After making all these changes, you'll have the best of both worlds—you can update the default UI Razor Pages HTML, without taking on the responsibility of maintaining the default UI code behind.

> **TIP** In the source code for the book you can see these changes in action, where the Register view has been customized to remove the references to external identity providers.

Unfortunately, it's not always possible to use the default UI `PageModel`. Sometimes, you *need* to update the page handlers, such as when you want to change the functionality of your Identity area, rather than just the look and feel. A common requirement is needing to store additional information about a user, as you'll see in the next section.

14.6 Managing users: adding custom data to users

In this section you'll see how to customize the `ClaimsPrincipal` assigned to your users by adding additional claims to the AspNetUserClaims table when the user is created. You'll also see how to access these claims in your Razor Pages and templates.

Very often, the next step after adding Identity to an application is to customize it. The default templates only require an email and password to register. What if you need more details, like a friendly name for the user? Also, I've mentioned that we use claims for security, so what if you want to add a claim called `IsAdmin` to certain users?

You know that every user principal has a collection of claims so, conceptually, adding any claim just requires adding it to the user's collection. There are two main times that you would want to grant a claim to a user:

- *For every user, when they first register on the app.* For example, you might want to add a "Name" field to the Register form and add that as a claim to the user when they register.
- *Manually, after the user has already registered.* This is common for claims used as "permissions," where an existing user might want to add an `IsAdmin` claim to a specific user after they have registered on the app.

In this section, I show you the first approach, automatically adding new claims to a user when they're created. The latter approach is the more flexible and, ultimately, is the approach many apps will need, especially line-of-business apps. Luckily, there's nothing conceptually difficult to it; it requires a simple UI that lets you view users and add a claim through the same mechanism I'll show here.

> **TIP** Another common approach is to customize the `IdentityUser` entity, by adding a `Name` property for example. This approach is sometimes easier to work with if you want to give users the ability to edit that property. The documentation describes the steps required to achieve that: https://docs.microsoft.com/aspnet/core/security/authentication/add-user-data.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Let's say you want to add a new `Claim` to a user called `FullName`. A typical approach would be:

1. Scaffold the Register.cshtml Razor Page, as you did in section 14.5.
2. Add a "Name" field to the `InputModel` in the Register.cshtml.cs `PageModel`.
3. Add a "Name" input field to the Register.cshtml Razor view template.
4. Create the new `ApplicationUser` entity as before in the `OnPost()` page handler, by calling `CreateAsync` on `UserManager<ApplicationUser>`.
5. Add a new `Claim` to the user by calling `UserManager.AddClaimAsync()`.
6. Continue the method as before, sending a confirmation email, or signing the user in if email confirmation is not required.

Steps 1-3 are fairly self-explanatory and just require updating the existing templates with the new field. Steps 4-6 all take place in Register.cshtml.cs in the `OnPost()` page handler, which is summarized in the following listing. In practice the page handler has more error checking and boilerplate; our focus here is on the additional lines that add the extra `Claim` to the `ApplicationUser`.

Listing 14.8 Adding a custom claim to a new user in the Register.cshtml.cs page

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser {                       #A
            UserName = Input.Email, Email = Input.Email };     #A
        var result = await _userManager.CreateAsync(           #B
            user, Input.Password);                             #B
        if (result.Succeeded)
        {
            var claim = new Claim("FullName", Input.Name);     #C
            await _userManager.AddClaimAsync(user, claim);     #D

            var code = await _userManager                       #E
                .GenerateEmailConfirmationTokenAsync(user);     #E
            await _emailSender.SendEmailAsync(                  #E
                 Input.Email, "Confirm your email", code );     #E
            await _signInManager.SignInAsync(user);             #F
            return LocalRedirect(returnUrl);
        }
        foreach (var error in result.Errors)                   #G
        {                                                      #G
            ModelState.AddModelError(                          #G
                string.Empty, error.Description);              #G
        }                                                      #G
    }
       return Page();                                             #G
}
```

#A Creates an instance of the ApplicationUser entity, as usual
#B Validates that the provided password is valid and creates the user in the database
#C Creates a claim, with a string name of "FullName" and the provided value
#D Adds the new claim to the ApplicationUser's collection

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

#E Sends a confirmation email to the user, if you have configured the email sender
#F Signs the user in by setting the HttpContext.User, the principal will include the custom claim
#G There was a problem creating the user. Add the errors to the ModelState, and redisplay the page

This is all that's required to *add* the new claim, but you're not *using* it anywhere currently. What if you want to display it? Well, you've added a claim to the `ClaimsPrincipal`, which was assigned to the `HttpContext.User` property when you called `SignInAsync`. That means you can retrieve the claims anywhere you have access to the `ClaimsPrincipal`—including in your page handlers and in view templates. For example, you could display the user's `FullName` claim anywhere in a Razor template with the following statement:

```
@User.Claims.FirstOrDefault(x=>x.Type == "FullName")?.Value
```

This finds the first claim on the current user principal with a `Type` of `"FullName"` and prints the assigned value (if the claim is not found, it prints nothing). The Identity system even includes a handy extension method that tidies up this LINQ expression (found in the `System.Security.Claims` namespace):

```
@User.FindFirstValue("FullName")
```

And with that last tidbit, we've reached the end of this chapter on ASP.NET Core Identity. I hope you've come to appreciate the amount of effort using Identity can save you, especially when you make use of the default Identity UI package.

Adding user accounts and authentication to an app is typically the first step to customizing your app further. Once you have authentication, you can have authorization, which lets you lock down certain actions in your app, based on the current user. In the next chapter, you'll learn about the ASP.NET Core authorization system and how you can use it to customize your apps, in particular, the recipe application, which is coming along nicely!

14.7 Summary

- Authentication is the process of determining who you are and authorization is the process of determining what you're allowed to do. You need to authenticate users before you can apply authorization.
- Every request in ASP.NET Core is associated with a user, also known as a principal. By default, without authentication, this is an anonymous user. You can use the claims principal to behave differently depending on who made a request.
- The current pr1incipal for a request is exposed on `HttpContext.User`. You can access this value from your Razor Pages and views to find out properties of the user such as their, ID, Name or Email.
- Every user has a collection of claims. These claims are single pieces of information about the user. Claims could be properties of the physical user, such as `Name` and `Email`, or they could be related to things the user has, such as `HasAdminAccess` or `IsVipCustomer`.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

- Earlier versions of ASP.NET used roles instead of claims. You can still use roles if you need to, but you should use claims where possible.
- Authentication in ASP.NET Core is provided by `AuthenticationMiddleware` and a number of authentication services. These services are responsible for setting the current principal when a user logs in, saving it to a cookie, and loading the principal from the cookie on subsequent requests.
- The `AuthenticationMiddleware` is added by calling `UseAuthentication()` in your middleware pipeline. This must be placed after the call to `UseRouting()` and before `UseAuthorization()` and `UseEndpoints()`.
- ASP.NET Core includes support for consuming bearer tokens for authenticating API calls and includes helper libraries for configuring IdentityServer. For more details see https://docs.microsoft.com/aspnet/core/security/authentication/identity-api-authorization.
- ASP.NET Core Identity handles low-level services needed for storing users in a database, ensuring their passwords are stored safely, and for logging users in and out. You must provide the UI for the functionality yourself and wire it up to the Identity sub-system.
- The Microsoft.AspNetCore.Identity.UI package provides a default UI for the Identity system, and includes email confirmation, 2FA, and external login provider support. You need to do some additional configuration to enable these features.
- The default template for a Web Application with Individual Account Authentication uses ASP.NET Core Identity to store users in the database with EF Core. It includes all the boilerplate code required to wire the UI up to the Identity system.
- You can use the `UserManager<T>` class to create new user accounts, load them from the database, and change their passwords. `SignInManager<T>` is used to sign a user in and out, by assigning the principal for the request and by setting an authentication cookie. The default UI uses these classes for you to facilitate user registration and login.
- You can update an EF Core `DbContext` to support Identity by deriving from `IdentityDbContext<TUser>` where `TUser` is a class that derives from `IdentityUser`.
- You can add additional claims to a user using the `UserManager<TUser>.AddClaimAsync(TUser user, Claim claim)` method. These claims are added to the `HttpContext.User` object when the user logs in to your app.
- Claims consist of a type and a value. Both values are strings. You can use standard values for types exposed on the `ClaimTypes` class, such as `ClaimTypes .GivenName` and `ClaimTypes.FirstName`, or you can use a custom string, such as `"FullName"`.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

15

Authorization: securing your application

This chapter covers

- **Using authorization to control who can use your app**
- **Using claims-based authorization with policies**
- **Creating custom policies to handle complex requirements**
- **Authorizing a request depending upon the resource being accessed**
- **Hiding elements from a Razor template that the user is unauthorized to access**

In chapter 14, I showed how to add users to an ASP.NET Core application by adding authentication. With authentication, users can register and log in to your app using an email and password. Whenever you add authentication to an app, you inevitably find you want to be able to restrict what some users can do. The process of determining whether a user can perform a given action on your app is called *authorization*.

On an e-commerce site, for example, you may have admin users who are allowed to add new products and change prices, sales users who are allowed to view completed orders, and customer users who are only allowed to place orders and buy products.

In this chapter, I show how to use authorization in an app to control what your users can do. In section 15.1, I introduce authorization and put it in the context of a real-life scenario you've probably experienced: an airport. I describe the sequence of events, from checking in, passing through security, to entering an airport lounge, and how these relate to the authorization concepts you'll see in this chapter.

In section 15.2, I show how authorization fits into an ASP.NET Core web application and how it relates to the `ClaimsPrincipal` class that you saw in the previous chapter. You'll see

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

how to enforce the simplest level of authorization in an ASP.NET Core app, ensuring that only authenticated users can execute a Razor Page or MVC action.

We'll extend that approach in section 15.3 by adding in the concept of policies. These let you make specific requirements about a given authenticated user, requiring that they have specific pieces of information in order to execute an action or Razor Page.

You'll use policies extensively in the ASP.NET Core authorization system, so in section 15.4, we explore how to handle more complex scenarios. You'll learn about authorization requirements and handlers, and how you can combine them to create specific policies that you can apply to your Razor Pages and actions.

Sometimes, whether a user is authorized depends on which resource or document they're attempting to access. A resource is anything that you're trying to protect, so it could be a document or a post in a social media app for example.

For example, you may allow users to create documents, or to read documents from other users, but only to edit documents that they created themselves. This type of authorization, where you need the details of the document to determine if the user is authorized, is called resource-based authorization, and is the focus of section 15.5.

In the final section of this chapter, I show how you can extend the resource-based authorization approach to your Razor view templates. This lets you modify the UI to hide elements that users aren't authorized to interact with. In particular, you'll see how to hide the Edit button when a user isn't authorized to edit the entity.

We'll start by looking more closely at the concept of authorization, how it differs from authentication, and how it relates to real-life concepts you might see in an airport.

15.1 Introduction to authorization

In this section, I provide an introduction to authorization, and how it compares to authentication. I use the real-life example of an airport as a case study to understand how claims-based authorization works.

For people who are new to web apps and security, authentication and authorization can sometimes be a little daunting. It certainly doesn't help that the words look so similar! The two concepts are often used together, but they're definitely distinct:

- *Authentication*—The process of determining who made a request
- *Authorization*—The process of determining whether the requested action is allowed

Typically, authentication occurs first, so that you know who is making a request to your app. For traditional web apps, your app authenticates a request by checking the encrypted cookie that was set when the user logged in (as you saw in the previous chapter). Web APIs typically use a header instead of a cookie for authentication, but the process is the same.

Once a request is authenticated and you know who is making the request, you can determine whether they're allowed to execute an action on your server. This process is called authorization and is the focus of this chapter.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Before we dive into code and start looking at authorization in ASP.NET Core, I'll put these concepts into a real-life scenario you're hopefully familiar with: checking in at an airport. To enter an airport and board a plane, you must pass through several steps: an initial step to prove who you are (authentication); and subsequent steps that check whether you're allowed to proceed (authorization). In simplified form, these might look like:

1. Show passport at check-in desk. Receive a boarding pass.
2. Show boarding pass to enter security. Pass through security.
3. Show frequent flyer card to enter the airline lounge. Enter lounge.
4. Show boarding pass to board flight. Enter airplane.

Obviously, these steps, also shown in figure 15.1, will vary somewhat (I don't have a frequent flyer card!), but we'll go with them for now. Let's explore each step a little further.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

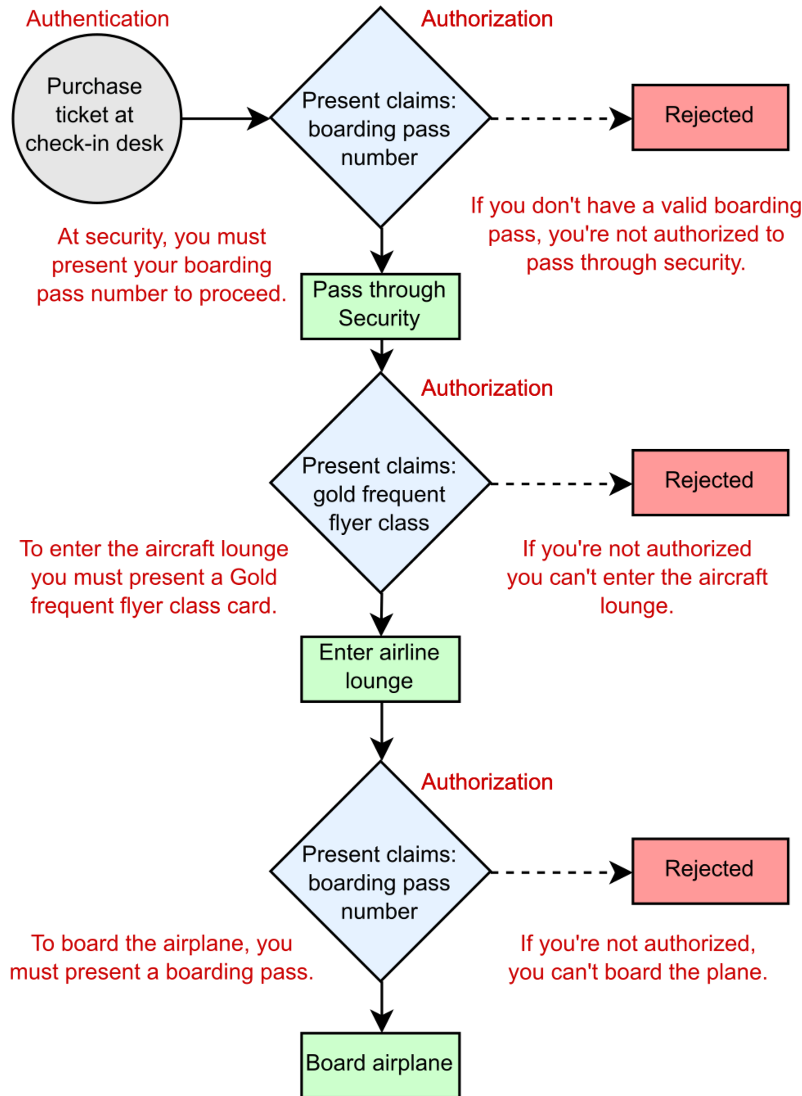


Figure 15.1 When boarding a plane at an airport, you pass through several authorization steps. At each authorization step, you must present a claim in the form of a boarding pass or a frequent flyer card. If you're not authorized, then access will be denied.

When you arrive at the airport, the first thing you do is go to the check-in counter. Here, you can purchase a plane ticket, but to do so, you need to prove who you are by providing a passport; you *authenticate* yourself. If you've forgotten your passport, you can't authenticate, and you can't go any further.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Once you've purchased your ticket, you're issued a boarding pass, which says which flight you're on. We'll assume it also includes a `BoardingPassNumber`. You can think of this number as an additional *claim* associated with your identity.

> **DEFINITION** A *claim* is a piece of information about a user that consists of a *type* and an optional *value*.

The next step is security. The security guards will ask you to present your boarding pass for inspection, which they'll use to check that you have a flight and so are allowed deeper into the airport. This is an authorization process: you must have the required claim (a `BoardingPassNumber`) to proceed.

If you don't have a valid `BoardingPassNumber`, there are two possibilities for what happens next:

- *If you haven't yet purchased a ticket*—You'll be directed back to the check-in desk, where you can authenticate and purchase a ticket. At that point, you can try to enter security again.
- *If you have an invalid ticket*—You won't be allowed through security and there's nothing else you can do. If, for example, you show up with a boarding pass a week late for your flight, they probably won't let you through. (Ask me how I know!)

Once you're through security, you need to wait for your flight to start boarding, but unfortunately there aren't any seats free. Typical! Luckily, you're a regular flier, and you've notched up enough miles to achieve a "Gold" frequent flyer status, so you can use the airline lounge.

You head to the lounge, where you're asked to present your Gold Frequent Flyer card to the attendant, and they let you in. This is another example of authorization. You must have a `FrequentFlyerClass` claim with a value of `Gold` to proceed.

> **NOTE** You've used authorization twice so far in this scenario. Each time, you presented a claim to proceed. In the first case, the presence of any `BoardingPassNumber` was sufficient, whereas for the `FrequentFlyerClass` claim, you needed the specific value of `Gold`.

When you're boarding the airplane, you have one final authorization step, in which you must present the `BoardingPassNumber` claim again. You presented this claim earlier, but boarding the aircraft is a distinct action from entering security, so you have to present it again.

This whole scenario has lots of parallels with requests to a web app:

- Both processes start with authentication.
- You have to prove *who* you are in order to retrieve the *claims* you need for authorization.
- You use authorization to protect sensitive actions like entering security and the airline lounge.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

I'll reuse this airport scenario throughout the chapter to build a simple web application that simulates the steps you take in an airport. We've covered the concept of authorization in general, so in the next section, we'll look at how authorization works in ASP.NET Core. You'll start with the most basic level of authorization, ensuring only authenticated users can execute an action, and look at what happens when you try to execute such an action.

15.2 Authorization in ASP.NET Core

In this section you'll see how the authorization principles described in the previous section apply to an ASP.NET Core application. You'll learn about the role of the `[Authorize]` attribute and `AuthorizationMiddleware` in authorizing requests to Razor Pages and MVC actions. Finally, you'll learn about the process of preventing unauthenticated users from executing endpoints, and what happens when users are unauthorized.

The ASP.NET Core framework has authorization built in, so you can use it anywhere in your app, but it's most common in .NET Core 3.1 to apply authorization using the `AuthorizationMiddleware`. The `AuthorizationMiddleware` should be placed *after* both the routing middleware and the authentication middleware, but *before* the endpoint middleware, as shown in figure 15.2.

> **REMINDER** In ASP.NET Core, an *endpoint* refers to the handler selected by the routing middleware, which will generate a response when executed. It is typically a Razor Page, or a Web API action method.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

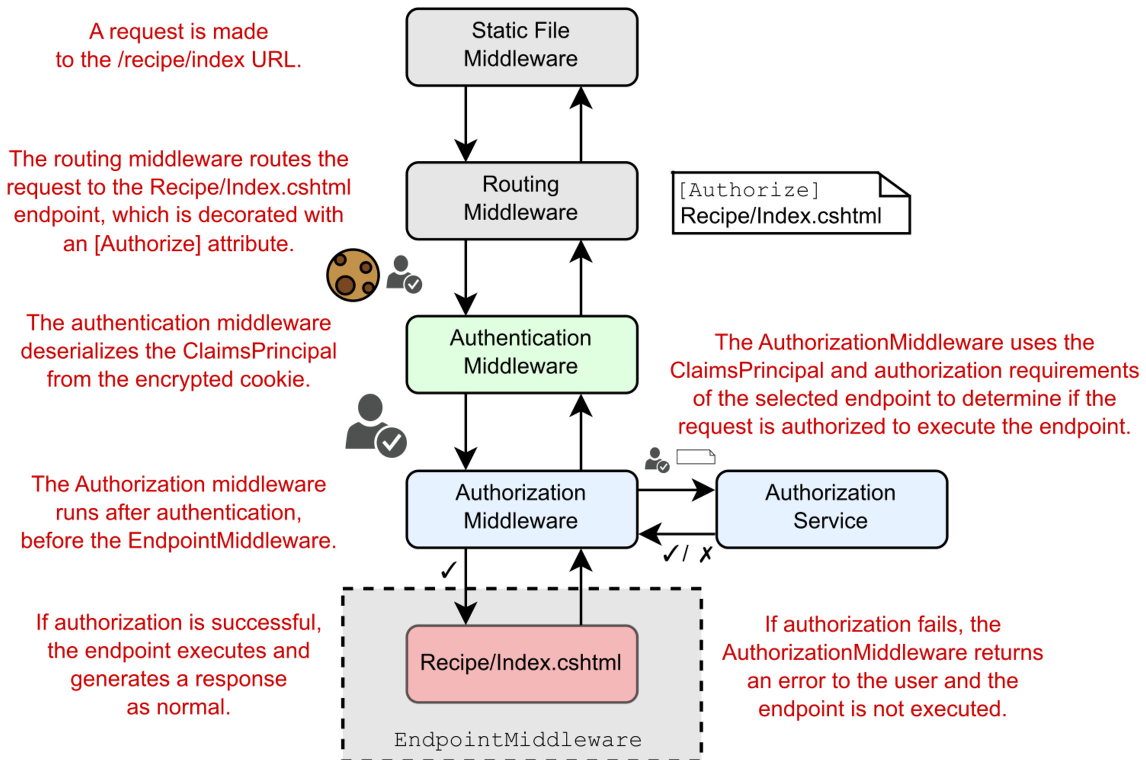


Figure 15.2 Authorization occurs after an endpoint has been selected, and after the request is authenticated, but before the action method or Razor Page endpoint is executed.

With this configuration, the `RoutingMiddleware` selects an endpoint to execute based on the request's URL, for example a Razor Page, as you saw in chapter 5. Metadata about the selected endpoint is available to all middleware that occurs after the routing middleware. This metadata includes details about any authorization requirements for the endpoint, and is typically attached by decorating an action or Razor Page with an `[Authorize]` attribute.

The `AuthenticationMiddleware` deserializes the encrypted cookie (or bearer token for APIs) associated with the request to create a `ClaimsPrincipal`. This object is set as the `HttpContext.User` for the request, so all subsequent middleware can access this value. It contains all the `Claims` that were added to the cookie when the user authenticated.

Now we come to the `AuthorizationMiddleware`. This middleware checks if the selected endpoint has any authorization requirements, based on the metadata provided by the `RoutingMiddleware`. If the endpoint has authorization requirements, the `AuthorizationMiddleware` uses the `HttpContext.User` to determine if the current request is authorized to execute the endpoint.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

If the request is authorized, the next middleware in the pipeline executes as normal. If the request is not authorized, the `AuthorizationMiddleware` short-circuits the middleware pipeline, and the endpoint middleware is never executed.

> **REMINDER** The order of middleware in your pipeline is very important. The call to `UseAuthorization()` must come after `UseRouting()` and `UseAuthentication()`, but before `UseEndpoints()`.

Changes to authorization in ASP.NET Core 3.0

The authorization system changed significantly in ASP.NET Core 3.0. Prior to this release, the `AuthorizationMiddleware` did not exist. Instead, the `[Authorize]` attribute executed the authorization logic as part of the MVC filter pipeline.

In practice, from the point of view of using authorization in your actions and Razor Pages, there is no real difference from a developer's point of view. Why change it then?

The new design, using the `AuthorizationMiddleware` in conjunction with endpoint routing (introduced at the same time), enables additional scenarios. The changes make it easier to apply authorization to non-MVC/Razor Page endpoints. You'll see how to create these types of endpoints in chapter 19. You can also read more about the authorization changes here: https://docs.microsoft.com/aspnet/core/migration/22-to-30#authorization.

The `AuthorizationMiddleware` is responsible for applying authorization requirements and ensuring that only authorized users can execute protected endpoints. In section 15.2.1 you'll learn how to apply the simplest authorization requirement, and in section 15.2.2 you'll see how the framework responds when a user is not authorized to execute an endpoint.

15.2.1 Preventing anonymous users from accessing your application

When you think about authorization, you typically think about checking whether a particular user has permission to execute an endpoint. In ASP.NET Core you normally achieve this by checking whether a user has a given claim.

There's an even more basic level of authorization we haven't considered yet—only allowing authenticated users to execute an endpoint. This is even simpler than the claims scenario (which we'll come to later) as there are only two possibilities:

- *The user is authenticated*—The action executes as normal.
- *The user is unauthenticated*—The user can't execute the endpoint.

You can achieve this basic level of authorization by using the `[Authorize]` attribute, which you saw in chapter 13, when we discussed authorization filters. You can apply this attribute to your actions, as shown in the following listing, to restrict them to authenticated (logged-in) users only. If an unauthenticated user tries to execute an action or Razor Page protected with the `[Authorize]` attribute in this way, they'll be redirected to the login page.

Listing 15.1 Applying `[Authorize]` to an action

```
public class RecipeApiController : ControllerBase
```

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
{
    public IActionResult List()                #A
    {
        return Ok();
    }

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

#A This action can be executed by anyone, even when not logged in.
#B Applies [Authorize] to individual actions, whole controllers, or Razor Pages.
#C 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 here: https://andrewlock.net/setting-global-authorization-policies-using-the-defaultpolicy-and-the-fallbackpolicy-in-aspnet-core-3/.

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]                                    #A
public class AccountController : ControllerBase
{
    public IActionResult ManageAccount()       #B
    {
        return Ok();
```

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
    }
    [AllowAnonymous]                          #C
    public IActionResult Login()              #D
    {
        return Ok();
    }
}
```

#A Applied at the controller scope, so user must be authenticated for all actions on the controller.
#B Only authenticated users may execute ManageAccount.
#C [AllowAnonymous] overrides [Authorize] to allow unauthenticated users.
#D Login can be executed by anonymous users.

> **WARNING If you apply the `[Authorize]` attribute globally, be sure to add the `[AllowAnonymous]` attribute to your login actions, error actions, password reset actions, and any other actions that you need unauthenticated users to execute. If you're using the default Identity UI described in chapter 14, then this is already configured for you.**

If an unauthenticated user attempts to execute an action protected by the `[Authorize]` attribute, traditional web apps will redirect them to the login page. But what about web APIs? And what about more complex scenarios, where a user is logged in but doesn't have the necessary claims to execute an action? In section 15.2.2, we'll look at how the ASP.NET Core authentication services handle all of this for you.

15.2.2 Handling unauthorized requests

In the previous section, you saw how to apply the `[Authorize]` attribute to an action to ensure only authenticated users can execute it. In section 15.3, we'll look at more complex examples that require you to also have a specific claim. In both cases, you must meet one or more authorization requirements (for example, you must be authenticated) to execute the action.

If the user meets the authorization requirements, then the request passes unimpeded through the `AuthorizationMiddleware`, and the endpoint is executed in the `EndpointMiddleware`. If they don't meet the requirements for the selected endpoint, the authorization middleware will short-circuit the request. Depending on why the request failed authorization, the authorization middleware generates one of two different types of response:

- Challenge—This response indicates the user was not authorized to execute the action, because they weren't yet logged in.
- Forbid—This response indicates that the user was logged in but didn't meet the requirements to execute the action. They didn't have a required claim, for example.

> **NOTE If you apply the `[Authorize]` attribute in basic form, as you did in section 15.2.1, then you will only generate challenge responses. In this case, a challenge response will be generated for unauthenticated users, but authenticated users will always be authorized.**

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

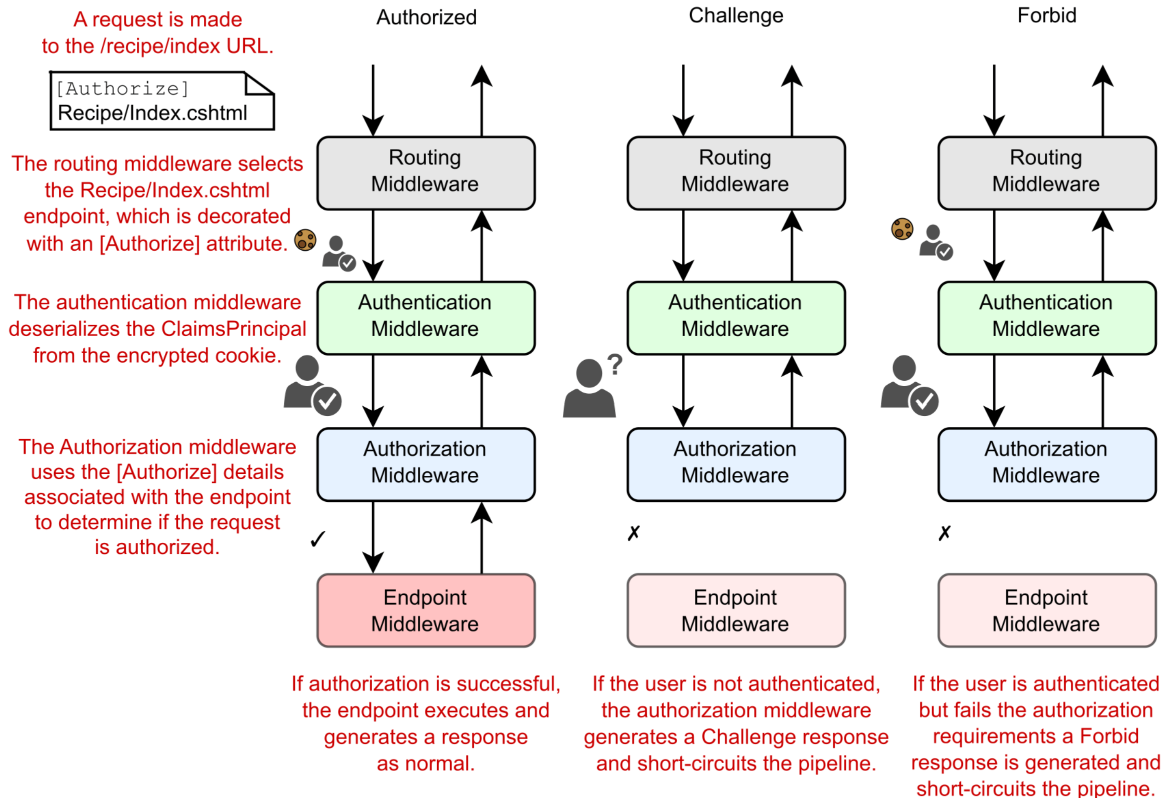


Figure 15.3. The three types of response to an authorization attempt. In the left example, the request contains an authentication cookie, so the user is authenticated in the `AuthenticationMiddleware`. The `AuthorizationMiddleware` confirms the authenticated user can access the selected endpoint, so the endpoint is executed. In the centre example, the request is not authenticated, so the `AuthorizationMiddleware` generates a Challenge response. In the right example, the request is authenticated, but the user does not have permission to execute the endpoint, so a Forbid response is generated.

The exact HTTP response generated by a challenge or forbid response typically depends on the type of application you're building and so the type of authentication your application uses: a traditional web application with Razor Pages, or an API application.

For traditional web apps using cookie authentication, such as when you use ASP.NET Core Identity, as in chapter 14, the challenge and forbid responses generate an HTTP redirect to a page in your application. A challenge response indicates the user isn't yet authenticated, so they're redirected to the login page for the app. After logging in, they can attempt to execute the protected resource again.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

A forbid response means the request was from a user that *already* logged in, but they're still not allowed to execute the action. Consequently, the user is redirected to a Forbidden or Access Denied web page, as shown in figure 15.4, which informs them they can't execute the action or Razor Page.

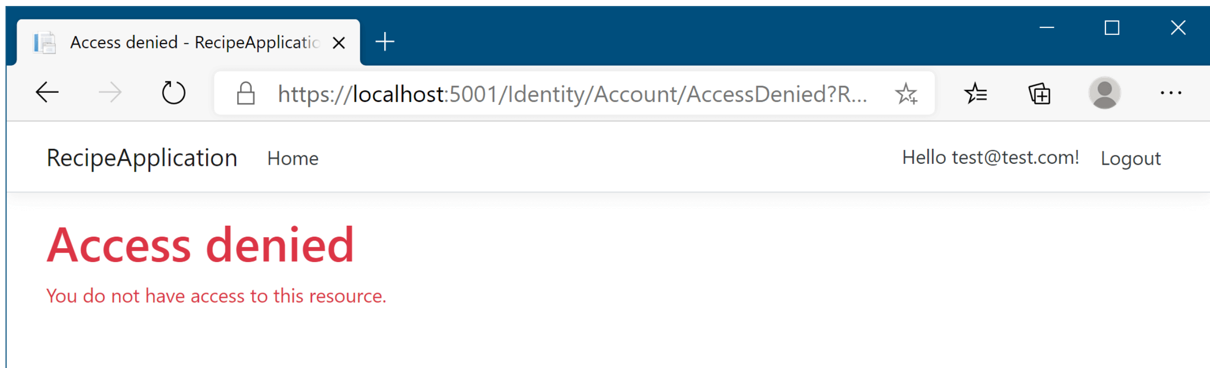


Figure 15.4 In traditional web apps using cookie authentication. If you don't have permission to execute a Razor Page and you're already logged in, you'll be redirected to an Access Denied page.

The preceding behavior is standard for traditional web apps, but Web APIs typically use a different approach to authentication, as you saw in chapter 14. Instead of logging in and using the API directly, you'd typically log in to a third-party application that provides a token to the client-side SPA or mobile app. The client-side app sends this token when it makes a request to your Web API.

Authenticating a request for a Web API using tokens is essentially identical to a traditional web app that uses cookies; `AuthenticationMiddleware` deserializes the cookie or token to create the `ClaimsPrincipal`. The difference is in how a Web API handles authorization failures.

When a web API app generates a challenge response, it returns a `401 Unauthorized` error response to the caller. Similarly, when the app generates a forbid response, it returns a `403 Forbidden` response. The traditional web app essentially handled these errors by automatically redirecting unauthorized users to the login or "access denied" page, but the web API doesn't do this. It's up to the client-side SPA or mobile app to detect these errors and handle them as appropriate.

> **TIP** **The difference in authorization behavior is one of the reasons I generally recommend creating separate apps for your APIs and Razor pages apps—it's possible to have both in the same app, but the configuration is more complex.**

The different behavior between traditional web apps and SPAs can be confusing initially, but you generally don't need to worry about that too much in practice. Whether you're building a

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Web API or a traditional MVC web app, the authorization code in your app looks the same in both cases. Apply `[Authorize]` attributes to your endpoints, and let the framework take care of the differences for you.

> **NOTE** In chapter 14, you saw how to configure ASP.NET Core Identity in a Razor Pages app. This chapter assumes you're building a Razor Pages app too, but the chapter is equally applicable if you're building a web API. Authorization policies are applied in the same way, whichever style of app you're building. It's only the final response of unauthorized requests that differ.

You've seen how to apply the most basic authorization requirement—restricting an endpoint to authenticated users only—but most apps need something more subtle than this all-or-nothing approach.

Consider the airport scenario from section 15.1. Being authenticated (having a passport) isn't enough to get you through security. Instead, you also need a specific claim: `BoardingPassNumber`. In the next section, we'll look at how you can implement a similar requirement in ASP.NET Core.

15.3 Using policies for claims-based authorization

In the previous section, you saw how to require that users are logged in to access an endpoint. In this section, you'll see how to apply additional requirements. You'll learn to use authorization policies to perform claims-based authorization, to require that a logged in user has the required claims to execute a given endpoint.

In chapter 14, you saw that authentication in ASP.NET Core centers around a `ClaimsPrincipal` object, which represents the user. This object has a collection of claims that contain pieces of information about the user, such as their name, email, and date of birth.

You can use these to customize the app for each user, by displaying a welcome message addressing the user by name for example, but you can also use claims for authorization. For example, you might only authorize a user if they have a specific claim (such as `BoardingPassNumber`) or if a claim has a specific value (`FrequentFlyerClass` claim with the value `Gold`).

In ASP.NET Core, the rules that define whether a user is authorized are encapsulated in a *policy*.

> **DEFINITION** A *policy* defines the requirements you must meet for a request to be authorized.

Policies can be applied to an action using the `[Authorize]` attribute, similar to the way you saw in section 15.2.1. This listing shows a Razor Page `PageModel` that represents the first authorization step in the airport scenario. The AirportSecurity.cshtml Razor Page is protected by an `[Authorize]` attribute, but you've also provided a policy name: `"CanEnterSecurity"`.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Listing 15.3 Applying an authorization policy to a Razor Page

