilder<T, string> ruleBuilder)
    {
        return ruleBuilder
            .Must(value => _allowedValues.Contains(value))
            .WithMessage("Not a valid currency code");
    }

    private static readonly string[] _allowedValues =
        new []{ "GBP", "USD", "CAD", "EUR" };
}
```

Creates an extension method that can be chained from RuleFor() for string properties

Applies the same validation logic as before

The currency code values to allow

You can then update your `CurrencyConverterModelValidator` to use the new extension method, removing the duplication in your validator and ensuring consistency across country-code fields:

```
        var latestRates = await _typedClient.GetLatestRatesAsync();

        _dbContext.Add(latestRates);
        await _dbContext.SaveChangesAsync();

        _logger.LogInformation("Latest rates updated");
    }
}
```

Download the rates from the remote API.

Save the rates to the database.

Functionally, the `IJob` in the listing 22.9 is doing a similar task to the `BackgroundService` implementation in listing 22.4, with a few notable exceptions:

- *The `IJob` only defines the task to execute; it doesn't define timing information.* In the `BackgroundService` implementation, we also had to control how often the task was executed.
- *A new `IJob` instance is created every time the job is executed.* In contrast, the `BackgroundService` implementation is only created once, and its `Execute` method is only invoked once.
- *We can inject scoped dependencies directly into the `IJob` implementation.* To use scoped dependencies in the `IHostedService` implementation, we had to manually create our own scope and use service location to load dependencies. Quartz.NET takes care of that for us, allowing us to use pure constructor injection. Every time the job is executed, a new scope is created and is used to create a new instance of the `IJob`.

The `IJob` defines *what* to execute, but it doesn't define *when* to execute it. For that, Quartz.NET uses *triggers*. Triggers can be used to define arbitrarily complex blocks of time during which a job should be executed. For example, you can specify start and end times, how many times to repeat, and blocks of time when a job should or shouldn't run (such as only 9 a.m. to 5 p.m., Monday–Friday).

In the following listing, we register the `UpdateExchangeRatesJob` with the DI container using the `AddJob<T>()` method, and we provide a unique name to identify the job. We also configure a trigger that fires immediately, and then every five minutes, until the application shuts down.

Listing 22.10 Configuring a Quartz.NET `IJob` and trigger

```
public void ConfigureServices(IService collection)
{
    services.AddQuartz(q =>
    {
        q.UseMicrosoftDependencyInjectionScopedJobFactory();

        var jobKey = new JobKey("Update exchange rates");
        q.AddJob<UpdateExchangeRatesJob>(opts =>
            opts.WithIdentity(jobKey));

        q.AddTrigger(opts => opts
            .ForJob(jobKey)
```

Create a unique key for the job, used to associate it with a trigger.

Add the IJob to the DI container, and associate it with the job key.

Register a trigger for the IJob via the job key.

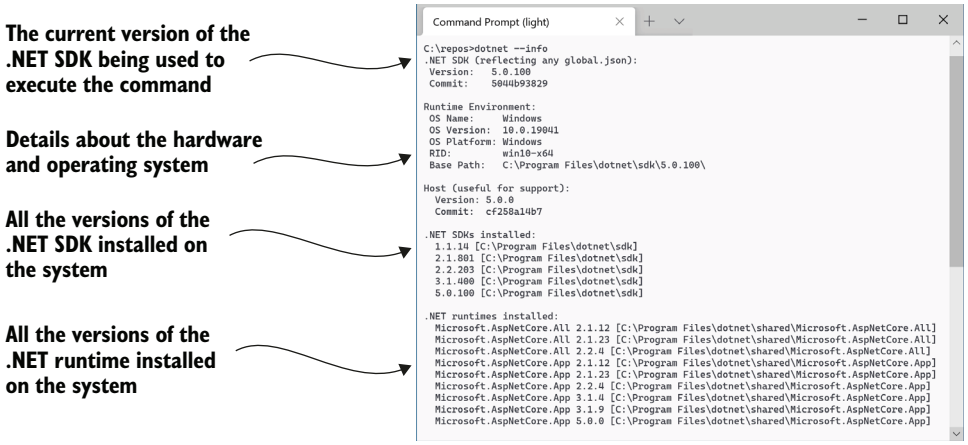

Figure A.1 Use `dotnet --info` to check which version of the .NET SDK is currently used and which versions are available. This screenshot shows I am currently using the release version of the .NET 5 SDK, version 5.0.100.

> **TIP** Some IDEs, such as Visual Studio, can automatically install .NET 5.0 as part of their installation process. There is no problem installing multiple versions of .NET Core and .NET 5.0 side by side, so you can always install the .NET SDK manually, whether your IDE installs a different version or not.

By default, when you run `dotnet` commands from the command line, you'll be using the latest version of the .NET SDK you have installed. You can control that and use an older version of the SDK by adding a global.json file to the folder. For an introduction to this file, how to use it, and understanding .NET's versioning system, see my blog entry, "Exploring the new rollForward and allowPrerelease settings in global.json": http://mng.bz/KMzP.

Once you have the .NET SDK installed, it's time to choose an IDE or editor. The choices available will depend on which operating system you're using and will largely be driven by personal preference.

A.2 Choosing an IDE or editor

In this section I'll describe a few of the most popular IDEs and editors for .NET development and how to install them. Choosing an IDE is a very personal choice, so this section only describes a few of the options. If your favorite IDE isn't listed here, check the documentation to see if .NET is supported.

A.2.1 Visual Studio (Windows)

For a long time, Windows has been the best system for building .NET applications, and with the availability of Visual Studio that's arguably still the case.

Visual Studio (figure A.2) is a full-featured IDE that provides one of the best all-around experiences for developing ASP.NET Core applications. Luckily, the Visual