```
[Authorize("CanEnterSecurity")]                    #A
public class AirportSecurityModel : PageModel
{
    public void OnGet()                           #B
    {

    }
}
```

#A Applying the "CanEnterSecurity" policy using [Authorize]
#B Only users that satisfy the "CanEnterSecurity" policy can execute the Razor Page.

If a user attempts to execute the AirportSecurity.cshtml Razor Page, the authorization middleware will verify whether the user satisfies the policy's requirements (we'll look at the policy itself shortly). This gives one of three possible outcomes:

- *The user satisfies the policy.*—The middleware pipeline continues, and the `EndpointMiddleware` executes the Razor Page as normal.
- *The user is unauthenticated.*—The user is redirected to the login page.
- *The user is authenticated but doesn't satisfy the policy.*—The user is redirected to a "Forbidden" or "Access Denied" page.

These three outcomes correlate with the real-life outcomes you might expect when trying to pass through security at the airport:

- *You have a valid boarding pass.*—You can enter security as normal.
- *You don't have a boarding pass.*—You're redirected to purchase a ticket.
- *Your boarding pass is invalid (you turned up a day late, for example).*—You're blocked from entering.

Listing 15.3 shows how you can apply a policy to a Razor Page using the `[Authorize]` attribute, but you still need to define the `CanEnterSecurity` policy.

You add policies to an ASP.NET Core application in the `ConfigureServices` method of Startup.cs, as shown in listing 15.4. First, you add the authorization services using `AddAuthorization()`, and then you can add policies by calling `AddPolicy()` on the `AuthorizationOptions` object. You define the policy itself by calling methods on a provided `AuthorizationPolicyBuilder` (called `policyBuilder` here).

Listing 15.4 Adding an authorization policy using `AuthorizationPolicyBuilder`

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthorization(options =>               #A
    {
        options.AddPolicy(                             #B
            "CanEnterSecurity",                        #C
            policyBuilder => policyBuilder             #D
                .RequireClaim("BoardingPassNumber"));   #D
    });
```

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
    // Additional service configuration
}
```

#A Calls AddAuthorization to configure AuthorizationOptions.
#B Adds a new policy.
#C Provides a name for the policy.
#D Defines the policy requirements using AuthorizationPolicyBuilder.

When you call `AddPolicy` you provide a name for the policy, which should match the value you use in your `[Authorize]` attributes, and you define the requirements of the policy. In this example, you have a single simple requirement: the user must have a claim of type `BoardingPassNumber`. If a user has this claim, whatever its value, then the policy will be satisfied, and the user will be authorized.

> **REMEMBER** A *claim* is information about the user, as a key-value pair. A policy defines the requirements for successful authorization. A policy can require that a user has a given claim, as well as more complex requirements, as you'll see shortly.

`AuthorizationPolicyBuilder` contains several methods for creating simple policies like this, as shown in table 15.1. For example, an overload of the `RequireClaim()` method lets you specify a specific value that a claim must have. The following would let you create a policy where the `"BoardingPassNumber"` claim must have a value of `"A1234"`:

```
policyBuilder => policyBuilder.RequireClaim("BoardingPassNumber", "A1234");
```

Table 15.1 Simple policy builder methods on `AuthorizationPolicyBuilder`

Method	Policy behavior
`RequireAuthenticatedUser()`	The required user must be authenticated. Creates a policy similar to the default `[Authorize]` attribute, where you don't set a policy.
`RequireClaim(claim, values)`	The user must have the specified claim. If provided, the claim must be one of the specified values.
`RequireUsername(username)`	The user must have the specified username.
`RequireAssertion(function)`	Executes the provided lambda function, which returns a `bool`, indicating whether the policy was satisfied.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Role-based authorization vs. claims-based authorization

If you look at all of the methods available on the `AuthorizationPolicyBuilder` type using IntelliSense, you might notice that there's a method I didn't mention in table 15.1, `RequireRole()`. This is a remnant of the role-based approach to authorization used in previous versions of ASP.NET, and I don't recommend using it.

Before Microsoft adopted the claims-based authorization used by ASP.NET Core and recent versions of ASP.NET, role-based authorization was the norm. Users were assigned to one or more roles, such as `Administrator` or `Manager`, and authorization involved checking whether the current user was in the required role.

This role-based approach to authorization is possible in ASP.NET Core, but it's primarily for legacy compatibility reasons. Claims-based authorization is the suggested approach. Unless you're porting a legacy app that uses roles, I suggest you embrace claims-based authorization and leave those roles behind!

You can use these methods to build simple policies that can handle basic situations, but often you'll need something more complicated. What if you wanted to create a policy that enforces only users over the age of 18 can execute an endpoint?

The `DateOfBirth` claim provides the information you need, but there's not a *single* correct value, so you couldn't use the `RequireClaim()` method. You *could* use the `RequireAssertion()` method and provide a function that calculates the age from the `DateOfBirth` claim, but that could get messy pretty quickly.

For more complex policies that can't be easily defined using the `RequireClaim()` method, I recommend you take a different approach and create a custom policy, as you'll see in the following section.

15.4 Creating custom policies for authorization

You've already seen how to create a policy by requiring a specific claim, or requiring a specific claim with a specific value, but often the requirements will be more complex than that. In this section you'll learn how to create custom authorization requirements and handlers. You'll also see how to configure authorization requirements where there are multiple ways to satisfy a policy, any of which are valid.

Let's return to the airport example. You've already configured the policy for passing through security, and now you're going to configure the policy that controls whether you're authorized to enter the airline lounge.

As you saw in figure 15.1, you're allowed to enter the lounge if you have a `FrequentFlyerClass` claim with a value of `Gold`. If this was the only requirement, you could use `AuthorizationPolicyBuilder` to create a policy using:

```
options.AddPolicy("CanAccessLounge", policyBuilder =>
    policyBuilder.RequireClaim("FrequentFlyerClass", "Gold");
```

But what if the requirements are more complicated than this? For example, you can enter the lounge if

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

- You're a gold-class frequent flyer (have a `FrequentFlyerClass` claim with value `"Gold"`)
- Or you're an employee of the airline (have an `EmployeeNumber` claim)
- And you're at least 18 years old (as calculated from the `DateOfBirth` claim)

If you've ever been banned from the lounge (you have an `IsBannedFromLounge` claim), then you won't be allowed in, even if you satisfy the other requirements.

There's no way of achieving this complex set of requirements with the basic usage of `AuthorizationPolicyBuilder` you've seen so far. Luckily, these methods are a wrapper around a set of building blocks that you can combine to achieve the desired policy.

15.4.1 Requirements and handlers: the building blocks of a policy

Every policy in ASP.NET Core consists of one or more *requirements*, and every requirement can have one or more *handlers*. For the airport lounge example, you have a single policy (`"CanAccessLounge"`), two requirements (`MinimumAgeRequirement` and `AllowedInLoungeRequirement`), and several handlers, as shown in figure 15.5.

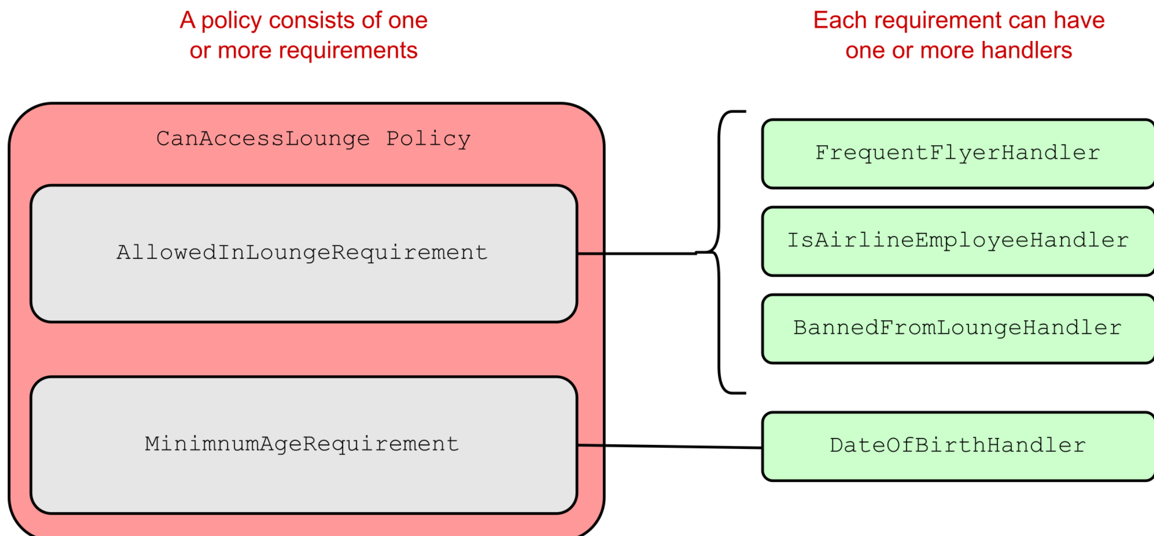


Figure 15.5 A policy can have many requirements, and every requirement can have many handlers. By combining multiple requirements in a policy, and by providing multiple handler implementations, you can create complex authorization policies that meet any of your business requirements.

For a policy to be satisfied, a user must fulfill *all* the requirements. If the user fails *any* of the requirements, the authorize middleware won't allow the protected endpoint to be executed. In this example, a user must be allowed to access the lounge *and* must be over 18 years old.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Each requirement can have one or more handlers, which will confirm that the requirement has been satisfied. For example, as shown in figure 15.5, `AllowedInLoungeRequirement` has two handlers that can satisfy the requirement:

- `FrequentFlyerHandler`
- `IsAirlineEmployeeHandler`

If the user satisfies either of these handlers, then `AllowedInLoungeRequirement` is satisfied. You don't need all handlers for a requirement to be satisfied, you just need one.

> **NOTE** **Figure 15.5 shows a third handler, `BannedFromLoungeHandler`, which I'll cover in section 15.4.2. It's slightly different, in that it can only *fail* a requirement, not *satisfy* it.**

You can use requirements and handlers to achieve most any combination of behavior you need for a policy. By combining handlers for a requirement, you can validate conditions using a logical `OR`: if any of the handlers are satisfied, the requirement is satisfied. By combining requirements, you create a logical `AND`: all the requirements must be satisfied for the policy to be satisfied, as shown in figure 15.6.

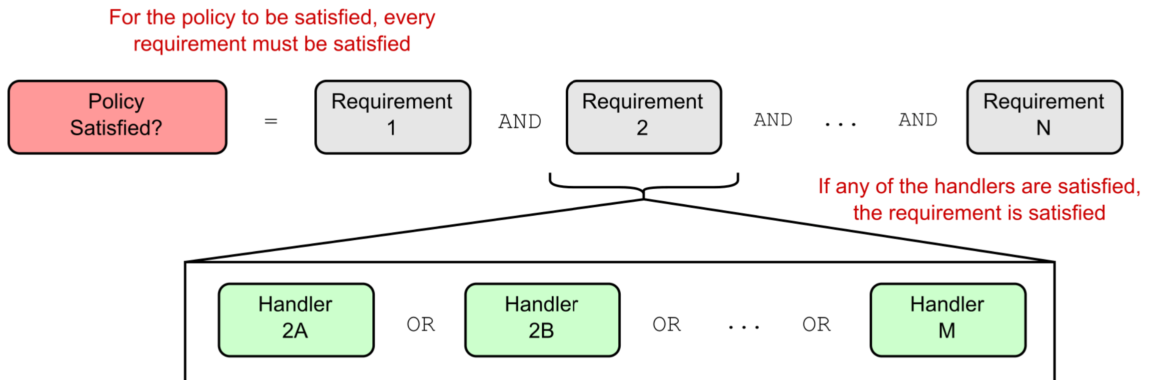


Figure 15.6 For a policy to be satisfied, every requirement must be satisfied. A requirement is satisfied if any of the handlers are satisfied.

> **TIP** **You can also add multiple policies to a Razor Page or action method by applying the `[Authorize]` attribute multiple times, for example `[Authorize("Policy1"), Authorize("Policy2")]`. All policies must be satisfied for the request to be authorized.**

I've highlighted requirements and handlers that will make up your `"CanAccessLounge"` policy, so in the next section, you'll build each of the components and apply them to the airport sample app.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

15.4.2 Creating a policy with a custom requirement and handler

You've seen all the pieces that make up a custom authorization policy, so in this section, we'll explore the implementation of the "`CanAccessLounge`" policy.

CREATING AN IAUTHORIZATIONREQUIREMENT TO REPRESENT A REQUIREMENT

As you've seen, a custom policy can have multiple requirements, but what *is* a requirement in code terms? Authorization requirements in ASP.NET Core are any class that implements the `IAuthorizationRequirement` interface. This is a blank, marker interface, which you can apply to any class to indicate that it represents a requirement.

If the interface doesn't have any members, you might be wondering what the requirement class needs to look like. Typically, they're simple, POCO classes. The following listing shows `AllowedInLoungeRequirement`, which is about as simple as a requirement can get! It has no properties or methods; it implements the required `IAuthorizationRequirement` interface.

Listing 15.5 `AllowedInLoungeRequirement`

```
public class AllowedInLoungeRequirement
    : IAuthorizationRequirement { }                 #A
```

#A The interface identifies the class as an authorization requirement.

This is the simplest form of requirement, but it's also common for them to have one or two properties that make the requirement more generalized. For example, instead of creating the highly specific `MustBe18YearsOldRequirement`, you could instead create a parametrized `MinimumAgeRequirement`, as shown in the following listing. By providing the minimum age as a parameter to the requirement, you can reuse the requirement for other policies with different minimum age requirements.

Listing 15.6 The parameterized `MinimumAgeRequirement`

```
public class MinimumAgeRequirement : IAuthorizationRequirement    #A
{
    public MinimumAgeRequirement(int minimumAge)                  #B
    {
        MinimumAge = minimumAge;
    }
    public int MinimumAge { get; }                                #C
}
```

#A The interface identifies the class as an authorization requirement.
#B The minimum age is provided when the requirement is created.
#C Handlers can use the exposed minimum age to determine whether the requirement is satisfied.

The requirements are the easy part. They represent each of the components of the policy that must be satisfied for the policy to be satisfied overall.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

CREATING A POLICY WITH MULTIPLE REQUIREMENTS

You've created the two requirements, so now you can configure the "CanAccessLounge" policy to use them. You configure your policies as you did before, in the ConfigureServices method of Startup.cs. Listing 15.7 shows how to do this by creating an instance of each requirement and passing them to AuthorizationPolicyBuilder. The authorization handlers will use these requirement objects when attempting to authorize the policy.

Listing 15.7 Creating an authorization policy with multiple requirements

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthorization(options =>
    {                                                           #A
        options.AddPolicy(                                      #A
            "CanEnterSecurity",                                 #A
            policyBuilder => policyBuilder                      #A
                .RequireClaim(Claims.BoardingPassNumber));      #A
        options.AddPolicy(                                      #B
            "CanAccessLounge",                                  #B
            policyBuilder => policyBuilder.AddRequirements(     #C
                new MinimumAgeRequirement(18),                  #C
                new AllowedInLoungeRequirement()                #C
            ));
    });
    // Additional service configuration
}
```

#A Adds the previous simple policy for passing through security
#B Adds a new policy for the airport lounge, called CanAccessLounge
#C Adds an instance of each IAuthorizationRequirement object

You now have a policy called "CanAccessLounge" with two requirements, so you can apply it to a Razor Page or action method using the [Authorize] attribute, in exactly the same way you did for the "CanEnterSecurity" policy:

```
[Authorize("CanAccessLounge")]
public class AirportLoungeModel : PageModel
{
    public void OnGet() { }
}
```

When a request is routed to the AirportLounge.cshtml Razor Page, the authorize middleware executes the authorization policy and each of the requirements are inspected. But you saw earlier that the requirements are purely data; they indicate what needs to be fulfilled, they don't describe how that has to happen. For that, you need to write some handlers.

CREATING AUTHORIZATION HANDLERS TO SATISFY YOUR REQUIREMENTS

Authorization handlers contain the logic of how a specific IAuthorizationRequirement can be satisfied. When executed, a handler can do one of three things:

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

- Mark the requirement handling as a success
- Not do anything
- Explicitly fail the requirement

Handlers should implement `AuthorizationHandler<T>`, where `T` is the type of requirement they handle. For example, the following listing shows a handler for `AllowedInLoungeRequirement` that checks whether the user has a claim called `FrequentFlyerClass` with a value of `Gold`.

Listing 15.8 `FrequentFlyerHandler` **for** `AllowedInLoungeRequirement`

```
public class FrequentFlyerHandler :
    AuthorizationHandler<AllowedInLoungeRequirement>           #A
{
    protected override Task HandleRequirementAsync(            #B
        AuthorizationHandlerContext context,                           #C
        AllowedInLoungeRequirement requirement)                        #D
    {
        if(context.User.HasClaim("FrequentFlyerClass", "Gold"))        #E
        {
            context.Succeed(requirement);                              #F
        }
        return Task.CompletedTask;                                     #G
    }
}
```

#A The handler implements AuthorizationHandler<T>.
#B You must override the abstract HandleRequirementAsync method.
#C The context contains details such as the ClaimsPrincipal user object.
#D The requirement instance to handle
#E Checks whether the user has the FrequentFlyerClass claim with the Gold value
#F If the user had the necessary claim, then mark the requirement as satisfied by calling Succeed.
#G If the requirement wasn't satisfied, do nothing.

This handler is functionally equivalent to the simple `RequireClaim()` handler you saw at the start of section 15.4, but using the requirement and handler approach instead.

When a request is routed to the AirportLounge.cshtml Razor Page, the authorization middleware sees the `[Authorize]` attribute on the endpoint with the `"CanAccessLounge"` policy. It loops through all the requirements in the policy, and all the handlers for each requirement, calling the `HandleRequirementAsync` method for each.

The authorization middleware passes the current `AuthorizationHandlerContext` and the requirement to be checked to each handler. The current `ClaimsPrincipal` being authorized is exposed on the context as the `User` property. In listing 15.8, `FrequentFlyerHandler` uses the context to check for a claim called `FrequentFlyerClass` with the `Gold` value, and if it exists, concludes that the user is allowed to enter the airline lounge, by calling `Succeed()`.

> **NOTE** **Handlers mark a requirement as being successfully satisfied by calling** `context.Succeed()` **and passing the requirement as an argument.**

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

It’s important to note the behavior when the user *doesn’t* have the claim. `FrequentFlyerHandler` doesn’t do anything if this is the case (it returns a completed `Task` to satisfy the method signature).

> **NOTE** Remember, if any of the handlers associated with a requirement pass, then the requirement is a success. Only *one* of the handlers must succeed for the requirement to be satisfied.

This behavior, whereby you either call `context.Succeed()` or do nothing, is typical for authorization handlers. The following listing shows the implementation of `IsAirlineEmployeeHandler`, which uses a similar claim check to determine whether the requirement is satisfied.

Listing 15.9 `IsAirlineEmployeeHandler` handler

```
public class IsAirlineEmployeeHandler :
    AuthorizationHandler<AllowedInLoungeRequirement>                 #A
{
    protected override Task HandleRequirementAsync(                  #B
        AuthorizationHandlerContext context,                         #B
        AllowedInLoungeRequirement requirement)                      #B
    {
        if(context.User.HasClaim(c => c.Type == "EmployeeNumber"))   #C
        {
            context.Succeed(requirement);                            #D
        }
        return Task.CompletedTask;                                   #E
    }
}
```

#A The handler implements AuthorizationHandler<T>.
#B You must override the abstract HandleRequirementAsync method.
#C Checks whether the user has the EmployeeNumber claim
#D If the user has the necessary claim, then mark the requirement as satisfied by calling Succeed.
#E If the requirement wasn’t satisfied, do nothing.

> **TIP** It’s possible to write very generic handlers that can be used with multiple requirements, but I suggest sticking to handling a single requirement only. If you need to extract some common functionality, move it to an external service and call that from both handlers.

This pattern of authorization handler is common,[66] but in some cases, instead of checking for a *success* condition, you might want to check for a *failure* condition. In the airport example, you don’t want to authorize someone who was previously banned from the lounge, even if they would otherwise be allowed to enter.

---

[66] I’ll leave the implementation of `MinimumAgeHandler` for `MinimumAgeRequirement` as an exercise for the reader. You can find an example in the code samples for the chapter.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

You can handle this scenario by using the `context.Fail()` method exposed on the context, as shown in the following listing. Calling `Fail()` in a handler will always cause the requirement, and hence the whole policy, to fail. You should only use it when you want to guarantee failure, even if other handlers indicate success.

Listing 15.10 Calling `context.Fail()` in a handler to fail the requirement

```
public class BannedFromLoungeHandler :
    AuthorizationHandler<AllowedInLoungeRequirement>            #A
{
    protected override Task HandleRequirementAsync(             #B
        AuthorizationHandlerContext context,                    #B
        AllowedInLoungeRequirement requirement)                 #B
    {
        if(context.User.HasClaim(c => c.Type == "IsBanned"))    #C
        {
            context.Fail();                                     #D
        }

        return Task.CompletedTask;                              #E
    }
}
```

#A The handler implements AuthorizationHandler<T>.
#B You must override the abstract HandleRequirementAsync method.
#C Checks whether the user has the IsBanned claim
#D If the user has the claim, then fail the requirement by calling Fail. The whole policy will fail.
#E If the claim wasn't found, do nothing.

In most cases, your handlers will either call `Succeed()` or will do nothing, but the `Fail()` method is useful when you need this kill-switch to guarantee that a requirement won't be satisfied.

> **NOTE** Whether a handler calls `Succeed()`, `Fail()`, or neither, the authorization system will always execute all of the handlers for a requirement, and all the requirements for a policy so you can be sure your handlers will always be called.

The final step to complete your authorization implementation for the app is to register the authorization handlers with the DI container, as shown in listing 15.11.

Listing 15.11 Registering the authorization handlers with the DI container

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthorization(options =>
    {
        options.AddPolicy(
            "CanEnterSecurity",
            policyBuilder => policyBuilder
                .RequireClaim(Claims.BoardingPassNumber));
        options.AddPolicy(
            "CanAccessLounge",
```

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
            policyBuilder => policyBuilder.AddRequirements(
                new MinimumAgeRequirement(18),
                new AllowedInLoungeRequirement()
            ));
    });
    services.AddSingleton<IAuthorizationHandler, MinimumAgeHandler>();
    services.AddSingleton<IAuthorizationHandler, FrequentFlyerHandler>();
    services
        .AddSingleton<IAuthorizationHandler, BannedFromLoungeHandler>();
    Services
        .AddSingleton<IAuthorizationHandler, IsAirlineEmployeeHandler>();
    // Additional service configuration
}
```

For this app, the handlers don't have any constructor dependencies, so I've registered them as singletons with the container. If your handlers have scoped or transient dependencies (the EF Core `DbContext`, for example), then you might want to register them as scoped instead, as appropriate.

> **REMINDER** Services are registered with a lifetime of either transient, scoped, or singleton, as discussed in chapter 10.

You can combine the concepts of policies, requirements, and handlers in many ways to achieve your goals for authorization in your application. The example in this section, although contrived, demonstrates each of the components you need to apply authorization declaratively at the action method or Razor Page level, by creating policies and applying the `[Authorize]` attribute as appropriate.

As well as applying the `[Authorize]` attribute explicitly to actions and Razor Pages, you can also configure it globally, so that a policy is applied to every Razor Page or controller in your application. Additionally, for Razor Pages, you can apply different authorization policies to different folders. You can read more about applying authorization policies using conventions here https://docs.microsoft.com/aspnet/core/security/authorization/razor-pages-authorization.

There's one area, however, where the `[Authorize]` attribute falls short: resource-based authorization. The `[Authorize]` attribute attaches metadata to an endpoint, so the authorization middleware can authorize the user *before* an endpoint is executed, but what if you need to authorize the action *during* the action method or Razor Page handler?

This is common when you're applying authorization at the document or resource level. If users are only allowed to edit documents they created, then you need to load the document before you can tell whether you're allowed to edit it! This isn't easy with the declarative `[Authorize]` attribute approach, so you must use an alternative, imperative approach. In the next section, you'll see how to apply this resource-based authorization in a Razor Page handler.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

15.5 Controlling access with resource-based authorization

In this section you'll learn about resource-based authorization. This is used when you need to know details about the resource being protected to determine if a user is authorized. You'll learn how to apply authorization policies manually using the `IAuthorizationService`, and how to create resource-based `AuthorizationHandler`s.

Resource-based authorization is a common problem for applications, especially when you have users that can create or edit some sort of document. Consider the recipe application you built in the previous 3 chapters. This app lets users create, view, and edit recipes.

Up to this point, everyone can create new recipes, and anyone can edit any recipe, even if they haven't logged in. Now you want to add some additional behavior:

- Only authenticated users should be able to create new recipes.
- You can only edit the recipes you created.

You've already seen how to achieve the first of these requirements; decorate the Create.cshtml Razor Page with an `[Authorize]` attribute and don't specify a policy, as shown in this listing. This will force the user to authenticate before they can create a new recipe.

Listing 15.12 Adding `AuthorizeAttribute` to the Create.cshtml Razor Page

```
[Authorize]                                          #A
public class CreateModel : PageModel
{
    [BindProperty]
    public CreateRecipeCommand Input { get; set; }

    public void OnGet()                             #B
    {                                               #B
        Input = new CreateRecipeCommand();          #B
    }                                               #B

    public async Task<IActionResult> OnPost()       #C
    {                                               #C
        // Method body not shown for brevity        #C
    }                                               #C
}
```

#A Users must be authenticated to execute the Create.cshtml Razor Page.
#B All page handlers are protected. You can only apply [Authorize] to the PageModel, not handlers
#C ...

> **TIP** **As with all filters, you can only apply the `[Authorize]` attribute to the Razor Page, not to individual page handlers. The attribute applies to all page handlers in the Razor Page.**

Adding the `[Authorize]` attribute fulfills your first requirement, but unfortunately, with the techniques you've seen so far, you have no way to fulfill the second. You could apply a policy that either permits or denies a user the ability to edit *all* recipes, but there's currently no easy way to restrict this so that a user can only edit *their own* recipes.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

In order to find out who created the `Recipe`, you must first load it from the database. Only then can you attempt to authorize the user, taking the specific recipe (resource) into account. The following listing shows a partially implemented page handler for how this might look, where authorization occurs part way through the method, after the `Recipe` object has been loaded.

Listing 15.13 The Edit.cshtml page must load the `Recipe` before authorizing the request

```
public IActionResult OnGet(int id)                #A
{
    var recipe = _service.GetRecipe(id);         #B
    var createdById = recipe.CreatedById;        #B
    // Authorize user based on createdById       #C
    if(isAuthorized)                             #D
    {                                            #D
        return View(recipe);                     #D
    }                                            #D
}
```

#A The id of the recipe to edit is provided by model binding.
#B You must load the Recipe from the database before you know who created it.
#C You must authorize the current user, to verify they're allowed to edit this specific Recipe.
#D The action method can only continue if the user was authorized.

You need access to the resource (in this case, the `Recipe` entity) to perform the authorization, so the declarative `[Authorize]` attribute can't help you. In section 15.5.1, you'll see the approach you need to take to handle these situations and to apply authorization inside the action method or Razor Page.

> **WARNING** Be careful when exposing the integer ID of your entities in the URL, as in listing 15.13. Users will be able to edit every entity by modifying the ID in the URL to access a different entity. Be sure to apply authorization checks, otherwise you could expose a security vulnerability called Insecure Direct Object Reference.[67]

15.5.1 Manually authorizing requests with IAuthorizationService

All of the approaches to authorization so far have been *declarative*. You apply the `[Authorize]` attribute, with or without a policy name, and you let the framework take care of performing the authorization itself.

For this recipe-editing example, you need to use *imperative* authorization, so you can authorize the user after you've loaded the `Recipe` from the database. Instead of applying a

---

[67] You can read about this vulnerability and ways to counteract it on the Open Web Application Security Project (OWASP): www.owasp.org/index.php/Top_10_2007-Insecure_Direct_Object_Reference.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

marker saying, "Authorize this method," you need to write some of the authorization code yourself.

> **DEFINITION** ***Declarative*** **and** ***imperative*** **are two different styles of programming. Declarative programming describes** ***what*** **you're trying to achieve, and lets the framework figure out how to achieve it. Imperative programming describes** ***how*** **to achieve something by providing each of the steps needed.**

ASP.NET Core exposes `IAuthorizationService`, which you can inject into your Razor Pages and controllers for imperative authorization. This listing shows how you can update the Edit.cshtml Razor Page (shown partially in listing 15.13) to use the `IAuthorizationService` and verify whether the action is allowed to continue execution.

Listing 15.14 Using `IAuthorizationService` for resource-based authorization

```
[Authorize]                                                     #A
public class EditModel : PageModel
{
    [BindProperty]
    public Recipe Recipe { get; set; }

    private readonly RecipeService _service;
    private readonly IAuthorizationService _authService;     #B

    public EditModel(
        RecipeService service,
        IAuthorizationService authService)                   #B
    {
        _service = service;
        _authService = authService;                          #B
    }

    public async Task<IActionResult> OnGet(int id)
    {
        Recipe = _service.GetRecipe(id);                         #C
        var authResult = await _authService                      #D
            .AuthorizeAsync(User, Recipe, "CanManageRecipe");    #D
        if (!authResult.Succeeded)                               #E
        {                                                        #E
            return new ForbidResult();                           #E
        }                                                        #E

        return Page();                                           #F
    }
}
```

#A Only authenticated users should be allowed to edit recipes.
#B IAuthorizationService is injected into the class constructor using DI.
#C Load the recipe from the database.
#D Calls IAuthorizationService, providing ClaimsPrinicipal, resource, and the policy name
#E If authorization failed, returns a Forbidden result
#F If authorization was successful, continues displaying the Razor Page

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

`IAuthorizationService` exposes an `AuthorizeAsync` method, which requires three things to authorize the request:

- The `ClaimsPrincipal` user object, exposed on the `PageModel` as `User`.
- The resource being authorized: `Recipe`.
- The policy to evaluate: `"CanManageRecipe"`.

The authorization attempt returns an `AuthorizationResult` object, which indicates whether the attempt was successful via the `Succeeded` property. If the attempt wasn't successful, then you should return a new `ForbidResult`, which will be converted either into an HTTP `403 Forbidden` response or will redirect the user to the "access denied" page, depending on if you're building a traditional web app with Razor Pages or a Web API.

> **NOTE** As mentioned in section 15.2.2, which type of response is generated depends on which authentication services are configured. The default Identity configuration, used by Razor Pages, generates redirects. The JWT bearer token authentication typically used with Web APIs generates HTTP `401` and `403` responses instead.

You've configured the imperative authorization in the Edit.cshtml Razor Page itself, but you still need to define the `"CanManageRecipe"` policy that you use to authorize the user. This is the same process as for declarative authorization, so you have to:

- Create a *policy* in `ConfigureServices` by calling `AddAuthorization()`
- Define one or more *requirements* for the policy
- Define one or more *handlers* for each requirement
- Register the handlers in the DI container

With the exception of the handler, these steps are all identical to the declarative authorization approach with the `[Authorize]` attribute, so I'll only run through them briefly here.

First, you can create a simple `IAuthorizationRequirement`. As with many requirements, this contains no data and simply implements the marker interface.

```
public class IsRecipeOwnerRequirement : IAuthorizationRequirement { }
```

Defining the policy in `ConfigureServices` is similarly simple, as you have only this single requirement. Note that there's nothing resource-specific in any of this code so far:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthorization(options => {
        options.AddPolicy("CanManageRecipe", policyBuilder =>
            policyBuilder.AddRequirements(new IsRecipeOwnerRequirement()));
    });
}
```

You're halfway there; all you need to do now is create an authorization handler for `IsRecipeOwnerRequirement` and register it with the DI container.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

15.5.2 Creating a resource-based AuthorizationHandler

Resource-based authorization handlers are essentially the same as the authorization handler implementations you saw in section 15.4.2. The only difference is that the handler also has access to the resource being authorized.

To create a resource-based handler, you should derive from the `AuthorizationHandler<TRequirement, TResource>` base class, where `TRequirement` is the type of requirement to handle, and `TResource` is the type of resource that you provide when calling `IAuthorizationService`. Compare this to the `AuthorizationHandler<T>` class you implemented previously, where you only specified the requirement.

This listing shows the handler implementation for your recipe application. You can see that you've specified the requirement as `IsRecipeOwnerRequirement`, the resource as `Recipe`, and have implemented the `HandleRequirementAsync` method.

Listing 15.15 `IsRecipeOwnerHandler` for resource-based authorization

```
public class IsRecipeOwnerHandler :
        AuthorizationHandler<IsRecipeOwnerRequirement, Recipe>         #A
{
    private readonly UserManager<ApplicationUser> _userManager;    #B
    public IsRecipeOwnerHandler(                                   #B
        UserManager<ApplicationUser> userManager)                  #B
    {                                                              #B
        _userManager = userManager;                                #B
    }                                                              #B
    protected override async Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        IsRecipeOwnerRequirement requirement,
        Recipe resource)                                          #C
    {
        var appUser = await _userManager.GetUserAsync(context.User);
        if(appUser == null)                                         #D
        {
            return;
        }
        if(resource.CreatedById == appUser.Id)      #E
        {
            context.Succeed(requirement);           #F
        }
    }
}
```

#A Implements the necessary base class, specifying the requirement and resource type
#B Injects an instance of the UserManager<T> class using DI
#C As well as the context and requirement, you're also provided the resource instance.
#D If you aren't authenticated, then appUser will be null.
#E Checks whether the current user created the Recipe by checking the CreatedById property
#F If the user created the document, Succeed the requirement; otherwise, do nothing.

This handler is slightly more complicated than the examples you've seen previously, primarily because you're using an additional service, `UserManager<>`, to load the `ApplicationUser` entity based on `ClaimsPrincipal` from the request.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

NOTE In practice, the `ClaimsPrincipal` will likely already have the `Id` added as a claim, making the extra step unnecessary in this case. This example shows the general pattern if you need to use dependency injected services.

The other significant difference is that the `HandleRequirementAsync` method has provided the `Recipe` resource as a method argument. This is the same object that you provided when calling `AuthorizeAsync` on `IAuthorizationService`. You can use this resource to verify whether the current user created it. If so, you `Succeed()` the requirement, otherwise you do nothing.

The final task is to add `IsRecipeOwnerHandler` to the DI container. Your handler uses an additional dependency, `UserManager<>`, which uses EF Core, so you should register the handler as a scoped service:

```
services.AddScoped<IAuthorizationHandler, IsRecipeOwnerHandler>();
```

TIP If you're wondering how to know whether you register a handler as scoped or a singleton, think back to chapter 10. Essentially, if you have scoped dependencies, then you must register the handler as scoped; otherwise, singleton is fine.

With everything hooked up, you can take the application for a spin. If you try to edit a recipe you didn't create by clicking the Edit button on the recipe, you'll either be redirected to the login page (if you hadn't yet authenticated) or you'll be presented with an Access Denied page, as shown in figure 15.7.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

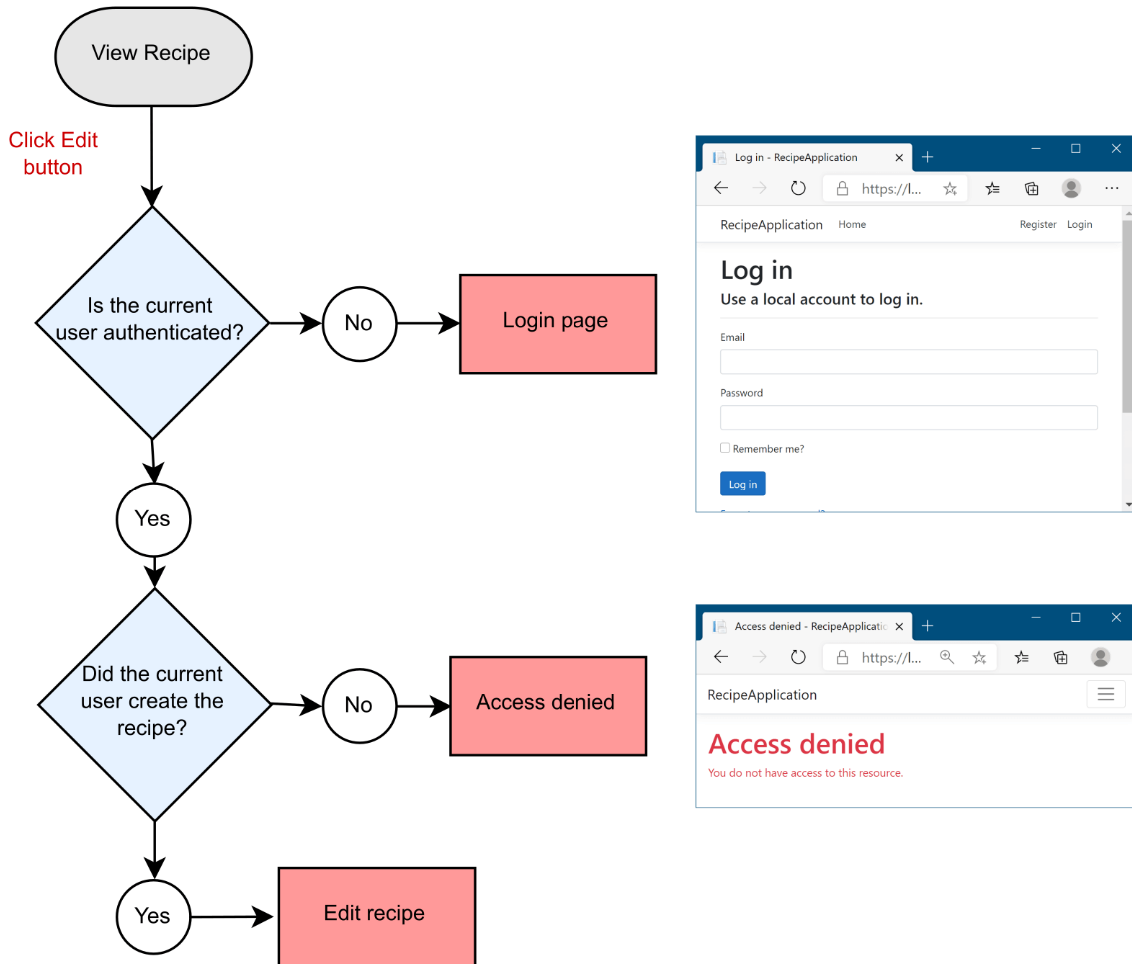


Figure 15.7 If you're logged in but not authorized to edit a recipe, you'll be redirected to an Access Denied page. If you're not logged in, you'll be redirected to the login page.

By using resource-based authorization, you're able to enact more fine-grained authorization requirements that you can apply at the level of an individual document or resource. Instead of only being able to authorize that a user can edit *any* recipe, you can authorize whether a user can edit *this* recipe.

All the authorization techniques you've seen so far have focused on server-side checks. Both the `[Authorize]` attribute and resource-based authorization approaches focus on stopping users from executing a protected action on the server. This is important from a

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

security point of view, but there's another aspect you should consider too: the user experience when they don't have permission.

Now you've protected the code executing on the server, but arguably, the Edit button should never have been visible to the user if they weren't going to be allowed to edit the recipe! In the next section, we'll look at how you can conditionally hide the Edit button by using resource-based authorization in your view models.

15.6 Hiding elements in Razor templates from unauthorized users

All the authorization code you've seen so far has revolved around protecting action methods or Razor Pages on the server side, rather than modifying the UI for users. This is important and should be the starting point whenever you add authorization to an app.

> **WARNING** Malicious users can easily circumvent your UI, so it's important to always authorize your actions and Razor Pages on the server, never on the client alone.

From a user-experience point of view, however, it's not friendly to have buttons or links that look like they're available, but which present you with an Access Denied page when they're clicked. A better experience would be for the links to be disabled, or not visible at all.

You can achieve this in several ways in your own Razor templates. In this section, I'm going to show you how to add an additional property to the `PageModel`, called `CanEditRecipe`, which the Razor view template will use to change the rendered HTML.

> **TIP** An alternative approach would be to inject `IAuthorizationService` directly into the view template using the `@inject` directive, as you saw in chapter 10, but you should prefer to keep logic like this in the page handler.

When you're finished, the rendered HTML will look unchanged for recipes you created, but the Edit button will be hidden when viewing a recipe someone else created, as shown in figure 15.8.

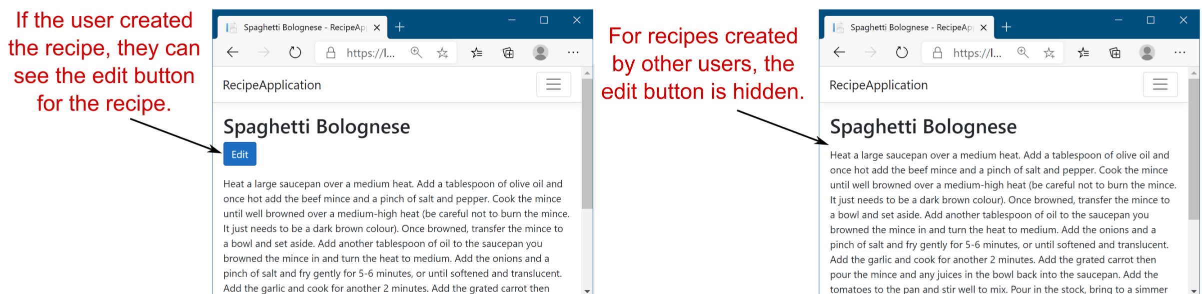


Figure 15.8 Although the HTML will appear unchanged for recipes you created, the Edit button is hidden when you view recipes created by a different user.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

The following listing shows the `PageModel` for the View.cshtml Razor Page, which is used to render the recipe page shown in figure 15.8. As you've already seen for resource-based authorization, you can use the `IAuthorizationService` to determine whether the current user has permission to edit the `Recipe` by calling `AuthorizeAsync`, You can then set this value as an additional property on the `PageModel`, called `CanEditRecipe`.

Listing 15.16 Setting the CanEditRecipe property in the View.cshtml Razor Page

```
public class ViewModel : PageModel
{
    public Recipe Recipe { get; set; }
    public bool CanEditRecipe { get; set; }                 #A

    private readonly RecipeService _service;
    private readonly IAuthorizationService _authService;
    public ViewModel(
        RecipeService service,
        IAuthorizationService authService)
    {
        _service = service;
        _authService = authService;
    }

    public async Task<IActionResult> OnGetAsync(int id)
    {
        Recipe = _service.GetRecipe(id);                         #B
        var isAuthorised = await _authService                    #C
            .AuthorizeAsync(User, recipe, "CanManageRecipe");    #C
        CanEditRecipe = isAuthorised.Succeeded;                  #D
        return Page();
    }
}
```

#A The CanEditRecipe property will be used to control whether the Edit button is rendered.
#B Loads the Recipe resource for use with IAuthorizationService
#C Verifies whether the user is authorized to edit the Recipe
#D Sets the CanEditRecipe property on the PageModel as appropriate

Instead of blocking execution of the Razor Page (as you did previously in the `Edit` action), use the result of the call to `AuthorizeAsync` to set the `CanEditRecipe` value on the `PageModel`. You can then make a simple change to the View.chstml Razor template: add an `if` clause around the rendering of the edit link.

```
@if(Model.CanEditRecipe)
{
    <a asp-action="Edit" asp-route-id="@Model.Id"
        class="btn btn-primary">Edit</a>
}
```

This ensures that only users who will be able to execute the Edit.cshtml Razor Page can see the link to that page.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

> **WARNING** Note that you *didn't* remove the server-side authorization check from the `Edit` action. This is important for preventing malicious users from circumventing your UI.

With that final change, you've finished adding authorization to the recipe application. Anonymous users can browse the recipes created by others, but they must log in to create new recipes. Additionally, authenticated users can only edit the recipes that they created and won't see an edit link for other people's recipes.

Authorization is a key aspect of most apps, so it's important to bear it in mind from an early point. Although it's possible to add authorization later, as you did with the recipe app, it's normally preferable to consider authorization sooner rather than later in the app's development.

In the next chapter, we're going to be looking at your ASP.NET Core application from a different point of view. Instead of focusing on the code and logic behind your app, we're going to look at how you prepare an app for production. You'll see how to specify the URLs your application uses, and how to publish an app so that it can be hosted in IIS. Finally, you'll learn about the bundling and minification of client-side assets, why you should care, and how to use `BundlerMinifier` in ASP.NET Core.

15.7 Summary

- Authentication is the process of determining who a user is. It's distinct from authorization, the process of determining what a user can do. Authentication typically occurs before authorization.
- You can use the authorization services in any part of your application, but it's typically applied using the `AuthorizationMiddleware` by calling `UseAuthorization()`. This should be placed after the calls to `UseRouting()` and `UseAuthentication()`, and before the call to `UseEndpoints()` for correct operation.
- You can protect Razor Pages and MVC actions by applying the `[Authorize]` attribute. The routing middleware records the presence of the attribute as metadata with the selected endpoint. The authorization middleware uses this metadata to determine how to authorize the request.
- The simplest form of authorization requires that a user is authenticated before executing an action. You can achieve this by applying the `[Authorize]` attribute to a Razor Page, action, controller, or globally. You can also apply attributes conventionally to a sub-set of Razor Pages.
- Claims-based authorization uses the current user's claims to determine whether they're authorized to execute an action. You define the claims needed to execute an action in a *policy*.
- Policies have a name and are configured in Startup.cs as part of the call to `AddAuthorization()` in `ConfigureServices`. You define the policy using `AddPolicy()`, passing in a name and a lambda that defines the claims needed.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

- You can apply a policy to an action or Razor Page by specifying the policy in the authorize attribute, for example `[Authorize("CanAccessLounge")]`. This policy will be used by the `AuthorizationMiddleware` to determine if the user is allowed to execute the selected endpoint.
- In a Razor Pages app, if an unauthenticated user attempts to execute a protected action, they'll be redirected to the login page for your app. If they're already authenticated, but don't have the required claims, they'll be shown an Access Denied page instead.
- For complex authorization policies, you can build a custom policy. A custom policy consists of one or more requirements, and a requirement can have one or more handlers. You can combine requirements and handlers to create policies of arbitrary complexity.
- For a policy to be authorized, every requirement must be satisfied. For a requirement to be satisfied, one or more of the associated handlers must indicate success, and none must indicate explicit failure.
- `AuthorizationHandler<T>` contains the logic that determines whether a requirement is satisfied. For example, if a requirement requires that users be over 18, the handler could look for a `DateOfBirth` claim and calculate the user's age.
- Handlers can mark a requirement as satisfied by calling `context.Succeed(requirement)`. If a handler can't satisfy the requirement, then it shouldn't call anything on the context, as a different handler could call `Succeed()` and satisfy the requirement.
- If a handler calls `context.Fail()`, then the requirement will fail, even if a different handler marked it as a success using `Succeed()`. Only use this method if you want to override any calls to `Succeed()` from other handlers, to ensure the authorization policy will fail authorization.
- Resource-based authorization uses details of the resource being protected to determine whether the current user is authorized. For example, if a user is only allowed to edit their own documents, then you need to know the author of the document before you can determine whether they're authorized.
- Resource-based authorization uses the same policy, requirements, and handler system as before. Instead of applying authorization with the `[Authorize]` attribute, you must manually call `IAuthorizationService` and provide the resource you're protecting.
- You can modify the user interface to account for user authorization by adding additional properties to your `PageModel`. If a user isn't authorized to execute an action, you can remove or disable the link to that action method in the UI. You should always authorize on the server, even if you've removed links from the UI.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

16
Publishing and deploying your application

This chapter covers

- **Publishing an ASP.NET Core application**
- **Hosting an ASP.NET Core application in IIS**
- **Customizing the URLs for an ASP.NET Core app**
- **Optimizing client-side assets with bundling and minification**

We've covered a vast amount of ground so far in this book. We've gone over the basic mechanics of building an ASP.NET Core application, such as configuring dependency injection, loading app settings, and building a middleware pipeline. We've looked at the UI side, using Razor templates and layouts to build an HTML response. And we've looked at higher-level abstractions, such as EF Core and ASP.NET Core Identity, that let you interact with a database and add users to your application.

In this chapter, we're taking a slightly different route. Instead of looking at ways to build bigger and better applications, we focus on what it means to deploy your application so that users can access it. You'll learn about

- The process of publishing an ASP.NET Core application so that it can be deployed to a server.
- How to prepare a reverse proxy (IIS) to host your application.
- How to optimize your app so that it's performant once deployed.

We start by looking again at the ASP.NET Core hosting model in section 16.1 and examining why you might want to host your application behind a reverse proxy instead of exposing your app directly to the internet. I show you the difference between running an ASP.NET Core app

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

in development using `dotnet run` and publishing the app for use on a remote server. Finally, I describe some of the options available to you when deciding how and where to deploy your app.

In section 16.2, I show you how to deploy your app to one such option, a Windows server running IIS (Internet Information Services). This is a typical deployment scenario for many developers already familiar with ASP.NET, so it acts as a useful case study, but it's certainly not the only possibility. I won't go into all the technical details of configuring the venerable IIS system, but I'll show the bare minimum required to get it up and running. If your focus is cross-platform development, then don't worry, I don't dwell on IIS for too long!

In section 16.3, I provide an introduction to hosting on Linux. You'll see how it differs from hosting applications on Windows, learn the changes you need to make to your apps, and find out about some gotchas to look out for. I describe how reverse proxies on Linux differ from IIS and point you to some resources you can use to configure your environments, rather than giving exhaustive instructions in this book.

If you're *not* hosting your application using IIS, then you'll likely need to set the URL that your ASP.NET Core app is using when you deploy your application. In section 16.4, I show two approaches to this: using the special `ASPNETCORE_URLS` environment variable and using command line arguments. Although generally not an issue during development, setting the correct URLs for your app is critical when you need to deploy it.

In the final section of this chapter, we look at a common optimization step used when deploying your application. Bundling and minification are used to reduce the number and size of requests that browsers must make to your app to fully load a page. I show you how to use a simple tool to create bundles when you build your application, and how to conditionally load these when in production to optimize your app's page size.

This chapter covers a relatively wide array of topics, all related to deploying your app. But before we get into the nitty-gritty, I'll go over the hosting model for ASP.NET Core so that we're all on the same page. This is significantly different from the hosting model of the previous version of ASP.NET, so if you're coming from that background, it's best to try to forget what you know!

16.1 Understanding the ASP.NET Core hosting model

In this section I discuss the various ways to deploy your ASP.NET Core applications to production. I start by describing the role of a reverse proxy in deployments, and whether you should use one. I then describe the difference between running an app during development and publishing an app for production. Finally, I briefly discuss the various options available for deploying your application to a production server.

If you think back to chapter 1, you may remember that we discussed the hosting model of ASP.NET Core. ASP.NET Core applications are, essentially, console applications. They have a `static void Main` function that serves as the entry point for the application, like a standard .NET console app would.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

What makes an app an ASP.NET Core app is that it runs a web server, typically Kestrel, inside the console app process. Kestrel provides the HTTP functionality to receive requests and return responses to clients. Kestrel passes any requests it receives to the body of your application to generate a response, as shown in figure 16.1. This hosting model decouples the server and reverse proxy from the application itself, so that the same application can run unchanged in multiple environments.

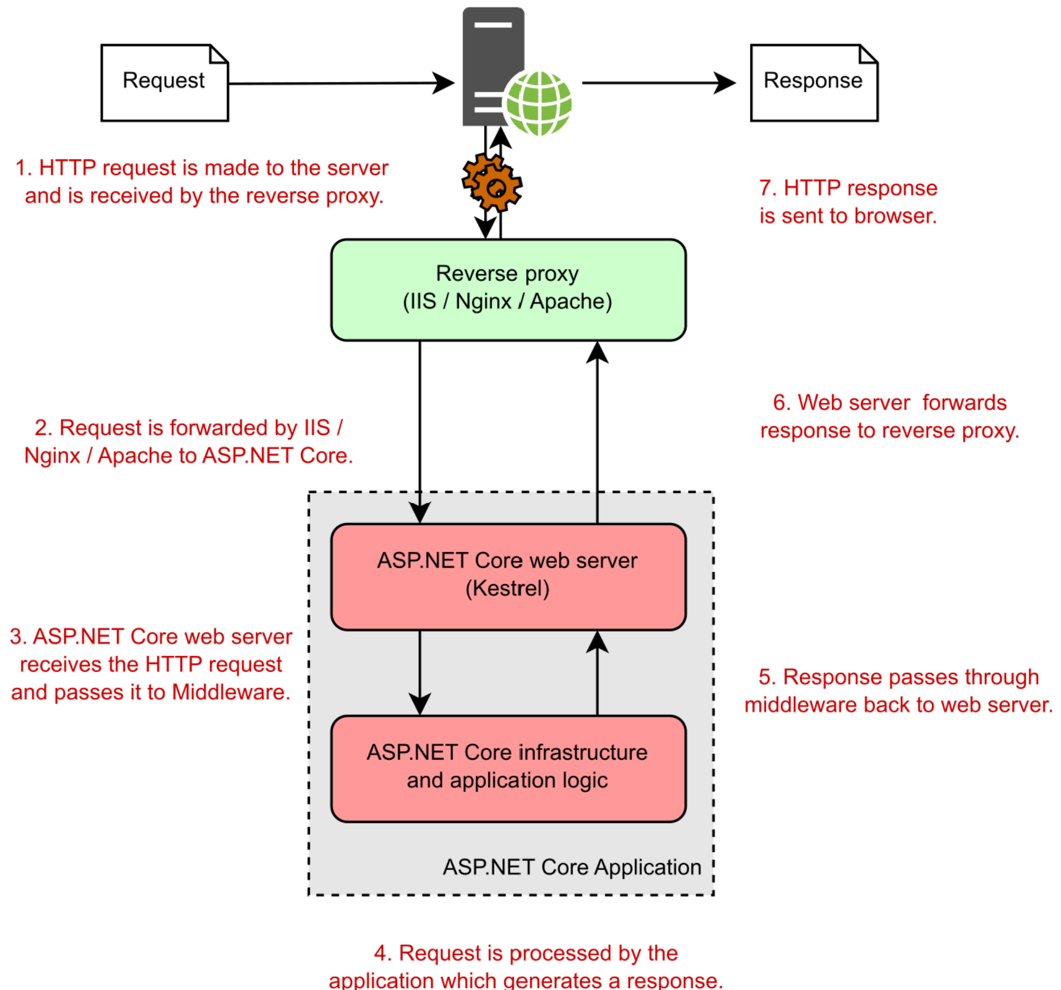


Figure 16.1 The hosting model for ASP.NET Core. Requests are received by the reverse proxy and are forwarded to the Kestrel web server. The same application can run behind various reverse proxies without modification.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

In this book, we've focused on the lower half of figure 16.1—the ASP.NET Core application itself—but the reality is that you'll often want to place your ASP.NET Core apps behind a reverse proxy, such as IIS on Windows, or NGINX or Apache on Linux. The *reverse proxy* is the program that listens for HTTP requests from the internet and then makes requests to your app as though the request had come from the internet directly.

> **DEFINITION** A *reverse proxy* is software that's responsible for receiving requests and forwarding them to the appropriate web server. The reverse proxy is exposed directly to the internet, whereas the underlying web server is exposed only to the proxy.

If you're running your application using a Platform as a Service (PAAS) offering, such as Azure App Service, then you're using a reverse proxy there too, one that is managed by Azure. Using a reverse proxy has many benefits:

- *Security*—Reverse proxies are specifically designed to be exposed to malicious internet traffic, so they're typically well-tested and battle-hardened.
- *Performance*—You can configure reverse proxies to provide performance improvements by aggressively caching responses to requests.
- *Process management*—An unfortunate reality is that apps sometimes crash. Some reverse proxies can act as a monitor/scheduler to ensure that if an app crashes, the proxy can automatically restart it.
- *Support for multiple apps*—It's common to have multiple apps running on a single server. Using a reverse proxy makes it easier to support this scenario by using the host name of a request to decide which app should receive the request.

I don't want to make it seem like using a reverse proxy is all sunshine and roses. There are some downsides:

- *Complexity*— One of the biggest complaints is how complex reverse proxies can be. If you're managing the proxy yourself (as opposed to relying on a PaaS implementation) there can be lots of proxy-specific pitfalls to look out for.
- *Inter process communication*—Most reverse proxies require two processes: a reverse proxy and your web app. Communicating between the two is often slower than if you directly expose your web app to requests from the internet.
- *Restricted features*—Not all reverse proxies support all the same features as an ASP.NET Core app. For example, Kestrel supports HTTP/2, but if your reverse proxy doesn't, then you won't see the benefits.

Whether you choose to use a reverse proxy or not, when the time comes to host your app, you can't copy your code files directly on to the server. First, you need to *publish* your ASP.NET Core app, to optimize it for production. In section 16.1.1, we look at building an ASP.NET Core app so it can be run on your development machine, compared to publishing it so that it can be run on a server.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

16.1.1 Running vs. publishing an ASP.NET Core app

One of the key changes in ASP.NET Core from previous versions of ASP.NET is making it easy to build apps using your favorite code editors and IDEs. Previously, Visual Studio was required for ASP.NET development, but with the .NET CLI and the OmniSharp plugin, you can now build apps with the tools you're comfortable with, on any platform.

As a result, whether you build using Visual Studio or the .NET CLI, the same tools are being used under the hood. Visual Studio provides an additional GUI, functionality, and wrappers for building your app, but it executes the same commands as the .NET CLI behind the scenes.

As a refresher, you've used four main .NET CLI commands so far to build your apps:

- `dotnet new`—Creates an ASP.NET Core application from a template
- `dotnet restore`—Downloads and installs any referenced NuGet packages for your project
- `dotnet build`—Compiles and builds your project
- `dotnet run`—Executes your app, so you can send requests to it

If you've ever built a .NET application, whether it's an ASP.NET app or a .NET Framework Console app, then you'll know that the output of the build process is written to the bin folder. The same is true for ASP.NET Core applications.

If your project compiles successfully when you call `dotnet build`, then the .NET CLI will write its output to a bin folder in your project's directory. Inside this bin folder are several files required to run your app, including a dll file that contains the code for your application. Figure 16.2 shows part of the output of the bin folder for an ASP.NET Core application:

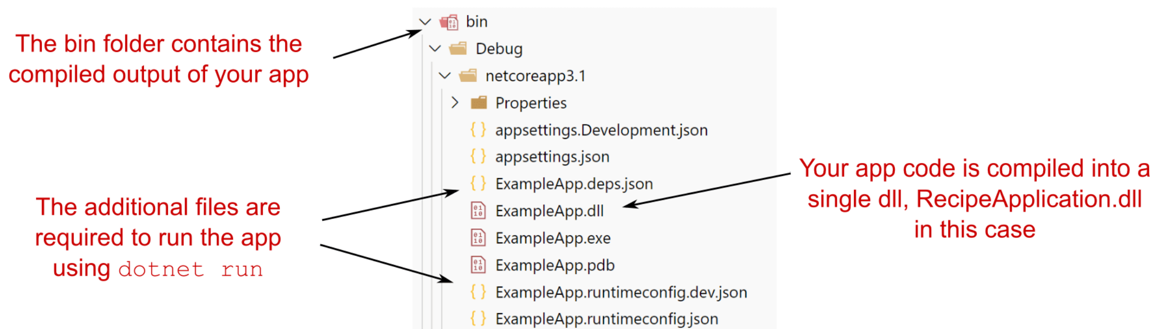


Figure 16.2 The bin folder for an ASP.NET Core app after running `dotnet build`. The application is compiled into a single dll, ExampleApp.dll.

> **NOTE** On Windows, you will also have an executable .exe file, ExampleApp.exe. This is a simple wrapper file for convenience that makes it easier to run the application contained in ExampleApp.dll.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

When you call `dotnet run` in your project folder (or run your application using Visual Studio), the .NET CLI uses the DLL to run your application. But this doesn't contain everything you need to deploy your app.

To host and deploy your app on a server, you first need to *publish* it. You can publish your ASP.NET Core app from the command line using the `dotnet publish` command. This builds and packages everything your app needs to run. The following command packages the app from the current directory and builds it to a subfolder called publish. I've used the `Release` configuration, instead of the default `Debug` configuration, so that the output will be fully optimized for running in production:

```
dotnet publish --output publish --configuration release
```

TIP **Always use the release configuration when publishing your app for deployment. This ensures the compiler generates optimized code for your app.**

Once the command completes, you'll find your published application in the publish folder, as shown in figure 16.3.

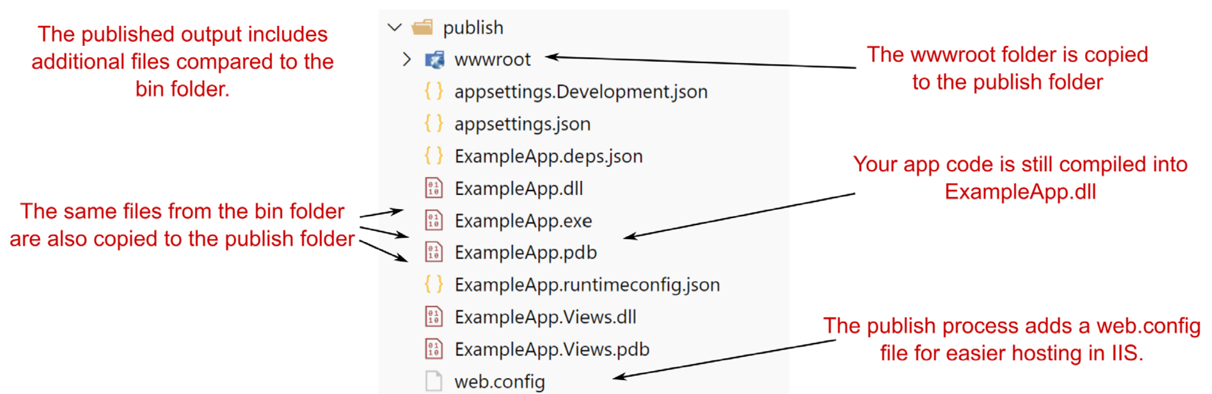


Figure 16.3 The publish folder for the app after running `dotnet publish`. The app is still compiled into a single dll, but all the additional files, such as wwwroot and appsettings.json are also copied to the output.

As you can see, the ExampleApp.dll file is still there, along with some additional files. Most notably, the publish process has copied across the wwwroot folder of static files. When running your application locally with `dotnet run`, the .NET CLI uses these files from your application's project folder application directly. Running `dotnet publish` copies the files to the output directory, so they're included when you deploy your app to a server.

If your first instinct is to try running the application in the publish folder using the `dotnet run` command you already know and love, then you'll be disappointed. Instead of the

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

application starting up, you're presented with a somewhat confusing message: "Couldn't find a project to run."

To run a published application, you need to use a slightly different command. Instead of calling `dotnet run`, you must pass the path to your application's dll file to the `dotnet` command. If you're running the command from the publish folder, then for the example app in figure 16.3, that would look something like

```
dotnet ExampleApp.dll
```

This is the command that your server will run when running your application in production. When you're developing, the `dotnet run` command does a whole load of work to make things easier on you: it makes sure your application is built, looks for a project file in the current folder, works out where the corresponding dlls will be (in the bin folder), and finally, runs your app.

In production, you don't need any of this extra work. Your app is already built, it only needs to be run. The `dotnet <dll>` syntax does this alone, so your app starts much faster.

> **NOTE** The `dotnet` command used to run your published application is part of the .NET Core Runtime, whereas the `dotnet` command used to build and run your application during development is part of the .NET Core SDK.

Framework-dependent deployments vs. self-contained deployments

.NET Core applications can be deployed in two different ways: runtime-dependent deployments (RDD) and self-contained deployments (SCD).

Most of the time, you'll use an RDD. This relies on the .NET Core runtime being installed on the target machine that runs your published app, but you can run your app on any platform—Windows, Linux, or macOS—without having to recompile.

In contrast, an SCD contains *all* the code required to run your app, so the target machine doesn't need to have .NET Core installed. Instead, publishing your app will package up the .NET Core runtime with your app's code and libraries.

Each approach has its pros and cons, but in most cases I tend to create RDDs. The final size of RDDs is much smaller, as they only contain your app code, instead of the whole .NET Core framework, as for SCDs. Also, you can deploy your RDD apps to any platform, whereas SCDs must be compiled specifically for the target machine's operating system, such as Windows 10 64-bit or Red Hat Enterprise Linux 64-bit.

In this book, I'll only discuss RDDs, but if you want to create an SCD, provide a runtime identifier, in this case Windows 10 64-bit, when you publish your app:

```
dotnet publish -c Release -r win10-x64 -o publish_folder
```

The output will contain an exe file, which is your application, and a *ton* of dlls (almost 100 MB of dlls for a sample app), which are the .NET Core framework. You need to deploy this whole folder to the target machine to run your app. For more details, see the documentation at https://docs.microsoft.com/dotnet/core/deploying/.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

We've established that publishing your app is important for preparing it to run in production, but how do you go about deploying it? How do you get the files from your computer on to a server so that people can access your app? You have many, many options for this, so in the next section, I'll give you a brief list of approaches to consider.

16.1.2 Choosing a deployment method for your application

To deploy any application to production, you generally have two fundamental requirements:

- A server that can run your app
- A means of loading your app on to the server

Historically, putting an app into production was a laborious, and error-prone, process. For many people, this is still true. If you're working at a company that hasn't changed practices in recent years, then you may need to request a server or virtual machine for your app and provide your application to an operations team who will install it for you. If that's the case, you may have your hands tied regarding how you deploy.

For those who have embraced continuous integration (CI) or continuous delivery/deployment (CD), there are many more possibilities. CI/CD is the process of detecting changes in your version control system (for example Git, SVN, Mercurial, Team Foundation Version Control) and automatically building, and potentially deploying, your application to a server, with little to no human intervention.[68]

You can find many different CI/CD systems out there: Azure DevOps, GitHub actions, Jenkins, TeamCity, AppVeyor, Travis, and Octopus Deploy, to name a few. Each can manage some or all of the CI/CD process and can integrate with many different systems.

Rather than pushing any particular system, I suggest trying out some of the services available and seeing which works best for you. Some are better suited to open source projects, some are better when you're deploying to cloud services—it all depends on your particular situation.

If you're getting started with ASP.NET Core and don't want to have to go through the setup process of getting CI working, then you still have lots of options. The easiest way to get started with Visual Studio is to use the built-in deployment options. These are available from Visual Studio via the Build > Publish *AppName* menu option, which presents you with the screen shown in figure 16.4.

[68] There are important but subtle differences between these terms. You can find a good comparison here: https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

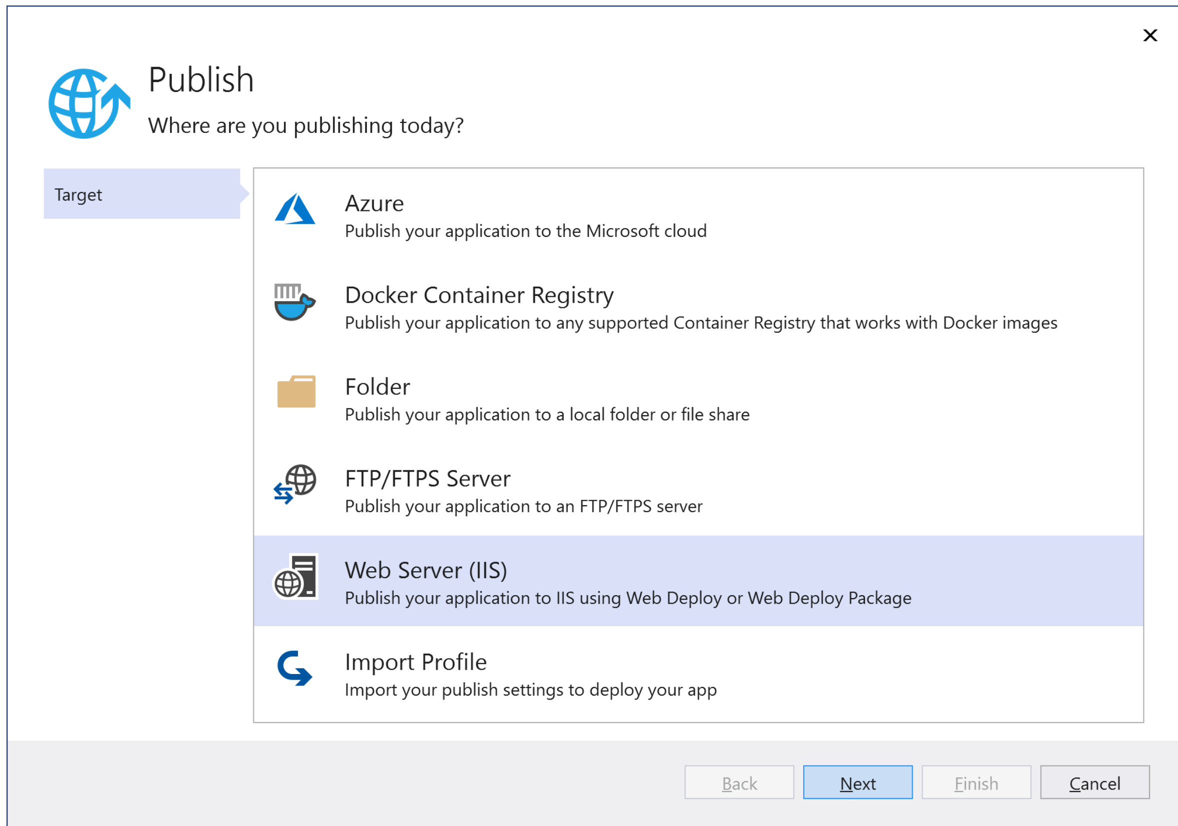


Figure 16.4 The Publish application screen in Visual Studio 2019. This provides easy options for publishing your application directly to Azure App Service, to IIS, to an FTP site, or to a folder on the local machine.

From here, you can publish your application directly from Visual Studio to many different locations.[69] This is great when you're getting started, though I recommend looking at a more automated and controlled approach for larger applications, or when you have a whole team working on a single app.

Given the number of possibilities available in this space, and the speed with which these options change, I'm going to focus on one specific scenario in this chapter: you've built an ASP.NET Core application and you need to deploy it; you have access to a Windows server

[69] For guidance on choosing your Visual Studio publishing options, see https://docs.microsoft.com/visualstudio/deployment/deploying-applications-services-and-components-resources.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

that's already serving (previous version) ASP.NET applications using IIS and you want to run your ASP.NET Core app alongside them.

In the next section, you'll see an overview of the steps required to run an ASP.NET Core application in production, using IIS as a reverse proxy. It won't be a master class in configuring IIS (there's so much depth to the 20-year-old product that I wouldn't know where to start!), but I'll cover the basics needed to get your application serving requests.

16.2 Publishing your app to IIS

In this section, I briefly show how to publish your first app to IIS. You'll add an application pool and website to IIS, and ensure your app has the necessary configuration to work with IIS as a reverse proxy. The deployment itself will be as simple as copying your published app to IIS's hosting folder.

In section 16.1, you learned about the need to publish an app before you deploy it, and the benefits of using a reverse proxy when you run an ASP.NET Core app in production. If you're deploying your application to Windows, then IIS will be your reverse proxy, and will be responsible for managing your application.

IIS is an old and complex beast, and I can't possibly cover everything related to configuring it in this book. Nor would you want me to—it would be very boring! Instead, in this section, I provide an overview of the basic requirements for running ASP.NET Core behind IIS, along with the changes you may need to make to your application to support IIS.

If you're on Windows and want to try out these steps locally, then you'll need to manually enable IIS on your development machine. If you've done this with older versions of Windows, nothing much has changed. You can find a step-by-step guide to configuring IIS and troubleshooting tips in the ASP.NET Core documentation at https://docs.microsoft.com/aspnet/core/publishing/iis.

16.2.1 Configuring IIS for ASP.NET Core

The first step in preparing IIS to host ASP.NET Core applications is to install the .NET Core Windows Hosting Bundle.[70] This includes several components needed to run .NET Core apps:

- *The .NET Core Runtime*—Runs your .NET Core application
- *The ASP.NET Core Runtime*—Required to run ASP.NET Core apps
- *The IIS AspNetCore Module*—Provides the link between IIS and your app, so that IIS can act as a reverse proxy

If you're going to be running IIS on your development machine, then make sure to install the bundle as well, otherwise you'll get strange errors from IIS.

---

[70] You can download the bundle from https://dotnet.microsoft.com/permalink/dotnetcore-current-windows-runtime-bundle-installer.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

TIP You only need the Windows Hosting Bundle for running ASP.NET Core behind IIS. However, wherever you're hosting your app, whether in IIS on Windows, or NGINX on Linux, you'll need to have the .NET Core Runtime and ASP.NET Core runtime installed to run runtime-dependent ASP.NET Core apps.

Once you've installed the bundle, you need to configure an *application pool* in IIS for your ASP.NET Core apps. Previous versions of ASP.NET would run in a *managed* app pool that used .NET Framework, but for ASP.NET Core you should create a *No Managed Code* pool. The native ASP.NET Core Module runs inside the pool, which boots the .NET Core runtime itself.

DEFINITION An *application pool* in IIS represents an application process. You can run each app in IIS in a separate application pool to keep them isolated from one another.

To create an unmanaged application pool, right-click Application Pools in IIS and choose Add Application Pool. Provide a name for the app pool in the resulting dialog, for example NetCore, and set the .NET CLR version to No Managed Code, as shown in figure 16.5.

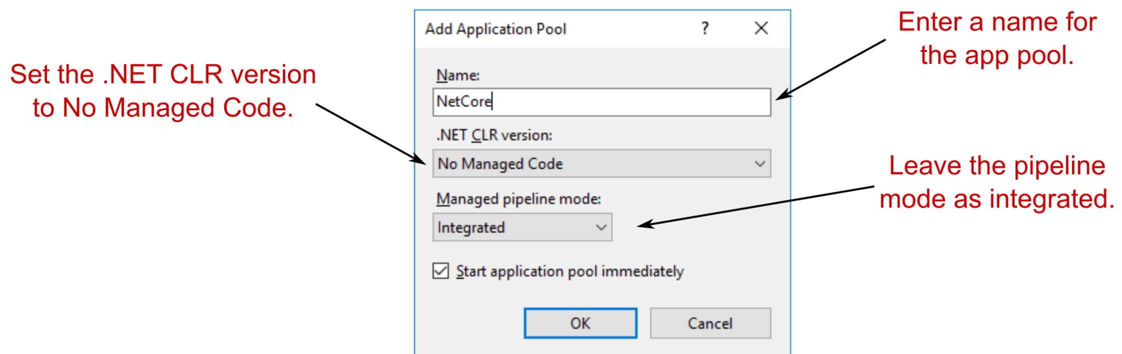


Figure 16.5 Creating an app pool in IIS for your ASP.NET Core app. The .NET CLR version should be set to No Managed Code.

Now you have an app pool, you can add a new website to IIS. Right-click the Sites node and choose Add Website. In the Add Website dialog, shown in figure 16.6, you provide a name for the website and the path to the folder where you'll publish your website. I've created a folder that I'll use to deploy the Recipe app from previous chapters. It's important to change the Application pool for the app to the new NetCore app pool you created. In production, you'd also provide a hostname for the application, but I've left it blank for now and changed the port to 81, so the application will bind to the URL http://localhost:81.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

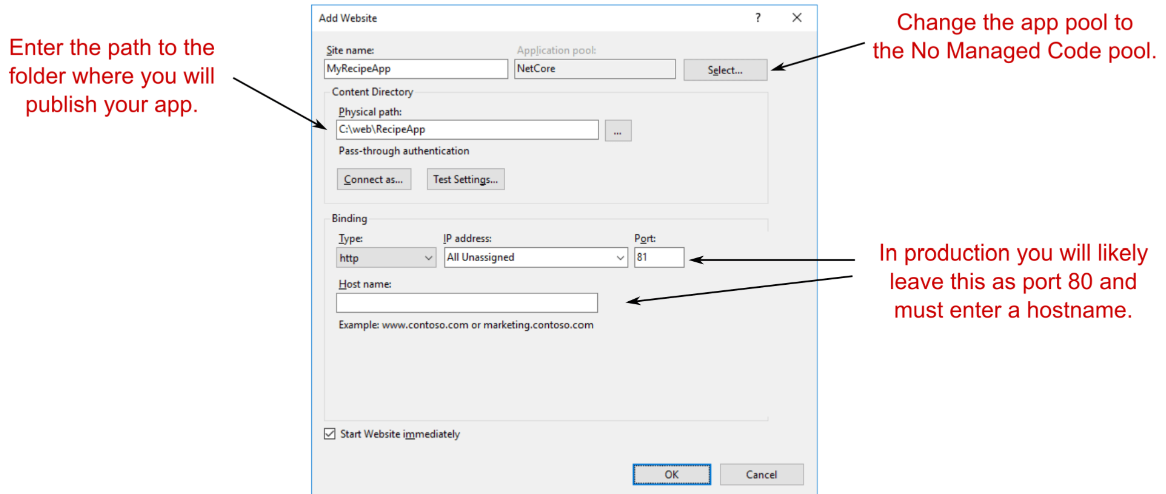


Figure 16.6 Adding a new website to IIS for your app. Be sure to change the Application pool to the No Managed Code pool created in the previous step. You also provide a name, the path where you'll publish your app files, and the URL IIS will use for your app.

> **NOTE** When you deploy an application to production, you need to register a host name with a domain registrar, so your site is accessible by people on the internet. Use that hostname when configuring your application in IIS, as shown in figure 16.6.

Once you click OK, IIS creates the application and attempts to start it. But you haven't published your app to the folder, so you won't be able view it in a browser yet.

You need to carry out one more critical setup step before you can publish and run your app: you must grant permissions for the NetCore app pool to access the path where you'll publish your app. To do this, right-click the folder that will host your app in Windows Explorer and choose Properties. In the Properties dialog, choose Security > Edit > Add… Enter `IIS AppPool\NetCore` in the textbox, as shown in figure 16.7, where NetCore is the name of your app pool, and click OK. Close all the dialog boxes by clicking OK and you're all set.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

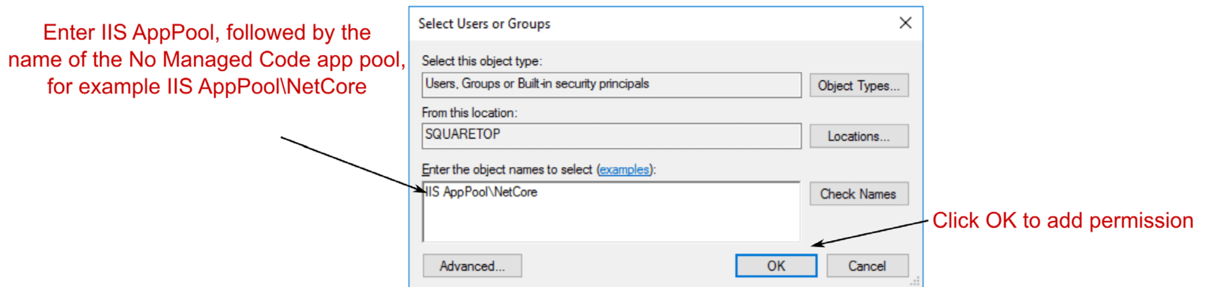


Figure 16.7 Adding permission for the NetCore app pool to the website publish folder.

Out of the box, the ASP.NET Core templates are configured to work seamlessly with IIS, but if you've created an app from scratch, then you may need to make a couple of changes. In the next section, I briefly show the changes you need to make and explain why they're necessary.

16.2.2 Preparing and publishing your application to IIS

As I discussed in section 16.1, IIS acts as a reverse proxy for your ASP.NET Core app. That means IIS needs to be able to communicate directly with your app to forward incoming requests and outgoing responses to and from your app.

IIS handles this with the ASP.NET Core Module, but there's a certain degree of negotiation required between IIS and your app. For this to work correctly, you need to configure your app to use IIS integration.

IIS integration is added by default when you use the `IHostBuilder.ConfigureWebHostDefaults()` helper method used in the default templates. If you're customizing your own `HostBuilder`, then you need to ensure you add IIS integration with the `UseIIS()` or `UseIISIntegration()` extension method.

Listing 16.1 Adding IIS Integration to a host builder

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }
    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHost(webBuilder =>          #A
            {
                webBuilder.UseKestrel();
                webBuilder.UseStartup<Startup>();
                webBuilder.UseIIS();                 #B
                webBuilder.UseIISIntegration();      #C

            });
}
```

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

#A Using a custom builder, instead of the default ConfigureWebHostDefaults used in templates
#B Configures your application for use with IIS with an in-process hosting model
#C Configures your application for use with IIS with an out-of-process hosting model

> **NOTE** If you're not using your application with IIS, the `UseIIS()` and `UseIISIntegration()` methods will have no effect on your app, so it's safe to include them anyway.

In-process vs. out-of-process hosting in IIS

The reverse-proxy description I gave in section 16.1 assumes that your application is running in a separate process to the reverse proxy itself. That is the case if you're running on Linux and was the default for IIS up until ASP.NET Core 3.0. In ASP.NET Core 3.0, ASP.NET Core switched to using an *in-process* hosting model by default for applications deployed to IIS. In this model, IIS hosts your application directly in the IIS process, reducing inter-process communication, and boosting performance.

You can switch to the out-of-process hosting model with IIS if you wish, which can sometimes be useful for troubleshooting problems. Rick Strahl has an excellent post on the differences between the hosting models, how to switch between them, and the advantages of each: https://weblog.west-wind.com/posts/2019/Mar/16/ASPNET-Core-Hosting-on-IIS-with-ASPNET-Core-22.

In general, you shouldn't need to worry about the differences between the hosting models, but it's something to be aware of if you're deploying to IIS. If you choose to use the out-of-process hosting model then you should use the `UseIISIntegration()` extension method. If you use the in-process model, use `UseIIS()`. Alternatively, play it safe and use both—the correct extension method will be activated based on the hosting model used in production. And neither extension does anything if you don't use IIS!

When running behind IIS, these extension methods configure your app to pair with IIS so that it can seamlessly accept requests. Among other things, the extensions:

- Define the URL that IIS will use to forward requests to your app and configure your app to listen on this URL
- Configure your app to interpret requests coming from IIS as coming from the client by setting up header forwarding
- Enable Windows authentication if required

Adding the IIS extension methods is the only change you need to make to your application to be able to host in IIS, but there's one additional aspect to be aware of when you publish your app.

As with previous versions of ASP.NET, IIS relies on a web.config file to configure the applications it runs. It's important your application includes a web.config file when it's

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

published to IIS, otherwise you could get broken behavior, or even expose files that shouldn't be exposed.[71]

If your ASP.NET Core project already includes a web.config file, the .NET CLI or Visual Studio will copy it to the publish directory when you publish your app. If your app doesn't include a web.config file, the publish command will create the correct one for you. If you don't need to customize the web.config file, it's generally best not to include one in your project and let the CLI create the correct file for you.

With these changes, you're finally in a position to publish your application to IIS. Publish your ASP.NET Core app to a folder, either from Visual Studio, or with the .NET CLI by running

```
dotnet publish --output publish_folder --configuration release
```

This will publish your application to the publish_folder folder. You can then copy your application to the path specified in IIS as shown in figure 16.6. At this point, if all has gone smoothly, you should be able to navigate to the URL you specified for your app (http://localhost:81, in my case) and see it running, as shown in figure 16.8.

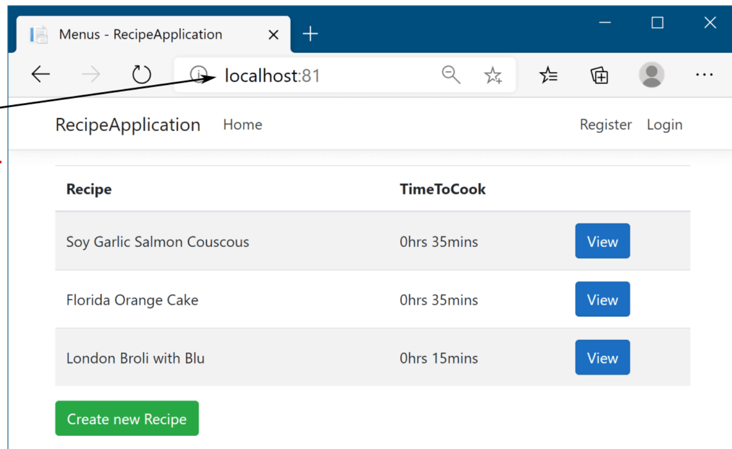


Figure 16.8 The published application, using IIS as a reverse proxy listening at the URL http://localhost:81.

And there you have it, your first application running behind a reverse proxy. Even though ASP.NET Core uses a different hosting model to previous versions of ASP.NET, the process of configuring IIS is similar.

[71] For details on using web.config to customize the IIS AspNetCore Module, see https://docs.microsoft.com/aspnet/core/host-and-deploy/aspnet-core-module.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

As is often the case when it comes to deployment, the success you have is highly dependent on your precise environment and your app itself. If, after following these steps, you find you can't get your application to start, I highly recommend checking out the documentation at https://docs.microsoft.com/aspnet/core/publishing/iis. This contains many troubleshooting steps to get you back on track if IIS decides to throw a hissy fit.

This section was deliberately tailored to deploying to IIS, as it provides a great segue for developers who are already used to deploying ASP.NET apps and want to deploy their first ASP.NET Core app. But that's not to say that IIS is the only, or best, place to host your application.

In the next section, I'll provide a brief introduction to hosting your app on Linux, behind a reverse proxy like NGINX or Apache. I won't go into configuration of the reverse proxy itself, but will provide an overview of things to consider, and resources you can use to run your applications on Linux.

16.3 Hosting an application on Linux

One of the great new features in ASP.NET Core is the ability to develop and deploy applications cross-platform, whether that be on Windows, Linux, or macOS. The ability to run on Linux, in particular, opens up the possibility of cheaper deployments to cloud hosting, deploying to small devices like a Raspberry Pi or to Docker containers.

One of the characteristics of Linux is that it's almost infinitely configurable. Although that's definitely a feature, it can also be extremely daunting, especially if coming from the Windows world of wizards and GUIs. This section provides an overview of what it takes to run an application on Linux. It focuses on the broad steps you need to take, rather than the somewhat tedious details of the configuration itself. Instead, I point to resources you can refer to as necessary.

16.3.1 Running an ASP.NET Core app behind a reverse proxy on Linux

You'll be glad to hear that running your application on Linux is broadly the same as running your application on Windows with IIS:

1. *Publish your app using* `dotnet publish`. If you're creating an RDD, the output is the same as you'd use with IIS. For an SCD, you must provide the target platform moniker, as described in section 16.1.1.
2. *Install the necessary prerequisites on the server.* For an RDD deployment, you must install the .NET Core Runtime and the necessary prerequisites. You can find details on this in the docs at https://docs.microsoft.com/dotnet/core/install/dependencies.
3. *Copy your app to the server.* You can use any mechanism you like, FTP, USB stick, whatever you need to get your files onto the server!
4. *Configure a reverse proxy and point it to your app.* As you know by now, you may want to run your app behind a reverse proxy, for the reasons described in section 16.1. On

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Windows, you'd use IIS, but on Linux you have more options. NGINX, Apache, and HAProxy are all commonly used options.

5. *Configure a process-management tool for your app.* On Windows, IIS acts both as a reverse proxy and a process manager, restarting your app if it crashes or stops responding. On Linux, you typically need to configure a separate process manager to handle these duties; the reverse proxies won't do it for you.

The first three points are generally the same whether you're running on Windows with IIS or on Linux, but the last two points are more interesting. In contrast to the monolithic IIS, Linux has a philosophy of small applications with a single responsibility.

IIS runs on the same server as your app and takes on multiple duties—proxying traffic from the internet to your app, but also monitoring the app process itself. If your app crashes or stops responding, IIS will restart the process to ensure you can keep handling requests.

In Linux, the reverse proxy might be running on the same server as your app, but it's also common for it to be running on a different server entirely, as shown in figure 16.9. This is similarly true if you choose to deploy your app to Docker; your app would typically be deployed in a container without a reverse proxy, and a reverse proxy on a server would point to your Docker container.

As the reverse proxies aren't necessarily on the same server as your app, they can't be used to restart your app if it crashes. Instead, you need to use a process manager such as systemd to monitor your app. If you're using Docker, you'd typically use a container orchestrator such as Kubernetes (https://kubernetes.io) to monitor the health of your containers.

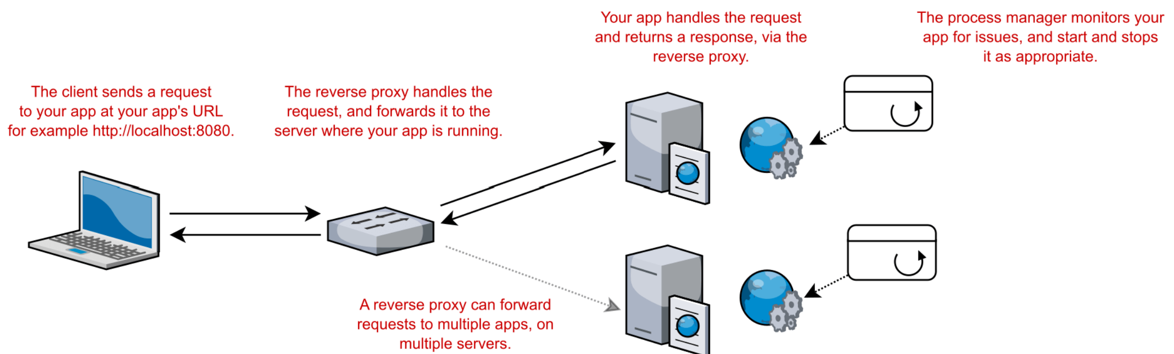


Figure 16.9 On Linux, it's common for a reverse proxy to be on a different server to your app. The reverse proxy forwards incoming requests to your app, while the process manager, for example systemd, monitors your apps for crashes and restarts it as appropriate.

Running ASP.NET Core applications in Docker

Docker is the most commonly used engine for *containerizing* your applications. A container is like a small, lightweight virtual machine, specific to your app. It contains an operating system, your app, and any dependencies for your app.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

This container can then be run on any machine that runs Docker, and your app will run exactly the same, regardless of the host operating system and what's installed on it. This makes deployments highly repeatable: you can be confident that if the container runs on your machine, it will run on the server too.

ASP.NET Core is well-suited to container deployments, but moving to Docker involves a big shift in your deployment methodology, and may or may not be right for you and your apps. If you're interested in the possibilities afforded by Docker and want to learn more, I suggest checking out the following resources:

- *Docker in Practice, Second Edition* by Ian Miell and Aidan Hobson Sayers (Manning, 2017) provides a vast array of practical techniques to help you get the most out of Docker (https://livebook.manning.com/book/docker-in-practice-second-edition/).
- Even if you're not deploying to Linux, you can use Docker with Docker for Windows. Check out the free e-book, *Introduction to Windows Containers* by John McCabe and Michael Friis (Microsoft Press, 2017) from https://aka.ms/containersebook.
- You can find a lot of details on building and running your ASP.NET Core applications on Docker in the .NET documentation at https://docs.microsoft.com/aspnet/core/host-and-deploy/docker.
- Steve Gordon has an excellent blog post series on Docker for ASP.NET Core developers at https://www.stevejgordon.co.uk/docker-dotnet-developers.

Configuring these systems is a laborious task that makes for dry reading, so I won't detail them here. Instead, I recommend checking out the ASP.NET Core docs. They have a guide for NGINX and systemd (https://docs.microsoft.com/aspnet/core/host-and-deploy/linux-nginx), and a guide for configuring Apache with systemd (https://docs.microsoft.com/aspnet/core/host-and-deploy/linux-apache).

Both guides cover the basic configuration of the respective reverse proxy and systemd supervisor, but more importantly, they also show how to configure them *securely*. The reverse proxy sits between your app and the unfettered internet, so it's important to get it right!

Configuring the reverse proxy and the process manager is typically the most complex part of deploying to Linux, and isn't specific to .NET development: the same would be true if you were deploying a Node.js web app. But you need to consider a few things *inside* your application when you're going to be deploying to Linux, as you'll see in the next section.

16.3.2 Preparing your app for deployment to Linux

Generally speaking, your app doesn't care which reverse proxy it sits behind, whether it's NGINX, Apache, or IIS—your app receives requests and responds to them without the reverse proxy affecting things. Just as when you're hosting behind IIS, you need to add `UseIISIntegration()`; when you're hosting on Linux, you need to add a similar extension method to your app setup.

When a request arrives at the reverse proxy, it contains some information that is lost after the request is forwarded to your app. For example, the IP address of the client/browser connecting to your app: once the request is forwarded from the reverse proxy, the IP address is that of the *reverse proxy*, not the *browser*. Also, if the reverse proxy is used for SSL offloading (see chapter 18) then a request that was originally made using HTTPS may arrive at your app as an HTTP request.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

The standard solution to these issues is for the reverse proxy to add additional headers before forwarding requests to your app. For example, the `X-Forwarded-For` header identifies the original client's IP address, while the `X-Forwarded-Proto` header indicates the original scheme of the request (`http` or `https`).

For your app to behave correctly, it needs to look for these headers in incoming requests and modify the request as appropriate. A request to http://localhost with the `X-Forwarded-Proto` header set to `https` should be treated the same as if the request was to https://localhost.

You can use `ForwardedHeadersMiddleware` in your middleware pipeline to achieve this. This middleware overrides `Request.Scheme` and other properties on `HttpContext` to correspond to the forwarded headers. If you're using the default `Host.CreateDefaultBuilder()` method in Program.cs, then this is partially handled for you—the middleware is automatically added to the pipeline in a disabled state. To enable it, set the environment variable ASPNETCORE_FORWARDEDHEADERS_ENABLED=true.

If you're using your own `HostBuilder` instance, instead of the default builder, you can add the middleware to the start of your middleware pipeline manually, as shown in listing 16.2, and configure it with the headers to look for.

> **WARNING** It's important that `ForwardedHeadersMiddleware` is placed early in the middleware pipeline to correct `Request.Scheme` before any middleware that depends on the scheme runs.

Listing 16.2 Configuring an app to use forwarded headers in Startup.cs

```
public class Startup
{
    public class Configure(IApplicationBuilder app)
    {
        app.UseForwardedHeaders(new ForwardedHeadersOptions         #A
        {
            ForwardedHeaders = ForwardedHeaders.XForwardedFor |     #B
                               ForwardedHeaders.XForwardedProto     #B
        });

        app.UseHttpsRedirection();   #C
        app.UseRouting();            #C

        app.UseAuthentication();      #C
        app.UseMvc();                 #C
    }
}
```

#A Adds ForwardedHeadersMiddleware early in your pipeline
#B Configures the headers the middleware should look for and use
#C The forwarded headers middleware must be placed before all other middleware.

> **NOTE** This behavior isn't specific to reverse proxies on Linux; the `UseIis()` extension adds `ForwardedHeadersMiddleware` under the hood as part of its configuration when your app is running behind IIS.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Aside from considering the forwarded headers, you need to consider a few minor things when deploying your app to Linux that may trip you up if you're used to deploying to Windows alone:

- *Line endings (`LF` on Linux versus `CRLF` on Windows).* Windows and Linux use different character codes in text to indicate the end of a line. This isn't often an issue for ASP.NET Core apps, but if you're writing text files on one platform and reading them on a different platform, then it's something to bear in mind.
- *Path directory separator (`"\"` on Windows, `"/"` on Linux).* This is one of the most common bugs I see when Windows developers move to Linux. Each platform uses a different separator in file paths, so while loading a file using the `"subdir\myfile.json"` path will work fine on Windows, it won't on Linux. Instead, you should use `Path.Combine` to create the appropriate separator for the current platform, for example `Path.Combine("subdir", "myfile.json")`.
- *Environment variables can't contain* `":"`. On some Linux distributions, the colon character, `":"`, isn't allowed in environment variables. As you saw in chapter 11, this character is typically used to denote different sections in ASP.NET Core configuration, so you often need to use it in environment variables. Instead, you can use a double underscore in your environment variables (`"__"`) and ASP.NET Core will treat it the same as if you'd used a colon.

As long as you set up `ForwardedHeadersMiddleware` and take care to use cross-platform constructs like `Path.Combine`, you shouldn't have any problems running your applications on Linux. But configuring a reverse proxy isn't the simplest of activities, so wherever you're planning on hosting your app, I suggest checking the documentation for guidance at https://docs.microsoft.com/aspnet/core/publishing.

16.4 Configuring the URLs for your application

At this point, you've deployed an application, but there's one aspect you haven't configured: the URLs for your application. When you're using IIS as a reverse proxy, you don't have to worry about this inside your app. IIS integration with the ASP.NET Core Module works by dynamically creating a URL that's used to forward requests between IIS and your app. The hostname you configure in IIS (in figure 16.6) is the URL that external users see for your app; the internal URL that IIS uses when forwarding requests is never exposed.

If you're not using IIS as a reverse proxy, maybe you're using NGINX or exposing your app directly to the internet, you may find you need to configure the URLs your application listens to directly.

By default, ASP.NET Core will listen for requests on the URL http://localhost:5000. There are lots of ways to set this URL, but in this section I'll describe two: using environment variables or using command line arguments. These are the two most common approaches I see (outside of IIS) to control which URLs your app uses.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

TIP For further ways to set your application's URL, see https://andrewlock.net/5-ways-to-set-the-urls-for-an-aspnetcore-app/.

In chapter 10, you learned about configuration in ASP.NET Core, and in particular about the concept of hosting environments so that you can use different settings when running in development compared to production. You choose the hosting environment by setting an environment variable on your machine called `ASPNETCORE_ENVIRONMENT`. The ASP.NET Core framework magically picks up this variable when your app starts and uses it to set the hosting environment.

You can use a similar special environment variable to specify the URL that your app uses; this variable is called `ASPNETCORE_URLS`. When your app starts up, it looks for this value and uses it as the application's URL. By changing this value, you can change the default URL used by all ASP.NET Core apps on the machine.

For example, you could set a temporary environment variable in Windows from the command window using

```
set ASPNETCORE_URLS=http://localhost:8000
```

Running a published application using `dotnet <app.dll>` within the same command window, as shown in figure 16.10, shows that the app is now listening on the URL provided in the `ASPNETCORE_URLS` variable.

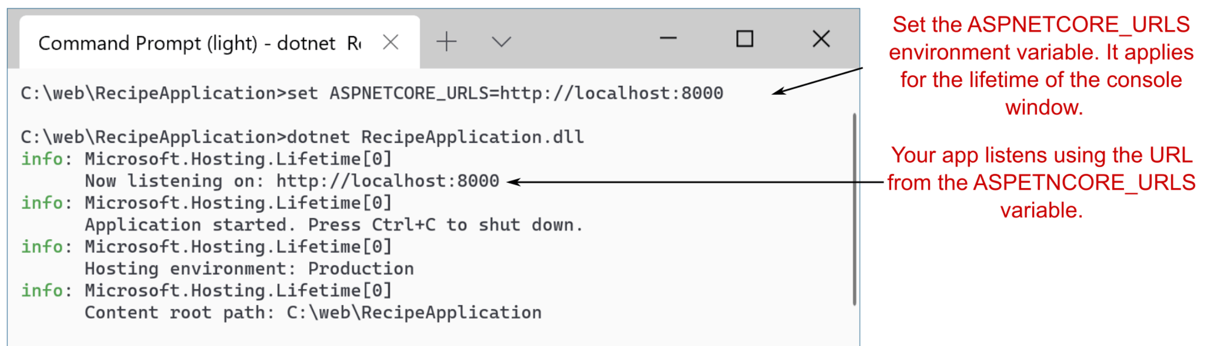


Figure 16.10 Change the `ASPNETCORE_URLS` environment variable to change the URL used by ASP.NET Core apps.

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
```

then your ASP.NET Core apps would listen for requests to:

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

- 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 request 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 files takes presence. 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.[72]

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.

In chapter 11 you saw that you could set the hosting environment using the `ASPNETCORE_ENVIRONMENT` variable, but you could also set the environment using the `--environment` flag when calling `dotnet run`:

```
dotnet run --no-launch-profile --environment Staging
```

You can set the URLs for your application in a similar way, using the `--urls` parameter. Using command line arguments enables you to have multiple ASP.NET Core applications running on the same machine, listening to different ports. For example, the following command would run the recipe application, set it to listen on port 8081, and set the environment to Staging, as shown in figure 16.11:

```
dotnet RecipeApplication.dll --urls "http://*:8081" --environment Staging
```

[72] 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 https://docs.microsoft.com/aspnet/core/host-and-deploy/docker/.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

The command line arguments are used to set both the hosting environment and the URLs.

```
C:\web\RecipeApplication>dotnet RecipeApplication.dll --urls "http://*:8081" --environment Staging
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://[::]:8081
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Staging
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\web\RecipeApplication
```

Figure 16.11 Setting the hosting environment and URLs for an application using command line arguments. The values passed at the command line override values provided from appSettings.json or environment variables.

Remember, you don't need to set your URLs in this way if you're using IIS as a reverse proxy; IIS integration handles this for you. Setting the URLs is only necessary when you're manually configuring the URL your app is listening on, for example if you're using NGINX or are exposing Kestrel directly to clients.

> **WARNING** If running your ASP.NET Core application without a reverse proxy, you should use *host filtering* for security reasons, to ensure your app only responds to requests for hostnames you expect. For more details, see https://andrewlock.net/adding-host-filtering-to-kestrel-in-aspnetcore/.

Continuing the theme of deployment-related tasks, in the next section, we take a look at optimizing some of your client-side assets for production. If you're building a Web API, then this isn't something you'll have to worry about in your ASP.NET Core app, but for traditional web apps it's worth considering.

16.5 Optimizing your client-side assets using BundlerMinifier

In this section, we'll explore the performance of your ASP.NET Core application in terms of the number and size of requests. You'll see how to improve the performance of your app using bundling and minification, but ensuring your app is still easy to debug while you're building it. Finally, we'll look at a common technique for improving app performance in production: using a content delivery network (CDN).

Have you ever used a web app or opened a web page that seemed to take forever to load? Once you stray off the beaten track of Amazon, Google, or Microsoft, it's only a matter of time before you're stuck twiddling your thumbs while the web page slowly pops into place.

Next time this happens to you, open the browser developer tools (for example, press F12 in Edge or Chrome) and take a look at the number, and size, of the requests the web app is making. In many cases, a high number of requests generating large responses will be responsible for the slow loading of a web page.

We'll start by exploring the problem of performance by looking at a single page from your recipe application: the login page. This is a simple page, and it isn't inherently slow, but even this is sufficient to investigate the impact of request size.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

As a user, when you click the login button, the browser sends a request to `/Identity/Account/Login`. Your ASP.NET Core app executes the Login.cshtml Razor Page in the default UI, which executes a Razor template and returns the generated HTML in the response, as shown in figure 16.12. That's a single request-response; a single round-trip.

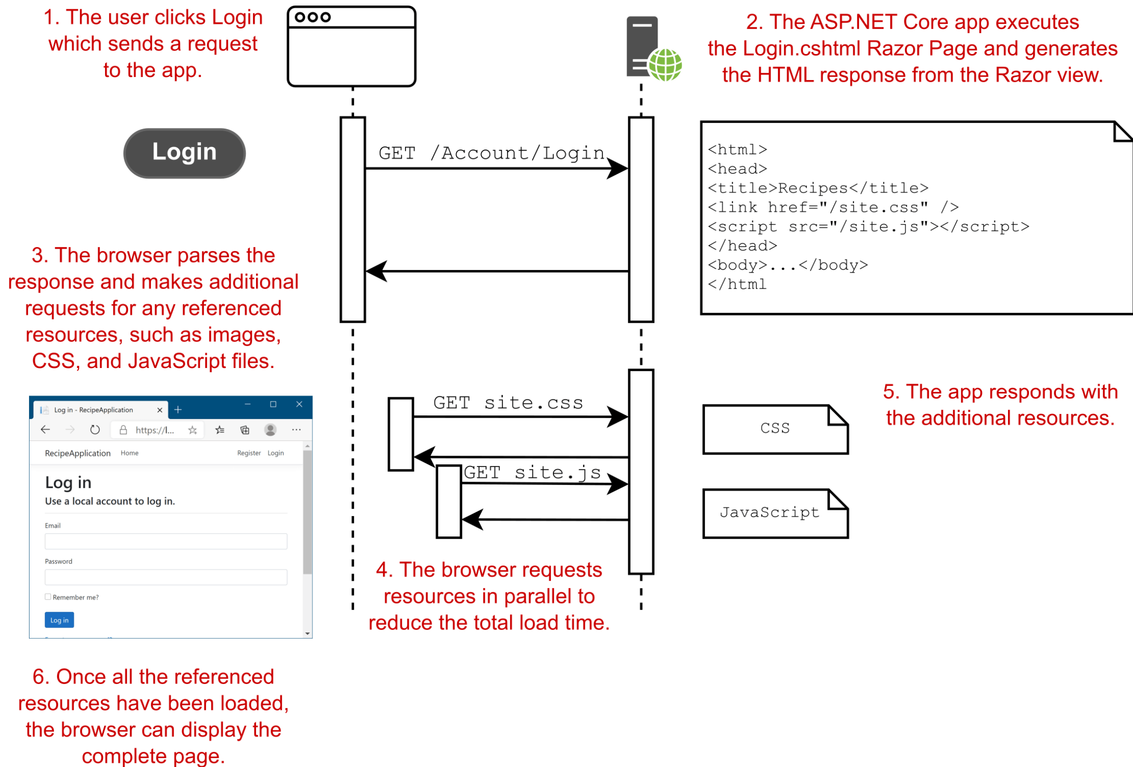


Figure 16.12 Loading a complete page for your app. The initial request returns the HTML for the page, but this may include links to other resources, such as CSS and JavaScript files. The browser must make additional requests to your app for all the outstanding resources before the page can be fully loaded.

But that's not it for the web page. The HTML returned by the page includes links to CSS files (for styling the page), JavaScript files (for client-side functionality—client-side form validation, in this case), and, potentially, images and other resources (though you don't have any others in this recipe app).

The browser must request each of these additional files and wait for the server to return them before the whole page is loaded. When you're developing locally, this all happens quickly, as the request doesn't have far to go at all, but once you deploy your app to production, it's a different matter.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Users will be requesting pages from your app from a wide variety of distances from the server, and over a wide variety of network speeds. Suddenly, the number and size of the requests and responses for your app will have a massive impact on the overall perceived speed of your app. This, in turn, can have a significant impact on how users perceive your site and, for e-commerce sites, even how much money they spend.[73]

A great way to explore how your app will behave for non-optimal networks is to use the network-throttling feature in Chrome's developer tools. This simulates the delays and network speeds associated with different types of networks, so you can get an idea of how your app behaves in the wild. In figure 16.13, I've loaded the login page for the recipe app, but this time with the network set to a modest Fast 3G speed.

> **NOTE** I've added additional files to the template, navigation.css and global.js, to make the page more representative of a real app.

Figure 16.13 Exploring the effect of network speed on application load times. Chrome and Edge let you simulate a slower network, so you can get an impression of the experience users will get when loading your application once it's in production.

Throttling the network doesn't change anything about the page or the data requested—there are 10 separate requests and 1MB loaded for this single page—but it dramatically impacts the time for the page to load. Without throttling, the login page loads locally in 200ms; with Fast 3G throttling, the login page takes 5.47 seconds to load!

---

[73]There has been a lot of research done on this, including stats such as "a 0.1-second delay in page load time equals 7% loss in conversions" from https://www.soasta.com/your-2017-guide-to-retail-performance-success/.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

NOTE Don't be too alarmed by these numbers. I'm making a point of reloading all the files with every request to emphasize the point, whereas in practice, browsers go to great lengths to cache files to avoid having to send this amount of data.

The time it takes to fully load a page of your app is based primarily on two things:

- *The total size of the responses*—This is straight-up math; you can only return data at a certain speed, so the more data you need to return, the longer it takes.
- *The number of requests*—In general, the more requests the browser must make, the longer it takes to fully load the page. In HTTP/1.0 and HTTP/1.1, you can only make six concurrent requests to a server, so any requests after the sixth must wait for an earlier request to *finish* before they can even *start*. HTTP/2.0, which is supported by Kestrel, doesn't have this limit, but you can't always rely on clients using it.

How can you improve your app speed, assuming all the files you're serving are needed? In general, this is a big topic, with lots of possibilities, such as using a CDN to serve your static files. Two of the simplest ways to improve your site's performance are to use *bundling* and *minification* to reduce the number and size of requests the browser must make to load your app.

16.5.1 Speeding up an app using bundling and minification

In figure 16.13 for the recipe app, you made a total of 10 requests to the server:

- One initial request for the HTML
- Three requests for CSS files
- Six requests for JavaScript files

Some of the CSS and JavaScript files are standard vendor libraries, like Bootstrap and jQuery, that are included as part of the default Razor templates, and some (navigation.css, site.css, global.js, and site.js) are files specific to your app. In this section, we're going to look at optimizing your custom CSS and JavaScript files.

If you're trying to reduce the number of requests for your app, then an obvious first thought is to avoid creating multiple files in the first place! For example, instead of creating a navigation.css file and a site.css file, you could use a single file that contains all the CSS, instead of separating it out.

That's a valid solution but putting all your code into one file may make it harder to manage and debug. As developers, we generally try to avoid this sort of monster file. A better solution is to let you split your code into as many files as you want, and then *bundle* the files when you come to deploy your code.

DEFINITION *Bundling* is the process of concatenating multiple files into a single file, to reduce the number of requests.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Similarly, when you write JavaScript, you should use descriptive variables names, comments where necessary, and whitespace to create easily readable and debuggable code. When you come to deploy your scripts, you can process and optimize them for size, instead of readability. This process is called *minification*.

> **DEFINITION** *Minification* involves processing code to reduce its size, without changing the behavior of the code. Processing has many different levels, which typically includes removing comments and whitespace, and can extend to renaming variables to give them shorter names or removing whole sections of code entirely if they're unused.

As an example, look at the JavaScript in the following listing. This (very contrived) function adds up some numbers and returns them. It includes (excessively) descriptive variable names, comments, and plenty of use of whitespace, but is representative of the JavaScript you might find in your own app.

Listing 16.3 Example JavaScript function before minification

```
function myFunc() {
    // this function doesn't really do anything,
    // it's just here so that we can show off minifying!
    function innerFunctionToAddTwoNumbers(
        thefirstnumber, theSecondNumber) {
        // i'm nested inside myFunc
        return thefirstnumber + theSecondNumber;
    }
    var shouldAddNumbers = true;
    var totalOfAllTheNumbers = 0;

    if (shouldAddNumbers == true) {
        for (var index = 0; i < 10; i++) {
            totalOfAllTheNumbers =
                innerFunctionToAddTwoNumbers(totalOfAllTheNumbers, index);
        }
    }
    return totalOfAllTheNumbers;
}
```

This function takes a total of 588 bytes as it's currently written, but after minification, that's reduced to 95 bytes—16% of its original size. The behavior of the code is identical, but the output, shown in the following listing, is optimized to reduce its size. It's clearly not something you'd want to debug, but you'd only use minified versions of your file in production; you'd use the original source files when developing.

Listing 16.4 Example JavaScript function after minification

```
function myFunc(){function r(n,t){return n+t}var
n=0,t;if(1)for(t=0;i<10;i++)n=r(n,t);return n}
```

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Optimizing your static files using bundling and minification can provide a free boost to your app's performance when you deploy your app to production, while letting you develop using easy-to-read and separated files.

Figure 16.14 shows the impact of bundling and minifying the files for the login page of your recipe app. Each of the vendor files has been minified to reduce its size, and your custom assets have been bundled and minified to reduce both their size and the number of requests. This reduced the number of requests from 10 to 8, the total amount of data from 580 KB to 270 KB, and the load time from 6.45 s to 3.15 s.

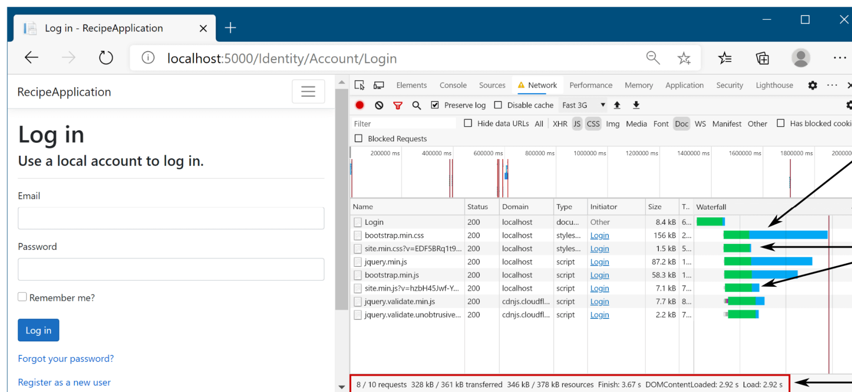


Figure 16.14 By bundling and minifying your client-side resources, you can reduce both the number of requests required and the total data to transfer, which can significantly improve performance. In this example, bundling and minification cut the time to fully load in half.

> **NOTE** The vendor assets, such as jQuery and Bootstrap, aren't bundled with your custom scripts in figure 16.16. This lets you load those files from a CDN, as I'll touch on in section 16.5.4.

This performance improvement can be achieved with little effort on your part, and no impact on your development process. In the next section, I'll show how you can include bundling and minification as part of your ASP.NET Core build process, and how to customize the bundling and minification processes for your app.

16.5.2 Adding BundlerMinifier to your application

Bundling and minification isn't a new idea, so you have many ways to achieve the same result. The previous version of ASP.NET performed bundling and minification in managed code, whereas JavaScript task runners such as gulp, grunt, and webpack are commonly used for these sorts of tasks. In fact, if you're writing a SPA, then you're almost certainly already performing bundling and minification as a matter of course.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

ASP.NET Core includes support for bundling and minification via a NuGet package called BuildBundlerMinifier or a Visual Studio extension version called Bundler & Minifier. You don't have to use either of these, and if you're already using other tools such as gulp or webpack, then I suggest you continue to use them instead. But if you're getting started with a new project, then I suggest considering BundlerMinifier; you can always switch to a different tool later.

You have two options for adding BundlerMinifier to your project:

- You can install the Bundler & Minifier Visual Studio extension from Tools > Extensions and Updates.
- You can add the BuildBundlerMinifier NuGet package to your project.

Whichever approach you use, they both use the same underlying BundlerMinifier library.[74] I prefer to use the NuGet package approach as it's cross-platform and will automatically bundle your resources for you, but the extension is useful for performing ad-hoc bundling. If you do use the Visual Studio extension, make sure to enable Bundle on build, as you'll see shortly.

You can install the BuildBundlerMinifier NuGet package in your project with this command:

```
dotnet add package BuildBundlerMinifier
```

With the BuildBundlerMinifier package installed, whenever you build your project the BundlerMinifier will check your CSS and JavaScript files for changes. If something has changed, new bundled and minified files are created, as shown in figure 16.15, where I modified a JavaScript file.

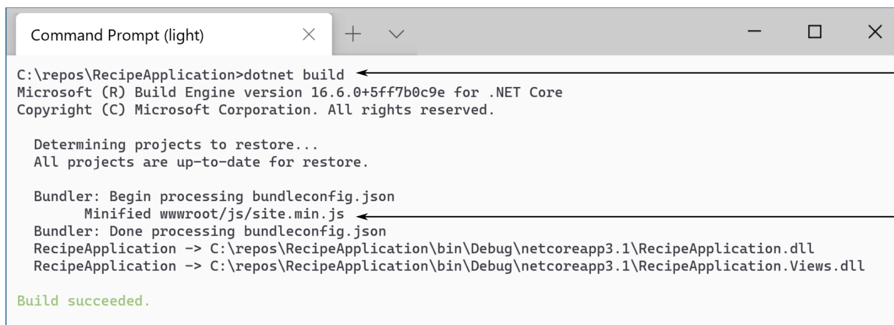


Figure 16.15 Whenever the project is built, the BuildBundlerMinifier tool looks for changes in the input files and builds new bundles as necessary.

[74] The library and extension are open source on GitHub https://github.com/madskristensen/BundlerMinifier/.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

As you can see in figure 16.15, the bundler minified your JavaScript code and created a new file at wwwroot/js/site.min.js. But why did it pick this name? Well, because you told it to in bundleconfig.json. You add this JSON file to the root folder of your project, and it controls the BundlerMinifier process.

Listing 16.5 shows a typical bundleconfig.json configuration for a small app. This defines which files to include in each bundle, where to write each bundle, and what minification options to use. Two bundles are configured here: one for CSS and one for JavaScript. You can add bundles to the JSON array, or you can customize the existing provided bundles.

Listing 16.5 A typical bundleconfig.json file

```
[
  {
    "outputFileName": "wwwroot/css/site.min.css",   #A
    "inputFiles": [                                 #B
      "wwwroot/css/site.css"                        #B
      "wwwroot/css/navigation.css"                  #B
    ]                                               #B
  },
  {
    "outputFileName": "wwwroot/js/site.min.js",     #C
    "inputFiles": [                                 #D
      "wwwroot/js/site.js"
       "wwwroot/js/*.js",                                #E
       "!wwwroot/js/site.min.js"                         #F
    ],
    "minify": {                 #G
      "enabled": true,          #G
      "renameLocals": true      #G
    },
    "sourceMap": false          #H
  }
]
```

#A The bundler will create a file with this name.
#B The files listed in inputFiles are minified and concatenated into outputFileName.
#C You can specify multiple bundles, each with an output filename.
#D You should create separate bundles for CSS and JavaScript.
#E You can use globbing patterns to specify files to include.
#F The ! symbol excludes the matching file from the bundle.
#G The JavaScript bundler has some additional options.
#H You can optionally create a source map for the bundled JavaScript file.

For each bundle you can list a specific set of files to include, as specified in the `inputFiles` property. If you add a new CSS or JavaScript file to your project, you have to remember to come back to bundleconfig.json and add it to the list.

Alternatively, you can use *globbing* patterns to automatically include new files by default. This isn’t always possible, for example when you need to ensure that files are concatenated in a given order, but I find it preferable in many cases. The example in the previous listing uses globbing for the JavaScript bundle: the pattern includes all .js files in the wwwroot/js folder but excludes the minified output file itself.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

DEFINITION *Globbing* patterns use wildcards to represent many different characters. For example, *.css would match all files with a .css extension, whatever the filename.

When you build your project, the BundlerMinifier optimizes your css files into a single wwwroot/css/site.min.css file, and your JavaScript into a single wwwroot/js/ site.min.js file, based on the settings in bundleconfig.json. In the next section, we'll look at how to include these files when you run in production, continuing to use the original files when developing locally.

NOTE The BundlerMinifier package is great for optimizing your CSS and JavaScript resources. But images are another important resource to consider for optimization and can easily constitute the bulk of your page's size. Unfortunately, there aren't any built-in tools for this, so you'll need to look at other options; https://tinypng.com is one.

16.5.3 Using minified files in production with the environment tag helper

Optimizing files using bundling and minification is great for performance, but you want to use the original files during development. The easiest way to achieve this split in ASP.NET Core is to use the Environment Tag Helper to conditionally include the original files when running in the Development hosting environment, and to use the optimized files in other environments.

You learned about the Environment Tag Helper in chapter 8, in which we used it to show a banner on the homepage when the app was running in the Testing environment. Listing 16.6 shows how you can achieve a similar approach in the _Layout .cshtml page for the CSS files of your Recipe app by using two Environment Tag Helpers: one for when you're in Development, and one for when you aren't in Development (you're in Production or Staging, for example). You can use similar Tag Helpers for the JavaScript files in the app.

Listing 16.6 Using Environment Tag Helpers to conditionally render optimized files

```
<environment include="Development">                          #A
    <link rel="stylesheet"                                   #B
        href="~/lib/bootstrap/dist/css/bootstrap.css" />     #B
    <link rel="stylesheet" href="~/css/navigation.css" />    #C
    <link rel="stylesheet" href="~/css/site.css" />          #C
</environment>
<environment exclude="Development">                             #D
    <link rel="stylesheet"                                      #E
        href="~/lib/bootstrap/dist/css/bootstrap.min.css" />    #E
    <link rel="stylesheet" href="~/css/site.min.css" />         #F
</environment>
```

#A Only render these links when running in Development environment.
#B The development version of Bootstrap
#C The development version of your styles
#D Only render these links when not in Development, such as in Staging or Production.
#E The minified version of Bootstrap
#F The bundled and minified version of your styles

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

When the app detects it isn't running in the Development hosting environment (as you learned in chapter 11), it will switch to rendering links for the optimized files. This gives you the best of both worlds: performance in production and ease of development.

The example in listing 16.6 also included a minified version of Bootstrap, even though you didn't configure this as part of the BundlerMinifier. It's common for CSS and JavaScript libraries like Bootstrap to include a pre-minified version of the file for you to use in Production. For those that do, it's often better to exclude them from your bundling process, as this allows you to potentially serve the file from a public CDN.

16.5.4 Serving common files from a CDN

A public CDN is a website that hosts commonly used files, such as Bootstrap or jQuery, which you can reference from your own apps. They have several advantages:

- They're normally fast.
- They save your server having to serve the file, saving bandwidth.
- Because the file is served from a different server, it doesn't count toward the six concurrent requests allowed to your server in HTTP/1.0 and HTTP/1.1.[75]
- Many different apps can reference the same file, so a user visiting your app may have already cached the file by visiting a different website, and may not need to download it at all.

It's easy to use a CDN in principle: reference the CDN file instead of the file on your own server. Unfortunately, you need to cater for the fact that, like any server, CDNs can sometimes fail. If you don't have a fallback mechanism to load the file from a different location, such as your server, then this can result in your app looking broken.

Luckily, ASP.NET Core includes several tag helpers to make working with CDNs and fallbacks easier. Listing 16.7 shows how you could update your CSS Environment Tag Helper to serve Bootstrap from a CDN when running in production, and to include a fallback. The fallback test creates a temporary HTML element and applies a Bootstrap style to it. If the element has the expected CSS properties, then the fallback test passes, because Bootstrap must be loaded. If it doesn't have the required properties, Bootstrap will be loaded from the alternative, local link instead.

Listing 16.7 Serving Bootstrap CSS styles from a CDN with a local fallback

```
<environment include="Development">
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/navigation.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</environment>
```

[75] This limit is not fixed in stone but modern browsers all use the same limit: https://docs.pushtechnology.com/cloud/latest/manual/html/designguide/solution/support/connection_limitations.html

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
<environment exclude="Development">
    <link rel="stylesheet"  href="https://stackpath.bootstrapcdn.com/    #A
[CA] bootstrap/4.3.1/css/bootstrap.min.css"                               #A
    asp-fallback-test-class="sr-only"                                    #B
    asp-fallback-test-property="position"                                #C
    asp-fallback-test-value="absolute"                                   #C
    asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css" />    #D

    <link rel="stylesheet" href="~/css/site.min.css" />
</environment>
```

#A By default, Bootstrap is loaded from a CDN.
#B The fallback test applies the sr-only class to an element ...
#C ... and checks the element has a CSS position of absolute. This indicates Bootstrap was loaded.
#D If the fallback check fails, the CDN must have failed, so Bootstrap is loaded from the local link.

Optimizing your static files is an important step to consider before you put your app into production, as it can have a significant impact on performance. Luckily, the BuildBundlerMinifier package makes it easy to optimize your CSS and JavaScript files. If you couple that with serving common files from a CDN, your app will be as speedy as possible for users in production.

That brings us to the end of this chapter on publishing your app. This last mile of app development, deploying an application to a server where users can access it, is a notoriously thorny issue. Publishing an ASP.NET Core application is easy enough, but the multitude of hosting options available makes providing concise steps for every situation difficult.

Whichever hosting option you choose, there's one critical topic which is often overlooked, but is crucial for resolving issues quickly: logging. In the next chapter, you'll learn about the logging abstractions in ASP.NET Core, and how you can use them to keep tabs on your app once it's in production.

16.6 Summary

- ASP.NET Core apps are console applications that self-host a web server. In production, you typically use a reverse proxy which handles the initial request and passes it to your app. Reverse proxies can provide additional security, operations, and performance benefits, but can also add complexity to your deployments.
- .NET Core has two parts: the .NET Core SDK (also known as the .NET CLI) and the .NET Core Runtime. When you're developing an application, you use the .NET CLI to restore, build, and run your application. Visual Studio uses the same .NET CLI commands from the IDE.
- When you want to deploy your app to production, you need to publish your application, using `dotnet publish`. This creates a folder containing your application as a DLL, along with all its dependencies.
- To run a published application, you don't need the .NET CLI as you won't be building the app. You only need the .NET Core Runtime to run a published app. You can run a

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

published application using the `dotnet app.dll` command, where app.dll is the application dll created by the `dotnet publish` command.

- To host ASP.NET Core applications in IIS, you must install the ASP.NET Core Module. This allows IIS to act as a reverse proxy for your ASP.NET Core app.
- IIS can host ASP.NET Core applications using one of two modes: in-process and out-of-process. The out-of-process mode runs your application as a separate process, as is typical for most reverse proxies. The in-process mode runs your application as part of the IIS process. This has performance benefits, as no inter-process communication is required.
- To prepare your application for publishing to IIS with ASP.NET Core, ensure you call `UseIISIntegration()` and `UseIIS()` on `WebHostBuilder`. The `ConfigureWebHostDefaults` static helper method does this automatically.
- When you publish your application using the .NET CLI, a web.config file will be added to the output folder. It's important that this file is deployed with your application when publishing to IIS, as it defines how your application should be run.
- The URL that your app will listen on is specified by default using the environment variable `ASPNETCORE_URLS`. Setting this value will change the URL for all the apps on your machine. Alternatively, pass the `--urls` command line argument when running your app, for example `dotnet app.dll --urls http://localhost:80`.
- It's important to optimize your client-side assets to reduce the size and number of files that must be downloaded by a client's web browser when a page loads. You can achieve this by bundling and minifying your assets.
- You can use the BuildBundlerMinifier package to combine multiple JavaScript or CSS files together in a process called bundling. You can reduce the size of the files in a process called minification, in which unnecessary whitespace is removed and variables are renamed while preserving the function of the file.
- You can install a Visual Studio extension to control bundling and minification, or you can install the BuildBundlerMinifier package to automatically perform bundling and minification on each build of the project. Using the extension allows you to minify on an ad-hoc basis, but using the NuGet package allows you to automate the process.
- The settings for bundling and minification are stored in the bundleconfig.json file, where you can define the different output bundle files and choose which files to include in the bundle. You can explicitly specify files, or you can use globbing patterns to include multiple files using wildcard characters. Globbing is typically easier and less error prone, but you will need to specify files explicitly if they must be bundled in a specific order.
- Use the Environment Tag Helper to conditionally render your bundles in production only. This lets you optimize for performance in production and readability in development.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

- For common files shared by multiple apps, such as jQuery or Bootstrap, you can serve files from a CDN. These are websites that host the common files, so your app doesn't need to serve them itself.
- Use Link and Script Tag Helpers to check that the file has loaded correctly from the CDN. These can test that a file has been downloaded by a client and ensures that your server is used as a fallback should the CDN fail.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Part 3

Extending your applications

We covered a huge amount of content in parts 1 and 2: we looked at all the main functional components you'll use to build both traditional server-rendered Razor Pages apps, as well as Web APIs. In part 3, we look at six different topics that build on what you've learned so far: logging, security, custom components, interacting with third-party HTTP APIs, background services, and testing.

Adding logging to your application is one of those activities that's often left until after you discover a problem in production. Adding sensible logging from the get-go will help you quickly diagnose and fix errors as they arise. Chapter 17 introduces the logging framework built in to ASP.NET Core. You'll see how you can use it to write log messages to a wide variety of locations, whether it's the console, a file, or a third-party remote-logging service.

Correctly securing your app is an important part of web development these days. Even if you don't feel you have any sensitive data in your application, you have to make sure you protect your users from attacks by adhering to security best practices. In chapter 18, I describe some common vulnerabilities, how attackers can exploit them, and what you can do to protect your applications.

In part 1, you learned about the middleware pipeline, and how it is fundamental to all ASP.NET Core applications. In chapter 19 you'll learn how to create your own custom middleware, as well as simple endpoints, for when you don't need the full power of Razor Pages or a Web API controller. You'll also learn how to handle some complex chicken-and-egg configuration issues that often arise in real-life applications. Finally, you'll learn how to replace the built-in dependency injection container with a third-party alternative.

In chapter 20 you'll learn how to create custom components for working with Razor Pages and API controllers. You'll learn how to create custom Tag Helpers and validation attributes, and I introduce a new component—view components—for encapsulating logic with Razor view rendering. You'll also learn how to replace the attribute-based validation framework used by default in ASP.NET Core with an alternative.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Most apps you build aren't designed to stand on their own. It's very common for your app to need to interact with third-party APIs, whether that's APIs for sending emails, fetching exchange rates, or taking payments. In chapter 21 you'll learn how to interact with third-party APIs using the `IHttpClientFactory` abstraction to simplify configuration, to add transient fault handling, and to avoid common pitfalls.

This book deals primarily with serving HTTP traffic, both server-rendered web pages using Razor Pages, and Web APIs commonly used by mobile and single-page applications. However, many apps require long-running background tasks that execute jobs on a schedule or process items from a queue. In chapter 22 I show how you can create these long background tasks in your ASP.NET Core applications. I also show how to create standalone services that *only* have background tasks, without any HTTP handling, and how to install them as a Windows Service or as a Linux systemd daemon.

Chapter 23, the final chapter, covers testing your application. The exact role of testing in application development can sometimes lead to philosophical arguments, but in chapter 23, I stick to the practicalities of testing your app using the xUnit test framework. You'll see how to create unit tests for your apps, how to test code that's dependent on EF Core using an in-memory database provider, and how to write integration tests that can test multiple aspects of your application at once.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

17 Monitoring and troubleshooting errors with logging

This chapter covers

- **Understanding the components of a log message**
- **Writing logs to multiple output locations**
- **Controlling log verbosity in different environments using filtering**
- **Using structured logging to make logs searchable**

Logging is one of those topics that seems unnecessary, right up until the point when you desperately need it! There's nothing more frustrating than finding an issue that you can only reproduce in production, and then discovering there are no logs to help you debug it.

Logging is the process of recording events or activities in an app and often involves writing a record to a console, a file, the Windows Event Log, or some other system. You can record anything in a log message, though there are generally two different types of message:

- *Informational messages*—A standard event occurred: a user logged in, a product was placed in a shopping cart, or a new post was created on a blogging app.
- *Warnings and errors*—An error or unexpected condition occurred: a user had a negative total in the shopping cart, or an exception occurred.

Historically, a common problem with logging in larger applications was that each library and framework would generate logs in a slightly different format, if at all. When an error occurred in your app and you were trying to diagnose it, this inconsistency made it harder to connect the dots in your app to get the full picture and understand the problem.

Luckily, ASP.NET Core includes a new, generic logging interface that you can plug into. It's used throughout the ASP.NET Core framework code itself, as well as by third-party libraries,

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

and you can easily use it to create logs in your own code. With the ASP.NET Core logging framework, you can control the verbosity of logs coming from each part of your code, including the framework and libraries, and you can write the log output to any destination that plugs into the framework.

In this chapter, I cover the ASP.NET Core logging framework in detail and explain how you can use it to record events and diagnose errors in your own apps. In section 17.1, I describe the architecture of the logging framework. You'll learn how DI makes it easy for both libraries and apps to create log messages, as well as to write those logs to multiple destinations.

In section 17.2, you'll learn how to write your own log messages in your apps with the `ILogger` interface. We'll break down the anatomy of a typical log record and look at its properties, such as the log level, category, and message.

Writing logs is only useful if you can read them, so in section 17.3, you'll learn how to add *logging providers* to your application. Logging providers control where your app writes your log messages. This could be to the console, to a file, or even an external service. I'll show you how to add a logging provider that writes logs to a file, and how to configure a popular third-party logging provider called Serilog in your app.

Logging is an important part of any application, but determining *how much* logging is enough can be a tricky question. On the one hand, you want to provide sufficient information to be able to diagnose any problems. On the other, you don't want to fill your logs with data that makes it hard to find the important information when you need it. Even worse, what is sufficient in development might be far too much once you're running in production.

In section 17.4, I explain how you can filter log messages from various sections of your app, such as the ASP.NET Core infrastructure libraries, so that your logging providers only write the important messages. This lets you keep that balance between extensive logging in development and only writing important logs in production.

In the final section of this chapter, I touch on some of the benefits of *structured logging*, an approach to logging that you can use with some providers for the ASP.NET Core logging framework. Structured logging involves attaching data to log messages as key-value pairs to make logs more easily searched and queried. You might attach a unique customer ID to every log message generated by your app, for example. Finding all the log messages associated with a user is much simpler with this approach, compared to recording the customer ID in an inconsistent manner as part of the log message.

We'll start this chapter by digging into what logging involves, and why your future self will thank you for using logging effectively in your application. Then we'll look at the pieces of the ASP.NET Core logging framework you'll use directly in your apps, and how they fit together.

17.1 Using logging effectively in a production app

In this section I discuss the concept of logging at a high level. You'll see why writing custom log messages can help you diagnose problems in production applications and get a taste for the logging built into the ASP.NET Core framework libraries. Finally, you'll learn about the logging abstractions built into .NET Core that you can use in your own applications.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Imagine you've just deployed a new app to production, when a customer calls saying that they're getting an error message using your app.

How would you identify what caused the problem? You could ask the customer what steps they were taking, and potentially try to recreate the error yourself, but if that doesn't work then you're left trawling through the code, trying to spot errors with nothing else to go on.

Logging can provide the extra context you need to quickly diagnose a problem. Arguably, the most important logs capture the details about the error itself, but the events that led to the error can be just as useful in diagnosing the cause of an error.

There are many reasons for adding logging to an application, but typically, the reasons fall into one of two categories:

- Logging for auditing or analytics reasons, to trace when events have occurred
- Logging errors, or to provide a breadcrumb trail of events when an error does occur

The first of these reasons is simple. You may be required to keep a record of every time a user logs in, for example, or you may want to keep track of how many times a particular API method is called. Logging is an easy way to record the behavior of your app, by writing a message to the log every time an interesting event occurs.

I find the second reason for logging to be the most common. When an app is working perfectly, logs often go completely untouched. It's when there's an issue and a customer comes calling that logs become invaluable. A good set of logs can help you understand the conditions in your app that caused an error, including the context of the error itself, but also the context in previous requests.

> **TIP** Even with extensive logging in place, you may not realize you have an issue in your app unless you look through your logs regularly! For any medium to large app this becomes impractical, so monitoring services such as https://raygun.io or https://sentry.io can be invaluable for notifying you of issues quickly.

If this sounds like a lot of work, then you're in luck. ASP.NET Core does a ton of the "breadcrumb" logging for you so that you can focus on creating high-quality log messages that provide the most value when diagnosing problems.

17.1.1 Highlighting problems using custom log messages

ASP.NET Core uses logging throughout its libraries. Depending on how you configure your app, you'll have access to the details of each request and EF Core query, even without adding additional logging messages to your own code. In figure 17.1 you can see the log messages created when you view a single recipe in the recipe application.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Command Prompt (light) - dotnet ru

```
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/2 GET https://localhost:5001/Recipes/View/5
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint '/Recipes/View'
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[3]
      Route matched with {page = "/Recipes/View", area = ""}. Executing page /Recipes/View
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[101]
      Executing handler method RecipeApplication.Pages.Recipes.ViewModel.OnGetAsync - ModelState is Valid
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core 3.1.3 initialized 'AppDbContext' using provider 'Microsoft.EntityFrameworkCore.Sqlite' with options: None
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (2ms) [Parameters=[@__id_0='?'], CommandType='Text', CommandTimeout='30']
      SELECT "t"."RecipeId", "t"."Name", "t"."Method", "i"."Name", "i"."Quantity", "i"."Unit", "i"."IngredientId"
      FROM (
          SELECT "r"."RecipeId", "r"."Name", "r"."Method"
          FROM "Recipes" AS "r"
          WHERE ("r"."RecipeId" = @__id_0) AND NOT ("r"."IsDeleted")
          LIMIT 2
      ) AS "t"
      LEFT JOIN "Ingredient" AS "i" ON "t"."RecipeId" = "i"."RecipeId"
      ORDER BY "t"."RecipeId", "i"."IngredientId"
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[102]
      Executed handler method OnGetAsync, returned result Microsoft.AspNetCore.Mvc.NotFoundResult.
info: Microsoft.AspNetCore.Mvc.StatusCodeResult[1]
      Executing HttpStatusCodeResult, setting HTTP status code 404
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[4]
      Executed page /Recipes/View in 76.1128ms
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint '/Recipes/View'
```

A single request is made to the URL /Recipe/View/2. Internal ASP.NET Core framework libraries generate logs for the Razor Pages endpoint.

The OnGetAsync handler on the Recipes/View.cshtml Razor Page is executed.

EF Core logs the SQL requests made to the database.

The ActionResult type and the HTTP response code are logged.

Figure 17.1 The ASP.NET Core Framework libraries use logging throughout. A single request generates multiple log messages that describe the flow of the request through your application.

This gives you a lot of useful information. You can see which URL was requested, the Razor Page and page handler that was invoked, the EF Core database command, the action result invoked, and the response. This information can be invaluable when trying to isolate a problem, whether a bug in a production app or a feature in development when you're working locally.

This infrastructure logging can be useful, but log messages that you create yourself can have even greater value. For example, you may be able to spot the cause of the error from the log messages in figure 17.1—we're attempting to view a recipe with an unknown `RecipeId` of `5`, but it's far from obvious. If you explicitly add a log message to your app when this happens, as in figure 17.2, then the problem is much more apparent.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[3]
      Route matched with {page = "/Recipes/View", area = ""}. Executing page /Recipes/View
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[101]
      Executing handler method RecipeApplication.Pages.Recipes.ViewModel.OnGetAsync - ModelState is Valid
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (2ms) [Parameters=[@__id_0='?'], CommandType='Text', CommandTimeout='30']
      SELECT "t"."RecipeId", "t"."Name", "t"."Method", "i"."Name", "i"."Quantity", "i"."Unit", "i"."IngredientId"
      FROM (
          SELECT "r"."RecipeId", "r"."Name", "r"."Method"
          FROM "Recipes" AS "r"
          WHERE ("r"."RecipeId" = @__id_0) AND NOT ("r"."IsDeleted")
          LIMIT 2
      ) AS "t"
      LEFT JOIN "Ingredient" AS "i" ON "t"."RecipeId" = "i"."RecipeId"
      ORDER BY "t"."RecipeId", "i"."IngredientId"
warn: RecipeApplication.Pages.Recipes.ViewModel[12]
      Could not find recipe with id 5
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[102]
      Executed handler method OnGetAsync, returned result Microsoft.AspNetCore.Mvc.NotFoundResult.
info: Microsoft.AspNetCore.Mvc.StatusCodeResult[1]
      Executing HttpStatusCodeResult, setting HTTP status code 404
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[4]
```

You can write your own logs from your app's classes, like this log from the Razor Page ViewModel. These are often more useful for diagnosing issues and tracing the behavior of your app.

Figure 17.2 You can write your own logs. These are often more useful for identifying issues and interesting events in your apps.

This custom log message easily stands out, and clearly states both the problem (the recipe with the requested ID doesn't exist) and the parameters/variables that led to the issue (the ID value of `5`). Adding similar log messages to your own applications will make it easier for you to diagnose problems, track important events, and generally give you an idea of what your app is doing.

Hopefully, you're now motivated to add logging to your apps, so we'll dig into the details of what that involves. In section 17.1.2, you'll see how to create a log message, and how to define where the log messages are written. We'll look in detail at these two aspects in section 17.2 and 17.3; for now, we'll only look at where they fit in terms of the ASP.NET Core logging framework as a whole.

17.1.2 The ASP.NET Core logging abstractions

The ASP.NET Core logging framework consists of a number of logging abstractions (interfaces, implementations, and helper classes), the most important of which are shown in figure 17.3:

- `ILogger`—This is the interface you'll interact with in your code. It has a `Log()` method, which is used to write a log message.
- `ILoggerProvider`—This is used to create a custom instance of an `ILogger`, depending on the provider. A "console" `ILoggerProvider` would create an `ILogger` that writes to the console, whereas a "file" `ILoggerProvider` would create an `ILogger` that writes to a file.
- `ILoggerFactory`—The glue between the `ILoggerProvider` instances and the `ILogger` you use in your code. You register `ILoggerProvider` instances with an `ILoggerFactory` and call `CreateLogger()` on the `ILoggerFactory` when you need an `ILogger`. The factory creates an `ILogger` that wraps each of the providers, so when you call the `Log()` method, the log is written to every provider.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

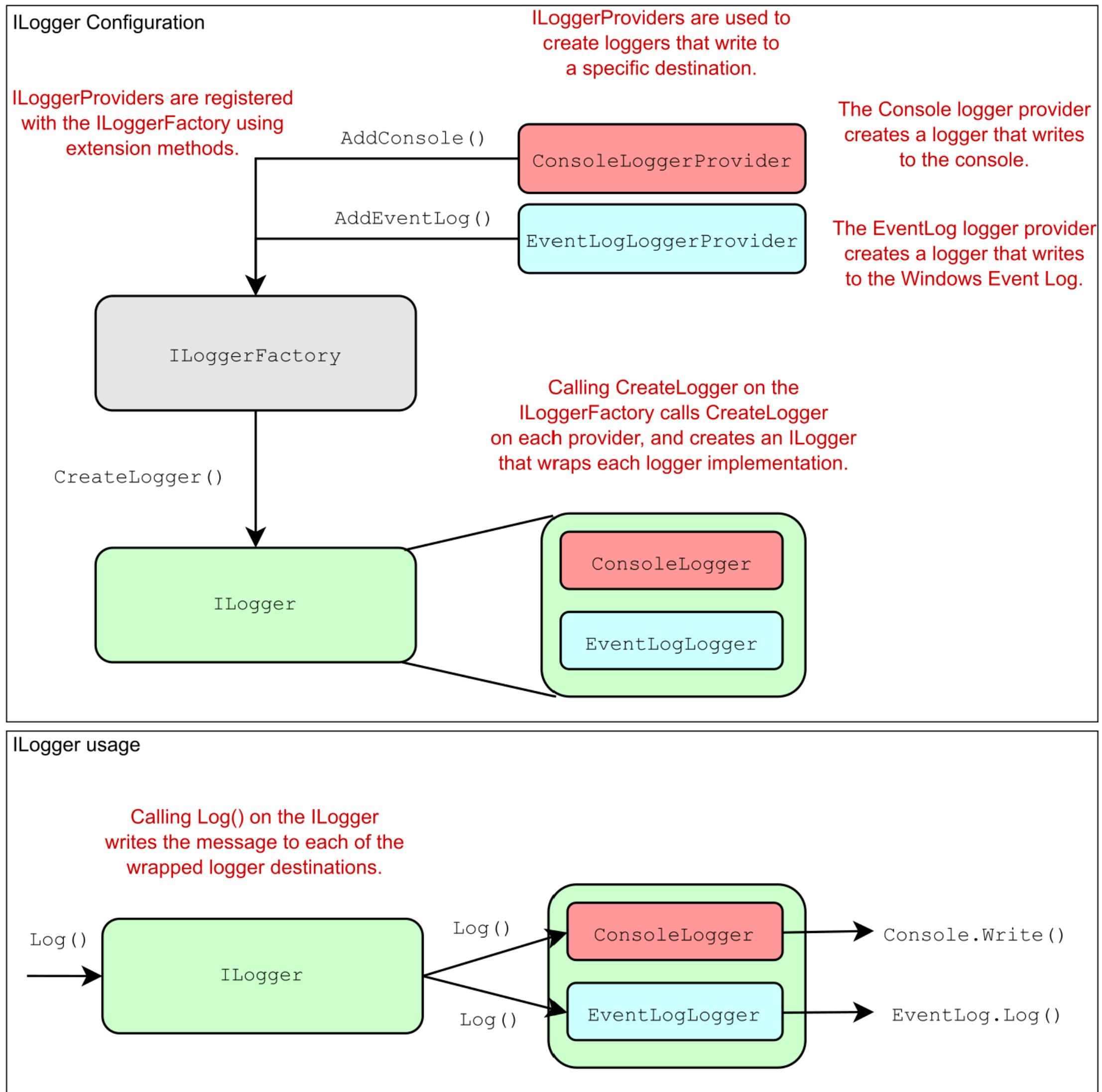


Figure 17.3 The components of the ASP.NET Core logging framework. You register logging providers with an `ILoggerFactory`, which are used to create implementations of `ILogger`. You write logs to the `ILogger`, which uses the `ILogger` implementations to output logs to the console or a file. This design allows you to send logs to multiple locations, without having to configure those locations when you create a log message.

This design in figure 17.3 makes it easy to add or change where your application writes the log messages, without having to change your application code. This listing shows all the code required to add an `ILoggerProvider` that writes logs to the console.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Listing 17.1 Adding a console log provider to `IHost` in Program.cs

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }
    public static IHostBuilder CreateHostBuilder(string[] args) =>
        new HostBuilder()
            .ConfigureLogging(builder =>builder.AddConsole())     #A
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

#A Add new providers with the ConfigureLogging extension method on HostBuilder.

> **NOTE** The console logger is added by default in the `CreateDefaultBuilder` method, as you'll see in section 17.3.

Other than this configuration on `IHostBuilder`, you don't interact with `ILoggerProvider` instances directly. Instead, you write logs using an instance of `ILogger`, as you'll see in the next section.

17.2 Adding log messages to your application

In this section, we'll look in detail at how to create log messages in your own application. You'll learn how to create an instance of `ILogger`, and how to use it to add logging to an existing application. Finally, we'll look at the properties that make up a logging record, what they mean, and what you can use them for.

Logging, like almost everything in ASP.NET Core, is available through DI. To add logging to your own services, you only need to inject an instance of `ILogger<T>`, where `T` is the type of your service.

> **NOTE** When you inject `ILogger<T>`, the DI container indirectly calls `ILoggerFactory.CreateLogger<T>()` to create the wrapped `ILogger` of figure 17.3. In section 17.2.2, you'll see how to work directly with `ILoggerFactory` if you prefer. The `ILogger<T>` interface also implements the non-generic `ILogger` interface but also adds additional convenience methods.

You can use the injected `ILogger` instance to create log messages, which writes the log to each configured `ILoggerProvider`. The following listing shows how to inject an `ILogger<>` instance into the `PageModel` of the Index.cshtml Razor Page for the recipe application from previous chapters and write a log message indicating how many recipes were found.

Listing 17.2 Injecting `ILogger` into a class and writing a log message

```
public class IndexModel : PageModel
```


Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
{
    private readonly RecipeService _service;
    private readonly ILogger<IndexModel> _log;    #A

    public ICollection<RecipeSummaryViewModel> Recipes { get; set; }

    public IndexModel(
        RecipeService service,
        ILogger<IndexModel> log)                  #A
    {
        _service = service;
        _log = log;                               #A
    }

    public void OnGet()
    {
        Recipes = _service.GetRecipes();
        _log.LogInformation(                                    #B
            "Loaded {RecipeCount} recipes", Recipes.Count);     #B
    }
}
```

#A Injects the generic ILogger<T> using DI, which implements ILogger
#B This writes an Information-level log. The RecipeCount variable is substituted in the message.

In this example, you're using one of the many extension methods on `ILogger` to create the log message, `LogInformation()`. There are many extension methods on `ILogger` that let you easily specify a `LogLevel` for the message.

> **DEFINITION** The *log level* of a log is how important it is and is defined by the `LogLevel` enum. Every log message has a log level.

You can also see that the message you pass to the `LogInformation` method has a placeholder indicated by braces, `{RecipeCount}`, and you pass an additional parameter, `Recipes.Count`, to the logger. The logger will replace the placeholder with the parameter at runtime. Placeholders are matched with parameters by position, so if you include two placeholders for example, the second placeholder is matched with the second parameter.

> **TIP** You could have used normal string interpolation to create the log message, for example, `$"Loaded {Recipes.Count} recipes"`. But I recommend always using placeholders, as they provide additional information for the logger that can be used for structured logging, as you'll see in section 17.5.

When the `OnGet` page handler in the `IndexModel` executes, `ILogger` writes a message to any configured logging providers. The exact format of the log message will vary from provider to provider, but figure 17.4 shows how the console provider would display the log message from listing 17.2.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

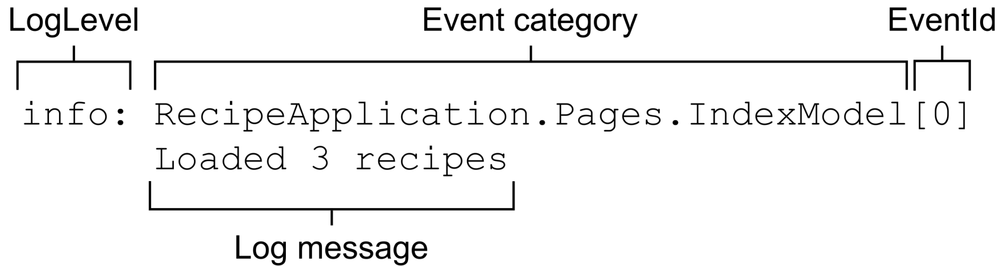


Figure 17.4 An example log message, as it's written to the default console provider. The Log Level category provides information about how important the message is and where it was generated. The EventId provides a way to identify similar log messages.

The exact presentation of the message will vary depending on where the log is written, but each log record includes up to six common elements:

- *Log level*—The log level of the log is how important it is and is defined by the `LogLevel` enum.
- *Event category*—The category may be any string value, but it's typically set to the name of the class creating the log. For `ILogger<T>`, the full name of the type `T` is the category.
- *Message*—This is the content of the log message. It can be a static string, or it can contain placeholders for variables, as shown in listing 17.2. Placeholders are indicated by braces, `{}`, and are substituted with the provided parameter values.
- *Parameters*—If the message contains placeholders, then they're associated with the provided parameters. For the example in listing 17.2, the value of `Recipes.Count` is assigned to the placeholder called `RecipeCount`. Some loggers can extract these values and expose them in your logs, as you'll see in section 17.5.
- *Exception*—If an exception occurs, you can pass the exception object to the logging function along with the message and other parameters. The logger will log the exception, in addition to the message itself.
- *EventId*—This is an optional integer identifier for the error, which can be used to quickly find all similar logs in a series of log messages. You might use an `EventId` of `1000` when a user attempts to load a non-existent recipe, and an `EventId` of `1001` when a user attempts to access a recipe they don't have permission to access. If you don't provide an `EventId`, the value `0` will be used.

Not every log message will have all these elements—you won't always have an `Exception` or parameters for example. There are various overloads to the logging methods that take these as additional method parameters. Besides these, each message will have, at the very least, a level, category, and message. These are the key features of the log, so we'll look at each in turn.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

17.2.1 Log level: how important is the log message?

Whenever you create a log using `ILogger`, you must specify the *log level*. This indicates how serious or important the log message is and is an important factor when it comes to filtering which logs get written by a provider, as well as finding the important log messages after the fact.

You might create an `Information` level log when a user starts to edit a recipe. This is useful for tracing the application's flow and behavior, but it's not important, because everything is normal. But if an exception is thrown when the user attempts to save the recipe, you might create a `Warning` or `Error` level log.

The log level is typically set by using one of several extension methods on the `ILogger` interface, as shown in listing 17.3. This example creates an `Information` level log when the `View` method executes, and a `Warning` level error if the requested recipe isn't found.

Listing 17.3 Specifying the log level using extension methods on `ILogger`

```
private readonly ILogger _log;                               #A
public async IActionResult OnGet(int id)
{
    _log.LogInformation(                                   #B
        "Loading recipe with id {RecipeId}", id);          #B

    Recipe = _service.GetRecipeDetail(id);
    if (Recipe is null)
    {
        _log.LogWarning(                                            #C
            "Could not find recipe with id {RecipeId}", id);    #C
        return NotFound();
    }
    return Page();
}
```

#A An ILogger instance is injected into the controller using constructor injection.
#B Writes an Information level log message
#C Writes a warning level log message

The `LogInformation` and `LogWarning` extension methods create log messages with a log level of `Information` and `Warning`, respectively. There are six log levels to choose from, ordered here from most to least serious:

- `Critical`—For disastrous failures that may leave the app unable to function correctly, such as out of memory exceptions, if the hard drive is out of disk space, or the server is on fire.
- `Error`—For errors and exceptions that you can't handle gracefully, for example, exceptions thrown when saving an edited entity in EF Core. The operation failed, but the app can continue to function for other requests and users.
- `Warning`—For when an unexpected or error condition arises that you can work around. You might log a `Warning` for handled exceptions, or when an entity isn't found, as in listing 17.3.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

- `Information`—For tracking normal application flow, for example, logging when a user logs in, or when they view a specific page in your app. Typically, these log messages provide context when you need to understand the steps leading up to an error message.
- `Debug`—For tracking detailed information that's particularly useful during development. Generally, this only has short-term usefulness.
- `Trace`—For tracking very detailed information, which may contain sensitive information like passwords or keys. It's rarely used, and not used at all by the framework libraries.

Think of these log levels in terms of a pyramid, as shown in figure 17.5. As you progress down the log levels, the importance of the messages goes down, but the frequency goes up. Generally, you'll find many `Debug` level log messages in your application, but (hopefully) few `Critical` or `Error` level messages.

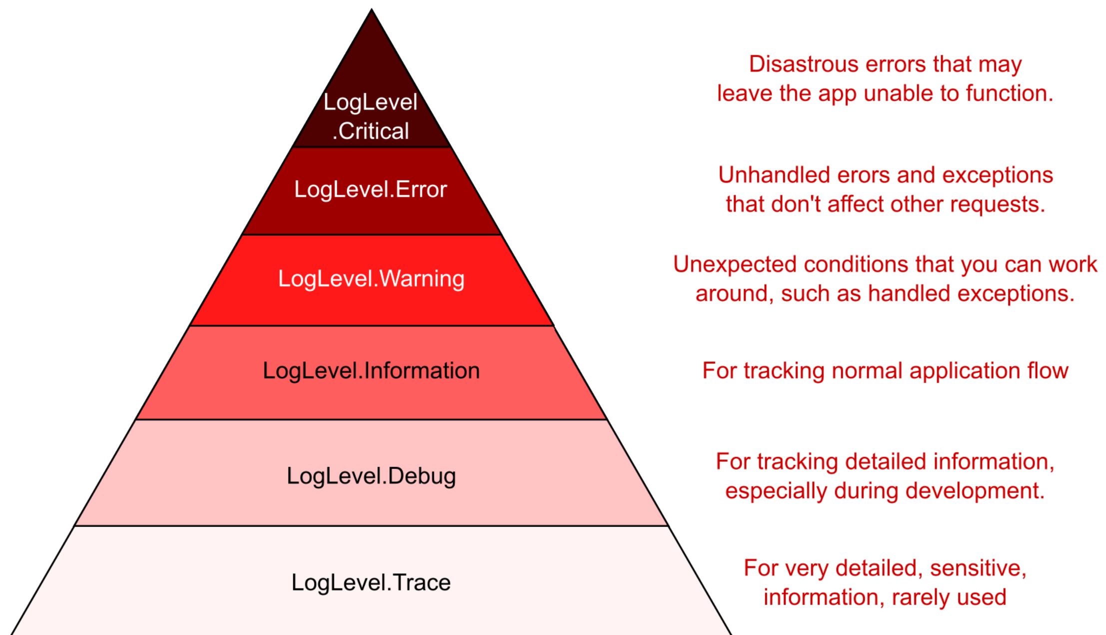


Figure 17.5 The pyramid of log levels. Logs with a level near the base of the pyramid are used more frequently but are less important. Logs with a level near the top should be rare but are important.

This pyramid shape will become more meaningful when we look at filtering in section 17.4. When an app is in production, you typically don't want to record all the `Debug` level messages generated by your application. The sheer volume of messages would be overwhelming to sort through and could end up filling your disk with messages that say, "Everything's OK!"

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Additionally, `Trace` messages shouldn't be enabled in production, as they may leak sensitive data. By filtering out the lower log levels, you can ensure that you generate a sane number of logs in production, while still having access to all the log levels in development.

In general, logs of a higher level are more important than lower-level logs, so a `Warning` log is more important than an `Information` log, but there's another aspect to consider. Where the log came from, or who created the log, is a key piece of information that's recorded with each log message and is called the *category*.

17.2.2 Log category: which component created the log

As well as a log level, every log message also has a *category*. You set the log level independently for every log message, but the category is set when you create the `ILogger` instance. Like log levels, the category is particularly useful for filtering, as you'll see in section 17.4, but it's also written to every log message, as shown in figure 17.6

Every log message has an associated category

The category is typically set to the name of the class creating the log message

```
Command Prompt (light) - dotnet ru
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[3]
      Route matched with {page = "/Recipes/View", area = ""}. Executing page /Recipes/View
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[101]
      Executing handler method RecipeApplication.Pages.Recipes.ViewModel.OnGetAsync - ModelState is Valid
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (2ms) [Parameters=[@__id_0='?'], CommandType='Text', CommandTimeout='30']
      SELECT "t"."RecipeId", "t"."Name", "t"."Method", "i"."Name", "i"."Quantity", "i"."Unit", "i"."IngredientId"
      FROM (
          SELECT "r"."RecipeId", "r"."Name", "r"."Method"
          FROM "Recipes" AS "r"
          WHERE ("r"."RecipeId" = @__id_0) AND NOT ("r"."IsDeleted")
          LIMIT 2
      ) AS "t"
      LEFT JOIN "Ingredient" AS "i" ON "t"."RecipeId" = "i"."RecipeId"
      ORDER BY "t"."RecipeId", "i"."IngredientId"
warn: RecipeApplication.Pages.Recipes.ViewModel[12]
      Could not find recipe with id 5
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[102]
      Executed handler method OnGetAsync, returned result Microsoft.AspNetCore.Mvc.NotFoundResult.
```

Figure 17.6 Every log message has an associated category, which is typically the class name of the component creating the log. The default console logging provider outputs the log category for every log.

The category is a `string`, so you can set it to anything, but the convention is to set it to the fully qualified name of the type that's using `ILogger`. In section 17.2, I achieved this by injecting `ILogger<T>` into `RecipeController`; the generic parameter `T` is used to set it to the category of the `ILogger`.

Alternatively, you can inject `ILoggerFactory` into your methods and pass an explicit category when creating an `ILogger` instance. This lets you change the category to an arbitrary string.

Listing 17.4 Injecting `ILoggerFactory` to use a custom category

```
public class RecipeService
{
    private readonly ILogger _log;
```

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
    public RecipeService(ILoggerFactory factory)                #A
    {
        _log = factory.CreateLogger("RecipeApp.RecipeService");  #B
    }
}
```

#A Injects an ILoggerFactory instead of an ILogger directly
#B Passes a category as a string when calling CreateLogger

There is also an overload of `CreateLogger()` with a generic parameter that uses the provided class to set the category. If the `RecipeService` in listing 17.4 was in the `RecipeApp` namespace, then the `CreateLogger` call could be written equivalently as

```
_log = factory.CreateLogger<RecipeService>();.
```

Similarly, the final `ILogger` instance created by this call would be the same as if you'd directly injected `ILogger<RecipeService>` instead of `ILoggerFactory`.

> **TIP** **Unless you're using heavily customized categories for some reason, favor injecting `ILogger<T>` into your methods over `ILoggerFactory`.**

The final compulsory part of every log entry is fairly obvious: the *log message*. At the simplest level, this can be any string, but it's worth thinking carefully about what information would be useful to record—anything that will help you diagnose issues later on!

17.2.3 Formatting messages and capturing parameter values

Whenever you create a log entry, you must provide a *message*. This can be any string you like, but as you saw in listing 17.2, you can also include placeholders indicated by braces, `{}`, in the message string:

```
_log.LogInformation("Loaded {RecipeCount} recipes", Recipes.Count);
```

Including a placeholder and a parameter value in your log message effectively creates a key-value pair, which some logging providers can store as additional information associated with the log. The previous log message would assign the value of `Recipes.Count` to a key, `RecipeCount`, and the log message itself is generated by replacing the placeholder with the parameter value, to give (if `Recipes.Count=3`):

```
"Loaded 3 recipes"
```

You can include multiple placeholders in a log message, and they'll be associated with the additional parameters passed to the log method. The order of the placeholders in the format string must match the order of the parameters you provide.

> **WARNING** **You must pass at least as many parameters to the log method as there are placeholders in the message. If you don't pass enough parameters, you'll get an exception at runtime.**

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

For example, the log message

```
_log.LogInformation("User {UserId} loaded recipe {RecipeId}", 123, 456)
```

would create the parameters `UserId=123` and `RecipeId=456`. *Structured logging* providers could store these values, in addition to the formatted log message `"User 123 loaded recipe 456"`. This makes it easier to search the logs for a particular `UserId` or `RecipeId`.

> **DEFINITION** *Structured or semantic logging* attaches additional structure to log messages to make them more easily searchable and filterable. Rather than storing only text, it stores additional contextual information, typically as key-value pairs. JSON is a comment format used for structured log messages, as it has all of these properties.

Not all logging providers use semantic logging. The default console logging provider doesn't, for example—the message is formatted to replace the placeholders, but there's no way of searching the console by key-value.

But even if you're not using structured logging initially, I recommend writing your log messages as though you are, with explicit placeholders and parameters. That way, if you decide to add a structured logging provider later, then you'll immediately see the benefits. Additionally, I find that thinking about the parameters that you can log in this way prompts you to record more parameter values, instead of only a log message. There's nothing more frustrating than seeing a message like `"Cannot insert record due to duplicate key"` but not having the key value logged!

> **TIP** Generally speaking, I'm a fan of C# 6's interpolated strings, but don't use them for your log messages when a placeholder and parameter would also make sense. Using placeholders instead of interpolated strings will give you the same output message but will also create key-value pairs that can be searched later.

We've looked a lot at how you can create log messages in your app, but we haven't focused on where those logs are written. In the next section, we'll look at the built-in ASP.NET Core logging providers, how they're configured, and how you can replace the defaults with a third-party provider.

17.3 Controlling where logs are written using logging providers

In this section you'll learn how to control where your log messages are written by adding additional `ILoggerProvider`s to your application. As an example, you'll see how to add a simple file logger provider that writes your log messages to a file, in addition to the existing console logger provider. In section 17.3.2 you'll learn how to swap out the default logging infrastructure entirely for an alternative implementation using the open source Serilog library.

Up to this point, we've been writing all our log messages to the console. If you've run any ASP.NET Core sample apps locally, you'll have probably already seen the log messages written to the console window.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

NOTE If you're using Visual Studio and debugging using the IIS Express option (the default), then you won't see the console window (though the log messages are written to the Debug Output window instead). For that reason, I normally ensure I select the app name from the dropdown in the debug toolbar, instead of IIS Express.

Writing log messages to the console is great when you're debugging, but it's not much use for production. No one's going to be monitoring a console window on a server, and the logs wouldn't be saved anywhere or be searchable. Clearly, you'll need to write your production logs somewhere else when in production.

As you saw in section 17.1, *logging providers* control the destination of your log messages in ASP.NET Core. They take the messages you create using the `ILogger` interface and write them to an output location, which varies depending on the provider.

NOTE This name always gets to me—the log *provider* effectively *consumes* the log messages you create and outputs them to a destination. You can probably see the origin of the name from figure 17.3, but I still find it somewhat counterintuitive!

Microsoft have written several first-party log providers for ASP.NET Core, which are available out-of-the-box in ASP.NET Core. These include

- *Console provider*—Writes messages to the console, as you've already seen.
- *Debug provider*—Writes messages to the debug window when you're debugging an app in Visual Studio or Visual Studio Code, for example.
- *EventLog provider*—Writes messages to the Windows Event Log. Only outputs log messages when running on Windows, as it requires Windows-specific APIs.
- *EventSource provider*—Writes messages using Event Tracing for Windows (ETW) or LTTng tracing on Linux.

There are also many third-party logging provider implementations, such as an Azure App Service provider, an elmah.io provider, and an ElasticSearch provider. On top of that, there are integrations with other pre-existing logging frameworks like NLog and Serilog. It's always worth looking to see whether your favorite .NET logging library or service has a provider for ASP.NET Core, as most do.

You configure the logging providers for your app in Program.cs using `HostBuilder`. The `CreateDefaultBuilder` helper method configures the console and debug providers for your application automatically, but it's likely you'll want to change or add to these.

You have two options when you need to customize logging for your app:

- Use your own `HostBuilder` instance, instead of `Host.CreateDefaultBuilder` and configure it explicitly.
- Add an additional `ConfigureLogging` call after `CreateDefaultBuilder`.

If you only need to customize logging, then the latter approach is simpler. But if you find you also want to customize the other aspects of the `HostBuilder` created by

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

`CreateDefaultBuilder` (such as your app configuration settings), then it may be worth ditching the `CreateDefaultBuilder` method and creating your own instance instead.

In section 17.3.1, I show how to add a simple third-party logging provider to your application that writes log messages to a file, so that your logs are persisted. In section 17.3.2, I'll show how to go one step further and replace the default `ILoggerFactory` in ASP.NET Core with an alternative implementation using the popular open source Serilog library.

17.3.1 Adding a new logging provider to your application

In this section, we're going to add a logging provider that writes to a rolling file, so our application writes logs to a new file each day. We'll continue to log using the console and debug providers as well, because they will be more useful than the file provider when developing locally.

To add a third-party logging provider in ASP.NET Core, you must

1. Add the logging provider NuGet package to the solution. I'm going to be using a provider called NetEscapades.Extensions.Logging.RollingFile, which is available on NuGet and GitHub. You can add it to your solution using the NuGet Package Manager in Visual Studio, or using the .NET CLI by running

   ```
   dotnet add package NetEscapades.Extensions.Logging.RollingFile
   ```

   from your application's project folder.

   NOTE This package is a simple file logging provider, available at https://github.com/andrewlock/NetEscapades.Extensions.Logging which is based on the Azure App Service logging provider. If you would like more control over your logs, such as the file format, consider using Serilog instead, as described in section 17.3.2.

2. Add the logging provider using the `IHostBuilder.ConfigureLogging()` extension method. You can add the file provider by calling `AddFile()`, as shown in the next listing. This is an extension method, provided by the logging provider package, to simplify adding the provider to your app.

Listing 17.5 Adding a third-party logging provider to `IHostBuilder`

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)                        #A
            .ConfigureLogging(builder => builder.AddFile())    #B
            .ConfigureWebHostDefaults(webBuilder =>
```


Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

#A The CreateDefaultBuilder method configures the console and debug providers as normal.
#B Adds the new file logging provider to the logger factory.

> **NOTE** Adding a new provider doesn't replace existing providers. Listing 17.5 uses the `CreateDefaultBuilder` helper method, so the console and debug logging providers have already been added. To remove them, call `builder.ClearProviders()` at the start of the `ConfigureLogging` method, or use a custom `HostBuilder`.

With the file logging provider configured, you can run the application and generate logs. Every time your application writes a log using an `ILogger` instance, `ILogger` writes the message to all configured providers, as shown in figure 17.7. The console messages are conveniently available, but you also have a persistent record of the logs stored in a file.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

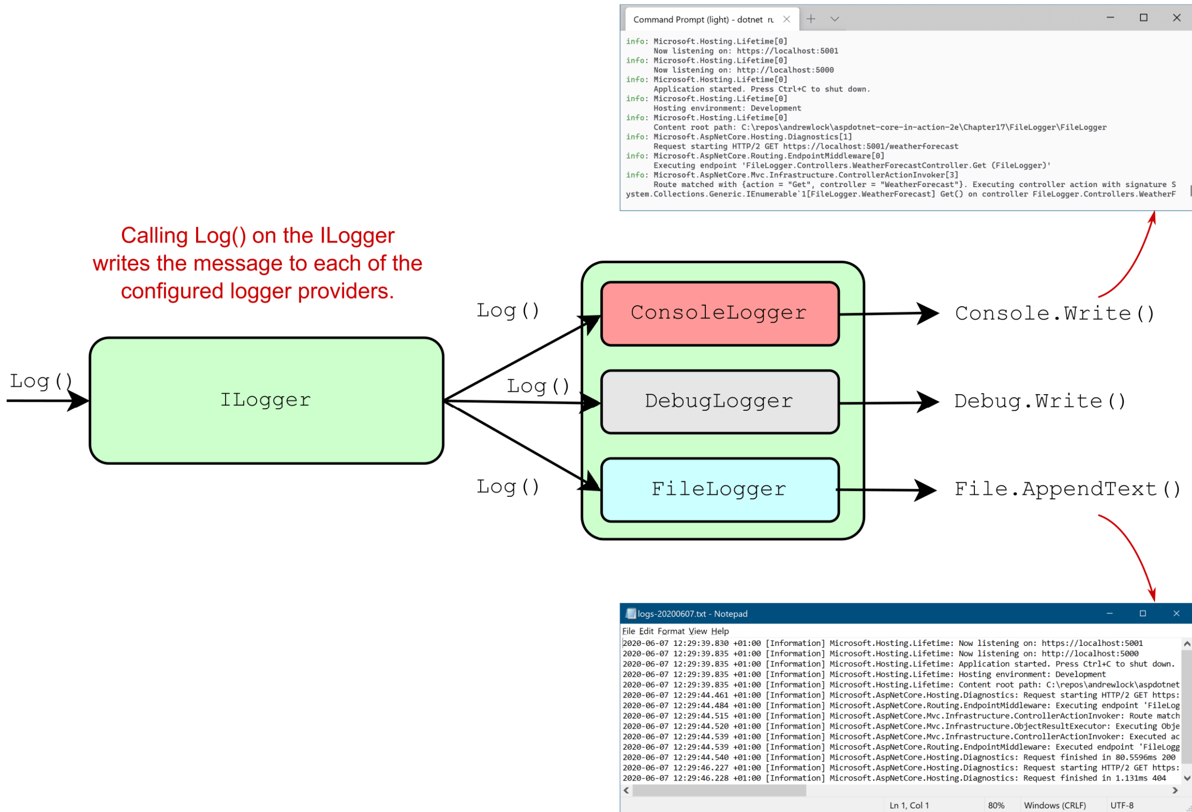


Figure 17.7 Logging a message with `ILogger` writes the log using all of the configured providers. This lets you, for example, log a convenient message to the console while also persisting the logs to a file.

> **TIP** By default, the rolling file provider will write logs to a subdirectory of your application. You can specify additional options such as filenames and file size limits using overloads of `AddFile()`. For production, I recommend using a more established logging provider, such as Serilog.

The key takeaway from this listing is that the provider system makes it easy to integrate existing logging frameworks and providers with the ASP.NET Core logging abstractions. Whichever logging provider you choose to use in your application, the principles are the same: call `ConfigureLogging` on `IHostBuilder` and add a new logging provider using extension methods like `AddConsole()`, or `AddFile()` in this case.

Logging your application messages to a file can be useful in some scenarios, and it's certainly better than logging to a non-existent console window in production, but it may still not be the best option.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

If you discovered a bug in production and you needed to quickly look at the logs to see what happened, for example, you'd need to log on to the remote server, find the log files on disk, and trawl through them to find the problem. If you have multiple web servers, then suddenly you'd have a mammoth job to fetch all the logs before you could even start to tackle the bug. Not fun. Add to that the possibility of file permission or drive space issues and file logging seems less attractive.

Instead, it's often better to send your logs to a centralized location, separate from your application. Exactly where this location may be is up to you; the key is that each instance of your app sends its logs to the same location, separate from the app itself.

If you're running your app on Azure, then you get centralized logging for free because you can collect logs using the Azure App Service provider. Alternatively, you could send your logs to a third-party log aggregator service such as Loggr (http://loggr.net/), elmah.io (https://elmah.io/), or Seq (https://getseq.net/). You can find ASP.NET Core logging providers for each of these services on NuGet, so adding them is the same process as adding the file provider you've seen already.

Another popular option is to use the open source Serilog library to simultaneously write to a variety of different locations. In the next section, I'll show how you can replace the default `ILoggerFactory` implementation with Serilog in your application, opening up a wide range of possible options for where your logs are written.

17.3.2 Replacing the default ILoggerFactory with Serilog

In this section, we'll replace the default `ILoggerFactory` in the Recipe app with an implementation that uses Serilog. Serilog (https://serilog.net) is an open source project that can write logs to many different locations, such as files, the console, an Elasticsearch cluster,[76] or a database. This is similar to the functionality you get with the default `ILoggerFactory`, but due to the maturity of Serilog, you may find you can write to more places.

Serilog predates ASP.NET Core, but thanks to the logging abstractions around `ILoggerFactory` and `ILoggerProvider`, you can easily integrate with Serilog while still using the `ILogger` abstractions in your application code.

Serilog uses a similar design philosophy to the ASP.NET Core logging abstractions—you write logs to a central logging object and the log messages are written to multiple locations, such as the console or a file. Serilog calls each of these locations a *sink*.[77]

When you use Serilog with ASP.NET Core, you'll typically replace the default `ILoggerFactory` with a custom factory that contains a single logging provider, `SerilogLoggerProvider`. This provider can write to multiple locations, as shown in figure

---

[76]Elasticsearch is a REST-based search engine that's often used for aggregating logs. You can find out more at www.elastic.co/elasticsearch/.
[77]For a full list of available sinks, see https://github.com/serilog/serilog/wiki/Provided-Sinks. There are 93 different sinks at the time of writing!

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

17.8. This configuration is a bit of a departure from the standard ASP.NET Core logging setup, but it prevents Serilog's features from conflicting with equivalent features of the default `LoggerFactory`, such as filtering (see section 17.4).

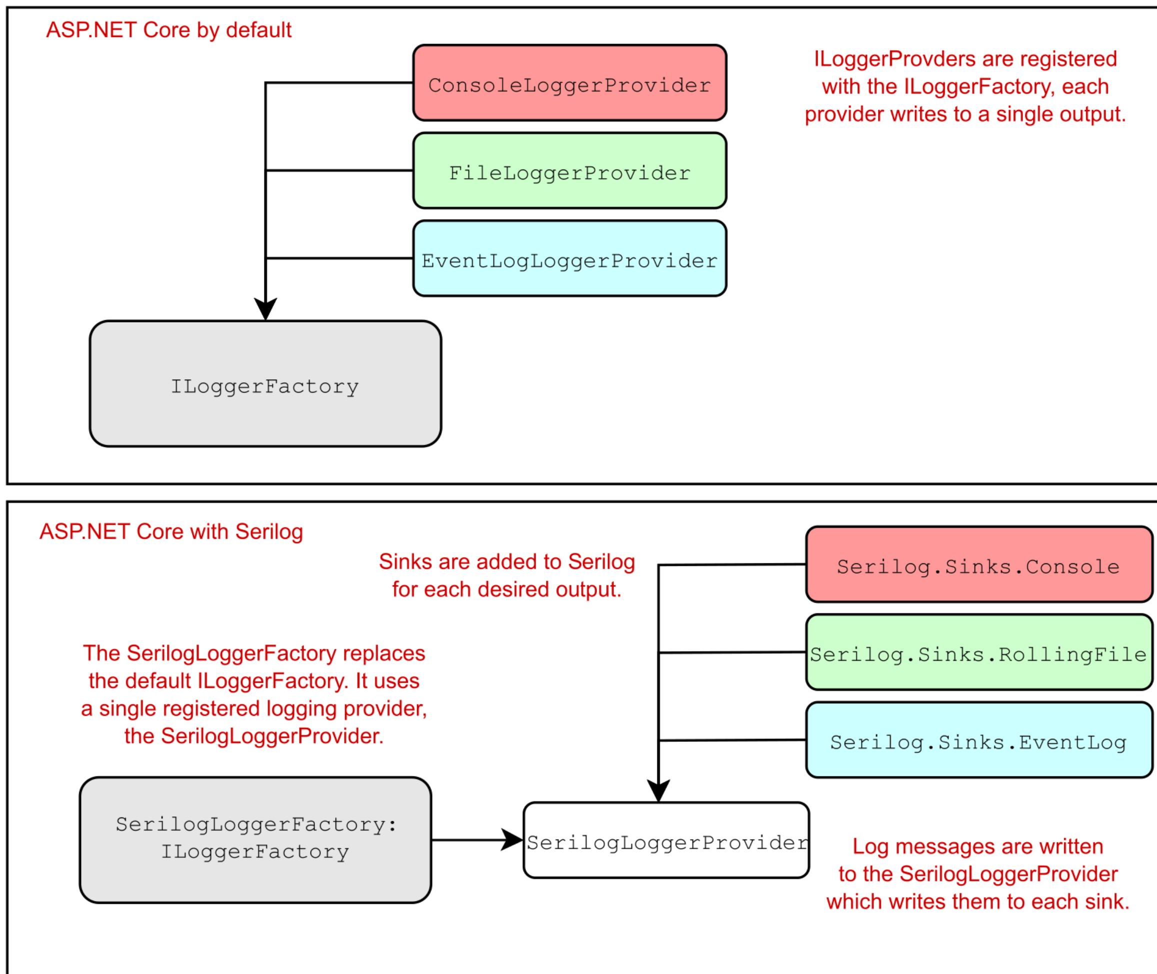


Figure 17.8 Configuration differences when using Serilog with ASP.NET Core compared to the default logging configuration. You can achieve the same functionality with both approaches, but you may find Serilog provides additional libraries for adding extra features.

> **TIP** If you're familiar with Serilog, you can use the examples in this section to easily integrate a working Serilog configuration with the ASP.NET Core logging infrastructure.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

In this section, we'll add a single sink to write the log messages to the console, but using the Serilog logging provider instead of the built-in console provider. Once you've set this up, adding additional sinks to write to other locations is trivial. Adding the Serilog logging provider to an application involves three steps:

1. Add the required Serilog NuGet packages to the solution.
2. Create a Serilog logger and configure it with the required sinks.
3. Call `UseSerilog()` on `IHostBuilder` to replace the default `ILoggerFactory` implementation with `SerilogLoggerFactory`. This configures the Serilog provider automatically and hooks up the already-configured Serilog logger.

To install Serilog into your ASP.NET Core app, you need to add the base NuGet package and the NuGet packages for any sinks you need. You can do this through the Visual Studio NuGet GUI, using the PMC, or using the .NET CLI. To add the Serilog ASP.NET Core package and a sink for writing to the console, run

```
dotnet add package Serilog.AspNetCore
dotnet add package Serilog.Sinks.Console
```

This adds the necessary NuGet packages to your project file and restores them. Next, create a Serilog logger and configure it to write to the console by adding the console sink, as shown in listing 17.6. This is the most basic of Serilog configurations, but you can add extra sinks and configuration here too.[78] I've also added a `try-catch-finally` block around our call to `CreateHostBuilder`, to ensure that logs are still written if there's an error starting up the web host or there's a fatal exception. Finally, the Serilog logger factory is configured by calling `UseSerilog()` on the `IHostBuilder`.

Listing 17.6 Configuring a Serilog logging provider to use a console sink

```
public class Program
{
    public static void Main(string[] args)
    {
        Log.Logger = new LoggerConfiguration()     #A
            .WriteTo.Console()                     #B
            .CreateLogger();                       #C
        try
        {
            CreateHostBuilder(args).Build().Run();
        }
        catch (Exception ex)
        {
            Log.Fatal(ex, "Host terminated unexpectedly");
        }
        finally
```

[78] You can customize Serilog until the cows come home, so it's worth consulting the documentation to see what's possible. The wiki is particularly useful: https://github.com/serilog/serilog/wiki/Configuration-Basics.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
        {
            Log.CloseAndFlush();
        }
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .UseSerilog()                          #D
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

#A Creates a LoggerConfiguration for configuring the Serilog logger.
#B Serilog will write logs to the console.
#C This creates a Serilog logger instance on the static Log.Logger property.
#D Registers the SerilogLoggerFactory and connects the Log.Logger as the sole logging provider.

With the Serilog logging provider configured, you can run the application and generate some logs. Every time the app generates a log, `ILogger` writes the message to the Serilog provider, which writes it to every sink. In the example, you've only configured the console sink, so the output will look something like figure 17.9. The Serilog console sink colorizes the logs more than the built-in console provider, so I find it's somewhat easier to parse the logs visually.

Figure 17.9 Example output using the Serilog provider and a console sink. The output has more colorization than the built-in console provider, though by default it doesn't display the log category for each log.[79]

> **TIP** Serilog has many great features in addition to this. One of my favorites is the ability to add *enrichers*. These automatically add additional information to all your log messages, such as the process ID or environment variables, which can be useful when diagnosing problems. For an in-depth look at the

[79] If you wish to display the log category in the console sink, you can customize the `outputTemplate` and add `{SourceContext}`. For details, see https://github.com/serilog/serilog-sinks-console#output-templates.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

recommended way to configure Serilog for ASP.NET Core apps, see this post by Nicholas Blumhardt, the creator of Serilog: https://nblumhardt.com/2019/10/serilog-in-aspnetcore-3/.

Serilog lets you easily plug in additional sinks to your application, in much the same way as you do with the default ASP.NET Core abstractions. Whether you choose to use Serilog or stick to other providers is up to you; the feature sets are quite similar, though Serilog is more mature. Whichever you choose, once you start running your apps in production, you'll quickly discover a different issue: the sheer number of log messages your app generates!

17.4 Changing log verbosity with filtering

In this section you'll see how to reduce the number of log messages written to the logger providers. You'll learn how to apply a base level filter, filter out messages from specific namespaces, and use logging provider-specific filters.

If you've been playing around with the logging samples, then you'll probably have noticed that you get a lot of log messages, even for a single request like the one in figure 17.2: messages from the Kestrel server, messages from EF Core, not to mention your own custom messages. When you're debugging locally, having access to all that detailed information is extremely useful, but in production you'll be so swamped by noise that it'll make picking out the important messages difficult.

ASP.NET Core includes the ability to filter out log messages *before* they're written based on a combination of three things:

- The log level of the message
- The category of the logger (who created the log)
- The logger provider (where the log will be written)

You can create multiple rules using these properties, and for each log that's created, the most specific rule will be applied to determine whether the log should be written to the output. You could create the following three rules:

- *The default minimum log level is* `Information`. If no other rules apply, only logs with a log level of `Information` or above will be written to providers.
- *For categories that start with* `Microsoft`, *the minimum log level is* `Warning`. Any logger created in a namespace that starts with `Microsoft` will only write logs that have a log level of `Warning` or above. This would filter out the "noisy" framework messages you saw in figure 17.6.
- *For the console provider, the minimum log level is* `Error`. Logs written to the console provider must have a minimum log level of `Error`. Logs with a lower level won't be written to the console, though they might be written using other providers.

Typically, the goal with log filtering is to reduce the number of logs written to certain providers, or from certain namespaces (based on the log category). Figure 17.10 shows a possible set of filtering rules that apply filtering rules to the console and file logging providers.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

In this example, the console logger explicitly restricts logs written in the `Microsoft` namespace to `Warning` or above, so the console logger ignores the log message shown. Conversely, the file logger doesn't have a rule that explicitly restricts the `Microsoft` namespace, so it uses the configured minimum level of `Information` and writes the log to the output.

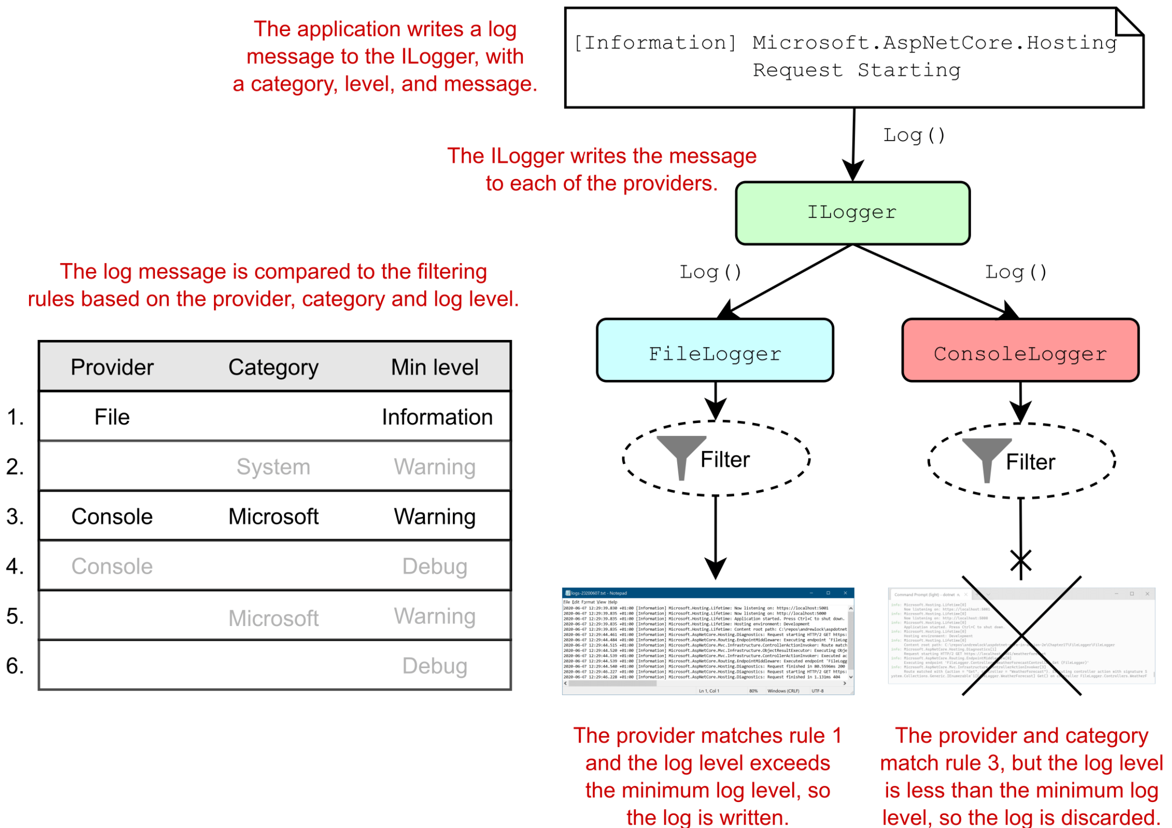


Figure 17.10 Applying filtering rules to a log message to determine whether a log should be written. For each provider, the most specific rule is selected. If the log exceeds the rule's required minimum level, the provider writes the log, otherwise it discards it.

> **TIP** Only a single rule is chosen when deciding whether a log message should be written; they aren't combined. In figure 17.10, rule 1 is considered more specific than rule 5, so the log is written to the file provider, even though both could technically apply.

You typically define your app's set of logging rules using the layered configuration approach discussed in chapter 11, because this lets you easily have different rules when running in

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

development and production. You do this by calling `AddConfiguration` when configuring logging in Program.cs, but `CreateDefaultBuilder()` also does this for you automatically.

This listing shows how you'd add configuration rules to your application when configuring your own `HostBuilder`, instead of using the `CreateDefaultBuilder` helper method.

Listing 17.7 Loading logging configuration in `ConfigureLogging`

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        new HostBuilder()
            .UseContentRoot(Directory.GetCurrentDirectory())
            .ConfigureAppConfiguration(config =>               #A
                config.AddJsonFile("appsettings.json"))        #A
            .ConfigureLogging((ctx, builder) =>
            {
                builder.AddConfiguration(                      #B
                    ctx.Configuration.GetSection("Logging"));  #B
                builder.AddConsole();                          #C
            })
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

#A Loads configuration values from appsettings.json
#B Loads the log filtering configuration from the Logging section and adds to ILoggingBuilder
#C Adds a console provider to the app

In this example, I've loaded the configuration from a single file, appsettings.json, which contains all of our app configuration. The logging configuration is specifically contained in the `"Logging"` section of the `IConfiguration` object, which is available when you call `Configurelogging()`.

> **TIP** As you saw in chapter 11, you can load configuration settings from multiple sources, like JSON files and environment variables, and can load them conditionally based on the `IHostingEnvironment`. A common practice is to include logging settings for your production environment in appsettings.json, and overrides for your local development environment in appsettings.Development.json.

The logging section of your configuration should look similar to the following listing, which shows how you could define the rules shown in figure 17.10.

Listing 17.8 The log filtering configuration section of appsettings.json

```
{
  "Logging": {
```


Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
    "LogLevel": {               #A
      "Default": "Debug",       #A
      "System": "Warning",      #A
      "Microsoft": "Warning"    #A
    },
    "File": {                      #B
      "LogLevel": {                #B
        "Default": "Information"   #B
      }
    },
    "Console": {                  #C
      "LogLevel": {               #C
        "Default": "Debug",       #C
        "Microsoft": "Warning"    #C
      }
    }
  }
}
```

#A Rules to apply if there are no applicable rules for a provider
#B Rules to apply to the File provider
#C Rules to apply to the Console provider

When creating your logging rules, the important thing to bear in mind is that if you have *any* provider-specific rules, these will take precedence over the category-based rules defined in the `"LogLevel"` section. Therefore, for the configuration defined in listing 17.8, if your app *only* uses the file or console logging providers, then the rules in the `"LogLevel"` section will effectively never apply.

If you find this confusing, don't worry, so do I. Whenever I'm setting up logging, I check the algorithm used to determine which rule will apply for a given provider and category, which is as follows:

1. Select all rules for the given provider. If no rules apply, select all rules that don't define a provider (the top `"LogLevel"` section from listing 17.8).
2. From the selected rules, select rules with the longest matching category prefix. If no selected rules match the category prefix, select the `"Default"` if present.
3. If multiple rules are selected, use the last one.
4. If no rules are selected, use the global minimum level, `"LogLevel:Default"` (`Debug` in listing 17.8).

Each of these steps (except the last) narrows down the applicable rules for a log message, until you're left with a single rule. You saw this in effect for a `"Microsoft"` category log in figure 17.10. Figure 17.11 shows the process in more detail.

> **WARNING** Log filtering rules aren't merged together; a single rule is selected. Including provider-specific rules will override global category-specific rules, so I tend to stick to category-specific rules in my apps to make the overall set of rules easier to understand.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

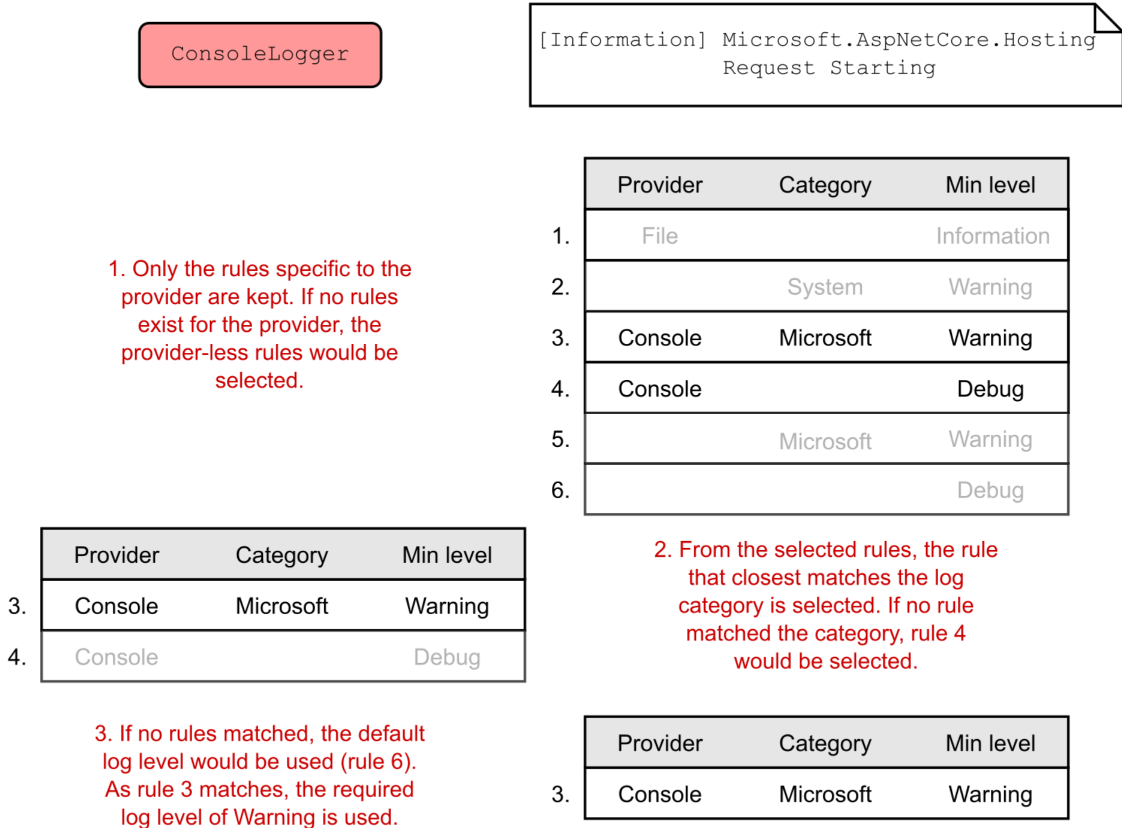


Figure 17.11 Selecting a rule to apply from the available set for the console provider and an `Information` level log. Each step reduces the number of rules that apply until you're left with only one.

With some effective filtering in place, your production logs should be much more manageable, as shown in figure 17.12. Generally, I find it's best to limit the logs from the ASP.NET Core infrastructure and referenced libraries to `Warning` or above, while keeping logs that my app writes to `Debug` in development and `Information` in production.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Figure 17.12 Using filtering to reduce the number of logs written. In this example, category filters have been added to the Microsoft and System namespaces, so only logs of Warning and above are recorded. That increases the number of logs that are directly relevant to your application.

This is close to the default configuration used in the ASP.NET Core templates. You may find you need to add additional category-specific filters, depending on which NuGet libraries you use and the categories they write to. The best way to find out is generally to run your app and see if you get flooded with uninteresting log messages!

Even with your log verbosity under your control, if you stick to the default logging providers like the file or console loggers, then you'll probably regret it in the long run. These log providers work perfectly well, but when it comes to finding specific error messages, or analyzing your logs, you'll have your work cut out for you. In the next section, you'll see how structured logging can help tackle this problem.

17.5 Structured logging: creating searchable, useful logs

In this section you'll learn how structured logging makes working with log messages easier. You'll learn to attach key-value pairs to log messages, and how to store and query for key values using the structured logging provider, Seq. Finally, you'll learn how to use scopes to attach key value pairs to all log messages within a block.

Let's imagine you've rolled out the recipe application we've been working on into production. You've added logging to the app so that you can keep track of any errors in your application, and you're storing the logs in a file.

One day, a customer calls and says they can't view their recipe. Sure enough, when you look through the log messages you a see a warning:

```
warn: RecipeApplication.Controllers.RecipeController[12]
      Could not find recipe with id 3245
```

This piques your interest—why did this happen? Has it happened before for this *customer*? Has it happened before for this *recipe*? Has it happened for *other* recipes? Does it happen regularly?

How would you go about answering these questions? Given that the logs are stored in a text file, you might start doing basic text searches in your editor of choice, looking for the phrase `"Could not find recipe with id"`. Depending on your notepad-fu skills, you could

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

probably get a fair way in answering your questions, but it would likely be a laborious, error-prone, and painful process.

The limiting factor is that the logs are stored as *unstructured* text, so text processing is the only option available to you. A better approach is to store the logs in a *structured* format, so that you can easily query the logs, filter them, and create analytics. Structured logs could be stored in any format, but these days they're typically represented as JSON. For example, a structured version of the same recipe warning log might look something like

```
{
  "eventLevel": "Warning",
  "cateogry": "RecipeApplication.Controllers.RecipeController",
  "eventId": "12",
  "messageTemplate": "Could not find recipe with {recipeId}",
  "message": "Could not find recipe with id 3245",
  "recipeId": "3245"
}
```

This structured log message contains all the same details as the unstructured version, but in a format that would easily let you search for specific log entries. It makes it simple to filter logs by their `EventLevel`, or to only show those logs relating to a specific recipe ID.

> **NOTE** This is only an example of what a structured log could look like. The format used for the logs will vary depending on the logging provider used and could be anything. The key point is that properties of the log are available as key-value pairs.

Adding structured logging to your app requires a logging provider that can create and store structured logs. Elasticsearch is a popular general search and analytics engine that can be used to store and query your logs. Serilog includes a sink for writing logs to Elasticsearch that you can add to your app in the same way as you added the console sink in section 17.3.2.

> **TIP** If you want to learn more about Elasticsearch, *Relevant Search* by Doug Turnbull and John Berryman (Manning, 2016) is a great choice, and will show how you can make the most of your structured logs.

Elasticsearch is a powerful production-scale engine for storing your logs, but setting it up isn't for the faint of heart. Even after you've got it up and running, there's a somewhat steep learning curve associated with the query syntax. If you're interested in something more user-friendly for your structured logging needs, then Seq (https://getseq.net) is a great option. In the next section, I'll show how adding Seq as a structured logging provider makes analyzing your logs that much easier.

17.5.1 Adding a structured logging provider to your app

To demonstrate the advantages of structured logging, in this section, you'll configure an app to write logs to Seq. You'll see that configuration is essentially identical to unstructured providers, but that the possibilities afforded by structured logging make considering it a no-brainer.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Seq is installed on a server or your local machine and collects structured log messages over HTTP, providing a web interface for you to view and analyze your logs. It is currently available as a Windows app or a Linux Docker container. You can install a free version for development,[80] which will allow you to experiment with structured logging in general.

From the point of view of your app, the process for adding the Seq provider should be familiar:

1. Install the Seq logging provider using Visual Studio or the .NET CLI with

```
dotnet add package Seq.Extensions.Logging
```

2. Add the Seq logging provider in Program.cs inside the `ConfigureLogging` method. To add the Seq provider in addition to the console and debug providers included as part of `CreateDefaultBuilder`, use

```
Host.CreateDefaultBuilder(args)
    .ConfigureLogging(builder => builder.AddSeq())
    .ConfigureWebHostDefaults(webBuilder =>
    {
        webBuilder.UseStartup<Startup>();
    });
```

That's all you need to add Seq to your app. This will send logs to the default local URL when you have Seq installed in your local environment. The `AddSeq()` extension method includes additional overloads to customize Seq when you move to production, but this is all you need to start experimenting locally.

If you haven't already, install Seq on your development machine and navigate to the Seq app at http://localhost:5341. In a different tab, open up your app and start browsing around and generating logs. Heading back to Seq, if you refresh the page, you'll see a list of logs, something like in figure 17.13. Clicking on a log expands it and shows you the structured data recorded for the log.

[80] You can download the Windows installer for Seq from https://getseq.net/Download.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Logs are listed in reverse chronological order

Refresh the list of logs or turn on auto-refresh

Clicking on a log reveals the structured data

Each log includes various key-value pairs, as well as the standard level and message properties.

Warning level logs are highlighted in yellow

Figure 17.13 The Seq UI. Logs are presented as a list. You can view the structured logging details of individual logs, view analytics for logs in aggregates, and search by log properties.

ASP.NET Core supports structured logging by treating each captured parameter from your message format string as a key-value-pair. If you create a log message using the format string

```
_log.LogInformation("Loaded {RecipeCount} recipes", Recipes.Count);
```

then the logging provider will create a `RecipeCount` parameter with a value of `Recipes.Count`. These parameters are added as properties to each structured log, as you can see in figure 17.13.

Structured logs are generally easier to read than your standard-issue console output, but their real power comes when you need to answer a specific question. Consider the problem from before, where you see the error

```
Could not find recipe with id 3245
```

and you want to get a feel for how widespread the problem is. The first step would be to identify how many times this error has occurred, and to see whether it's happened to any

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

other recipes. Seq lets you filter your logs, but it also lets you craft SQL queries to analyze your data, so finding the answer to the question takes a matter of seconds, as shown in figure 17.14.

> **NOTE** You don't need query languages like SQL for simple queries, but it makes digging into the data easier. Other structured logging providers may provide query languages other than SQL, but the principal is the same as this Seq example.

Figure 17.14 Querying logs in Seq. Structured logging makes log analysis like this example easy.

A quick search shows that you've recorded the log message with `EventId.Id=12` (the `EventId` of the warning we're interested in) 13 times, and every time the offending `RecipeId` was 3245. This suggests that there may be something wrong with that recipe in particular, which points you in the right direction to find the problem.

More often than not, figuring out errors in production involves logging detective work like this to isolate where the problem occurred. Structured logging makes this process significantly easier, so it's well worth considering, whether you choose Seq, Elasticsearch, or a different provider.

I've already described how you can add structured properties to your log messages using variables and parameters from the message, but as you can see in figure 17.13, there are far more properties visible than exist in the message alone.

Scopes provide a way to add arbitrary data to your log messages. They're available in some unstructured logging providers, but they shine when used with structured logging providers. In the final section of this chapter, I'll demonstrate how you can use them to add additional data to your log messages.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

17.5.2 Using scopes to add additional properties to your logs

You'll often find in your apps that you have a group of operations that all use the same data, which would be useful to attach to logs. For example, you might have a series of database operations that all use the same transaction ID, or you might be performing multiple operations with the same user ID or recipe ID. *Logging scopes* provide a way of associating the same data to every log message in such a group.

> **DEFINITION** *Logging scopes* are used to group multiple operations by adding the same data to each log message.

Logging scopes in ASP.NET Core are created by calling `ILogger.BeginScope<T>(T state)` and providing the `state` data to be logged. You create scopes inside a `using` block; any log messages written inside the scope block will have the associated data, whereas those outside won't.

Listing 17.9 Adding scope properties to log messages with `BeginScope`

```
_logger.LogInformation("No, I don't have scope");          #A

using(_logger.BeginScope("Scope value"))                   #B
using(_logger.BeginScope(new Dictionary<string, object>    #C
    {{ "CustomValue1", 12345 } }))                         #C
{
    _logger.LogInformation("Yes, I have the scope!");      #D
}

_logger.LogInformation("No, I lost it again");             #A
```

#A Log messages written outside the scope block don't include the scope state.
#B Calling BeginScope starts a scope block, with a scope state of "Scope value".
#C You can pass anything as the state for a scope.
#D Log messages written inside the scope block include the scope state.

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.[81]

[81] Nicholas Blumhardt, the creator of Serilog and Seq, has examples and the reasoning for this on his blog: https://nblumhardt.com/2016/11/ilogger-beginscope/.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

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.

Dictionary state is added to the log as key-value pairs. The CustomValue1 property is added in this way.

Other state values are added to the Scope property as an array of values

Logs written outside the scope block do not have the additional state.

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 that make up your app's infrastructure, like Kestrel and EF Core. Whichever you choose, I encourage you to add more logs than you *think* you'll need—future-you will thank me when it comes to tracking down a problem!

In the next chapter, we'll look in detail at a variety of web security problems that you should consider when building your apps. ASP.NET Core takes care of some of these issues for

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

you automatically, but it’s important to understand where your app’s vulnerabilities lie, so you can mitigate them as best you can.

17.6 Summary

- Logging is critical to quickly diagnosing errors in production apps. You should always configure logging for your application so that logs are written to a durable location.
- You can add logging to your own services by injecting `ILogger<T>`, where `T` is the name of the service. Alternatively, inject `ILoggerFactory` and call `CreateLogger()`.
- The log level of a message is how important it is and ranges from `Trace` to `Critical`. Typically, you’ll create many low-importance log messages, and a few high-importance log messages.
- You specify the log level of a log by using the appropriate extension method of `ILogger` to create your log. To write an `Information` level log, use `ILogger.LogInformation(message)`.
- The log category indicates which component created the log. It is typically set to the fully qualified name of the class creating the log, but you can set it to any string if you wish. `ILogger<T>` will have a log category of `T`.
- You can format messages with placeholder values, similar to the `string.Format` method, but with meaningful names for the parameters. Calling `logger.LogInfo("Loading Recipe with id {RecipeId}", 1234)` would create a log reading `"Loading Recipe with id 1234"`, but would also capture the value `RecipeId=1234`. This *structured logging* makes analyzing log messages much easier.
- ASP.NET Core includes many logging providers out of the box. These include the console, debug, EventLog, and EventSource providers. Alternatively, you can add third-party logging providers.
- You can configure multiple `ILoggerProvider` instances in ASP.NET Core, which define where logs are output. The `CreateDefaultBuilder` method adds the console and debug providers, and you can add additional providers by calling `ConfigureLogging()`.
- Serilog is a mature logging framework that includes support for a large number of output locations. You can add Serilog to your app with the Serilog.AspNetCore package. This replaces the default `ILoggerFactory` with a Serilog-specific version.
- You can control logging output verbosity using configuration. The `CreateDefaultBuilder` helper uses the `"Logging"` configuration section to control output verbosity. You typically will filter out more logs in production compared to when developing your application.
- Only a single log filtering rule is selected for each logging provider when determining whether to output a log message. The most specific rule is selected based on the logging provider and the category of the log message.
- Structured logging involves recording logs so that they can be easily queried and filtered, instead of the default unstructured format that's output to the console. This makes analyzing logs, searching for issues, and identifying patterns easier.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

- You can add additional properties to a structured log by using scope blocks. A scope block is created by calling `ILogger.BeginScope<T>(state)` in a `using` block. The state can be any object and is added to all log messages inside the scope block.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

18 Improving your application's security

This chapter covers

- **Encrypting traffic using HTTPS and configuring local SSL certificates**
- **Defending against cross-site scripting attacks**
- **Protecting from cross-site request forgery attacks**
- **Allowing calls to your API from other apps using CORS**

Web application security is a hot topic at the moment. Practically every week another breach is reported, or confidential details are leaked. It may seem like the situation is hopeless, but the reality is that the vast majority of breaches could've been avoided with the smallest amount of effort.

In this chapter, we look at a few different ways to protect your application and your application's users from attackers. Because security is an extremely broad topic that covers lots of different avenues, this chapter is by no means an exhaustive guide. It's intended to make you aware of some of the most common threats to your app and how to counteract them, and to highlight areas where you can inadvertently introduce vulnerabilities if you're not careful.

> **TIP** I strongly advise exploring additional resources around security after you've read this chapter. The Open Web Application Security Project (OWASP) (www.owasp.org) is an excellent resource, though it can be a little dry. Alternatively, Troy Hunt (www.troyhunt.com/) has some excellent courses and workshops on security, geared towards .NET developers.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

We’ll start by looking at how to add HTTPS encryption to your website so that users can access your app without the risk of third parties spying on or modifying the content as it travels over the internet. This is effectively mandatory for production apps these days, and it is heavily encouraged by the makers of modern browsers such as Chrome and Firefox. You’ll see how to use the .NET Core development certificate to use HTTPS locally, how to configure an app for HTTPS in production, and how to enforce HTTPS across your whole app.

In sections 18.2 and 18.3, you’ll learn about two potential attacks that should be on your radar: cross-site scripting (XSS) and cross-site request forgery (CSRF). We’ll explore how the attacks work and how you can prevent them in your apps. ASP.NET Core has built-in protection against both types of attack, but you have to remember to use the protection correctly and resist the temptation to circumvent it, unless you’re certain it’s safe to do so.

Section 18.4 deals with a common scenario—you have an application that wants to use JavaScript AJAX (Asynchronous JavaScript and XML) requests to retrieve data from a second app. By default, web browsers block requests to other apps, so you need to enable cross-origin resource sharing (CORS) in your API to achieve this. We’ll look at how CORS works, how to create a CORS policy for your app, and how to apply it to specific action methods.

The final section of this chapter, section 18.5, is a collection of common threats to your application. Each one represents a potentially critical flaw that an attacker could use to compromise your application. The solutions to each threat are generally relatively simple; the important thing is to recognize where the flaws could exist in your own apps so that you can ensure you don’t leave yourself vulnerable.

We’ll start by looking at HTTPS and why you should use it to encrypt the traffic between your users’ browsers and your app. Without HTTPS, attackers could subvert many of the safeguards you add to your app, so it’s an important first step to take.

18.1 Adding HTTPS to an application

In this section you’ll learn about HTTPS, what it is, and why you need to be aware of it for all your production applications. You’ll see two approaches to adding HTTPS to your application: supporting HTTPS directly in your application and using SSL/TLS-offloading with a reverse-proxy. You’ll then learn how to use the development certificate to work with HTTPS on your local machine, and how to add an HTTPS certificate to your app in production. Finally, you’ll learn how to enforce HTTPS in your app using best practices such as security headers, and HTTP redirection.

So far in this book, I’ve shown how the user’s browser sends a request across the internet to your app using the HTTP protocol. We haven’t looked too much into the details of that protocol, other than to establish that it uses *verbs* to describe the type of request (such as GET and POST), that it contains *headers* with metadata about the request, and optionally a *body* payload of data.

By default, HTTP requests are unencrypted; they’re plain text files being sent over the internet. Anyone on the same network as a user (such as using the same public Wi-Fi in a

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

coffee shop) can read the requests and responses sent back and forth. Attackers can even *modify* the requests or responses as they're in transit.

Using unencrypted web apps in this way presents both a privacy and a security risk to your users. Attackers could read the data sent in forms and returned by your app, inject malicious code into your responses to attack users, or steal authentication cookies and impersonate the user on your app.

To protect your users, your app should encrypt the traffic between the user's browser and your app as it travels over the network by using the HTTPS protocol. This is similar to HTTP traffic, but it uses an SSL/TLS[82] certificate to encrypt requests and responses, so attackers cannot read or modify the contents. In browsers, you can tell a site is using HTTPS by the https:// prefix to URLs (notice the "s"), or sometimes, by a padlock, as shown in figure 18.1.

> **TIP** For details about how the SSL/TLS protocols work, see *Real-World Cryptography* by David Wong (Manning, 2021) https://livebook.manning.com/book/real-world-cryptography/chapter-9/.

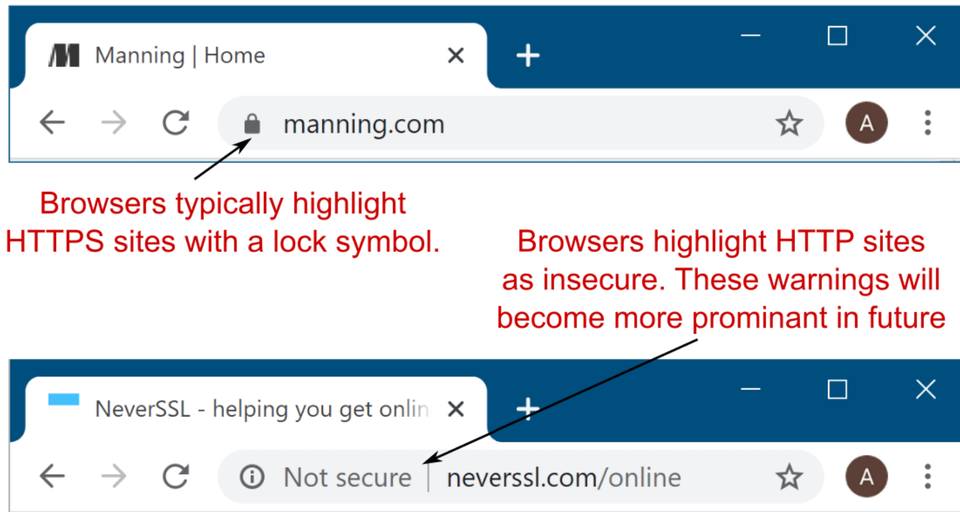


Figure 18.1 Encrypted apps using HTTPS and unencrypted apps using HTTP in Edge. Using HTTPS protects your application from being viewed of tampered with by attackers.

The reality is that, these days, you should always serve your production websites over HTTPS. The industry is pushing toward HTTPS by default, with most browsers moving to mark HTTP

[82] SSL is an older standard that facilitates HTTPS, but the SSL protocol has been superseded by Transport Layer Security (TLS) so I'll be using TLS preferentially throughout this chapter.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

sites as explicitly “not secure.” Skipping HTTPS will hurt the perception of your app in the long run, so even if you’re not interested in the security benefits, it’s in your best interest to set up HTTPS.

In order to configure HTTPS, you need to obtain and configure a TLS certificate for your server. Unfortunately, although that process is a lot easier than it used to be, and is now essentially free thanks to Let’s Encrypt (https://letsencrypt.org/), it’s still far from simple in many cases. If you’re setting up a production server, I recommend carefully following the tutorials on the Let’s Encrpyt site. It’s easy to get it wrong, so take your time!

> **TIP** If you’re hosting your app in the cloud, then most providers will provide “one-click” TLS certificates so that you don’t have to manage certificates yourself. This is *extremely* useful, and I highly recommend it for everyone.[83]

As an ASP.NET Core application developer, you can often get away without *directly* supporting HTTPS in your app by taking advantage of the reverse proxy architecture, as shown in figure 18.2, in a process called SSL/TLS offloading/termination. Instead of your application handling requests using HTTPS directly, it continues to use HTTP. The reverse proxy is responsible for encrypting and decrypting HTTPS traffic to the browser. This often gives you the best of both worlds—data is encrypted between the user’s browser and the server, but you don’t have to worry about configuring certificates in your application.[84]

---

[83] You don’t even have to be hosting your application in the cloud to take advantage of this. Cloudflare (www.cloudflare.com) provides a CDN service that you can add SSL to. You can even use it for free.

[84] If you’re concerned that the traffic is unencrypted between the reverse proxy and your app, then I recommend reading this post by Troy Hunt: http://mng.bz/eHCi. It discusses the pros and cons of the issue as it relates to using Cloudflare to provide HTTPS encryption.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

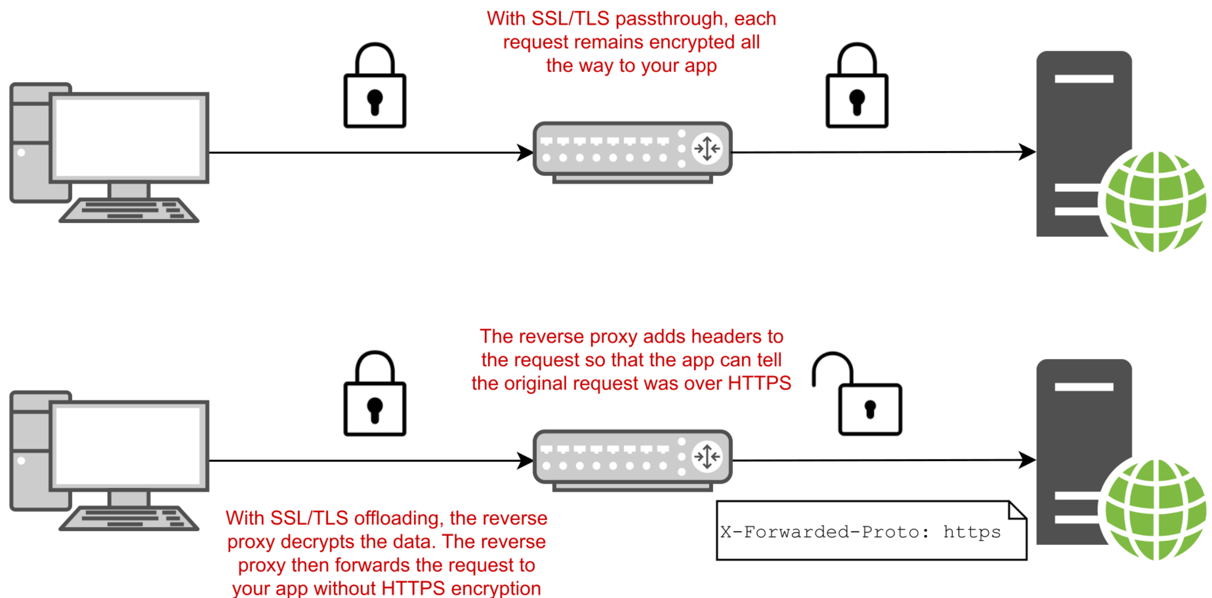


Figure 18.2 You have two options when using HTTPS with a reverse proxy: SSL/TLS passthrough and SSL/TLS offloading. In SSL/TLS passthrough, the data is encrypted all the way to your ASP.NET Core app. For SSL/TLS termination, the reverse proxy handles decrypting the data, so your app doesn't have to.

Depending on the specific infrastructure where you're hosting your app, SSL/TLS could be offloaded to a dedicated device on your network, a third-party service like Cloudflare, or a reverse proxy (such as IIS, NGINX, or HAProxy), running on the same or a different server.

Nevertheless, in some situations, you may need to handle SSL/TLS directly in your app, for example:

- *If you're exposing Kestrel to the internet directly, without a reverse proxy*. This became more common with ASP.NET Core 3.0 due to hardening of the Kestrel server. It is also often the case when you're developing your app locally.
- *If having HTTP between the reverse proxy and your app is not acceptable.* While securing traffic *inside* your network is less critical compared to external traffic, it is undoubtedly more secure to use HTTPS for internal traffic too.
- *If you're using technology that requires HTTPS*. Some newer network protocols, such as gRPC (discussed in chapter 20) and HTTP/2 require an HTTPS connection.

In each of these scenarios, you need to configure a TLS certificate for your application, so Kestrel can receive HTTPS traffic. In section 18.1.1 you'll see the easiest way to get started with HTTPS when developing locally, and in section 18.1.2 you'll see how to configure your application for production.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

18.1.1 Using the .NET Core and IIS Express HTTPS development certificates

Working with HTTPS certificates is easier than it used to be, but unfortunately it can still be confusing topic, especially as a newcomer to the web. The .NET Core SDK, Visual Studio, and IIS Express try and improve this experience by handling a lot of the grunt-work for you.

The first time you run a `dotnet` command using the .NET Core SDK, the SDK installs an HTTPS development certificate onto your machine. Any ASP.NET Core application you create using the default templates (or for which you don't explicitly configure certificates) will use this development certificate to handle HTTPS traffic. However, the development certificate is not *trusted* by default. That means you'll get a browser warning, as shown in figure 18.3 when accessing a site after first installing the .NET Core SDK.

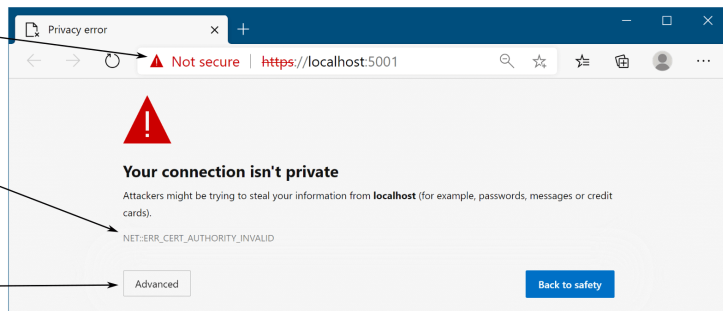


Figure 18.3. The developer certificate is not trusted by default, so apps serving HTTPS traffic using it will be marked as insecure by browsers. Although you can bypass the warnings if necessary, you should instead update the certificate to be trusted.

A brief primer on certificates and signing

HTTPS uses *public key cryptography* as part of the data-encryption process. This uses two keys: a *public* key that *anyone* can see, and a *private* key that only *your* server can see. Anything encrypted with the public key can only be decrypted with the private key. That way, a browser can encrypt something with your server's public key, and only your server can decrypt it. A complete TLS certificate consists of both the public and private parts.

When a browser connects to your app, the server sends the public key part of the TLS certificate. But how does the browser know that it was definitely *your* server that sent the certificate? To achieve this, your TLS certificate contains additional certificates, including a certificate from a third party, a certificate authority (CA). This trusted certificate is called a *root certificate*.

CAs are special trusted entities. Browsers are hardcoded to trust certain root certificates. So, in order for the TLS certificate for your app to be trusted, it must contain (or be signed by) a trusted root certificate.

When you use the .NET Core development certificate, or if you create your own self-signed certificate, your site's HTTPS is missing that trusted root certificate. That means browsers won't trust your certificate and won't connect to your server by default. To get around this, you need to tell your development machine to explicitly trust the certificate.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

In production, you can't use a development or self-signed certificate, as a user's browser won't trust it. Instead, you need to obtain a signed HTTPS certificate from a service like Let's Encrpyt, or from a cloud provider like AWS, Azure, or Cloudflare. These certificates will already be signed by a trusted CA, so will be automatically trusted by browsers.

To solve these browser warnings, you need to *trust* the certificate. Trusting a certificate is a sensitive operation, as it's saying "I know this certificate doesn't look quite right, but just ignore that", so it's hard to do automatically. If you're running on Windows or macOS, you can trust the development certificate by running

```
dotnet dev-certs https --trust
```

This command trusts the certificate, by registering it in the operating system "certificate store". After you run this command, you should be able to access your websites without seeing any warnings or "not secure" labels, as shown in figure 16.4.

TIP You may need to close your browser after trusting the certificate to clear the browser's cache.

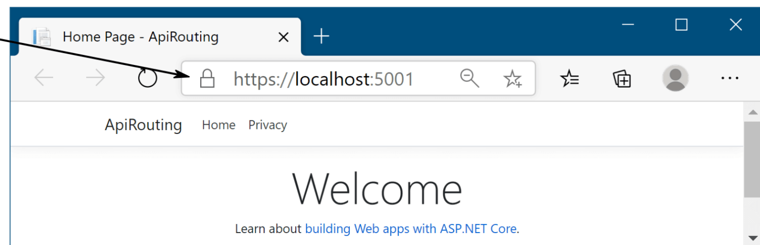


Figure 18.4. Once the development certificate is trusted, you will no longer see browser warnings about the connection.

The developer certificate works smoothly on Windows and macOS. Unfortunately, trusting the certificate in Linux is a little trickier, and depends on the particular flavor you're using. On top of that, software on Linux often uses its own certificate store, so you'll probably need to add the certificate directly to your favorite browser. I suggest looking at the documentation for your favorite browser to figure out the best approach. For advice on other platforms, such as Docker, see the documentation at https://docs.microsoft.com/aspnet/core/security/enforcing-ssl#how-to-set-up-a-developer-certificate-for-docker.

If you're using Windows, Visual Studio, and IIS Express for development, then you may not find the need to trust the development certificate. IIS Express acts as a reverse proxy when you're developing locally, so handles the SSL/TLS setup itself. On top of that, Visual Studio should trust the IIS development certificate as part of installation, so you may never see the browser warnings at all!

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

The .NET Core and IIS development certificates make it easy to use Kestrel with HTTPS locally, but those certificates won't help you once you move to production. In the next section, I show how to configure Kestrel to use a production TLS certificate.

18.1.2 Configuring Kestrel with a production HTTPS certificate

Creating a TLS certificate for production is often a laborious process, as it requires proving to a third-party certificate authority (CA) that you own the domain you're creating the certificate for. This an important step in the "trust" process and ensures that attackers can't impersonate your servers. The result of the process is one or more files, which is the HTTPS certificate you need to configure for your app.

> **TIP** The specifics of how to obtain a certificate vary by provider, and by your OS platform, so follow your provider's documentation carefully. The vagaries and complexities of this process is one of the reasons I strongly favor the SSL/TLS-offloading or "one-click" approaches described previously. Those approaches mean my apps don't need to deal with certificates, and I don't need to use the approaches described in section 18.1.2; I delegate that responsibility to another piece of the network, or the underlying platform.

Once you have a certificate, you need to configure Kestrel to use it to serve HTTPS traffic. In chapter 16, you saw how to set the port your application listens on with the `ASPNETCORE_URLS` environment variable or via the command line, and you saw that you could provide an https URL. As you didn't provide any certificate configuration, Kestrel used the development certificate by default. In production, you need to tell Kestrel which certificate to use.

Kestrel is very configurable, allowing you to configure your certificates in multiple ways. You can use different certificates for different ports, you can load from a .pfx file or from the OS certificate store, or you can have different configuration for each URL endpoint you expose. For full details, see the documentation at https://docs.microsoft.com/aspnet/core/fundamentals/servers/kestrel#endpoint-configuration.

The following listing shows one possible way to set a custom HTTPS certificate for your production app, by configuring the default certificate Kestrel uses for HTTPS connections. You can add the "`Kestrel:Certificates:Default`" section to your appsettings.json file (or using any other configuration source, as described in chapter11) to define the pfx file of the certificate to use. You must also provide the password for accessing the certificate.

Listing 18.1 Configuring the default HTTPS certificate for Kestrel using a pfx file

```
{
  "Kestrel": {                                  #A
    "Certificates": {                           #A
      "Default": {                              #A
        "Path": "localhost.pfx",                #B
        "Password": "testpassword"              #C
      }
    }
  }
```

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
}
```

#A Create a configuration section at Kestrel:Certificates:Default
#B The relative or absolute path to the certificate
#C The password for opening the certificate

The example above is the simplest way to replace the HTTPS certificate, as it doesn't require changing any of Kestrel's defaults. You can use a similar approach to load the HTTPS certificate from the OS certificate store (on Windows or macOS), as shown in the documentation: https://docs.microsoft.com/aspnet/core/fundamentals/servers/kestrel#endpoint-configuration.

> **WARNING** Listing 18.1 has hardcoded the certificate filename and password for simplicity, but you should either load these from a configuration store like user-secrets, as you saw in Chapter 11, or load the certificate from the local store. **Never** put production passwords in your appsettings.json files.

All the default ASP.NET Core templates configure your application to serve both HTTP and HTTPS traffic, and with the configuration you've seen so far, you can ensure your application can handle both HTTP and HTTPS in development and in production.

However, whether you use HTTP or HTTPS may depend on the URL users use when they first browse to your app. For example, if your app listens using the default URLS, http://localhost:5000 for HTTP traffic and https://localhost:5001 for HTTPS traffic, then if a user navigates to the HTTP URL, then their traffic will be unencrypted. Seeing as you've gone to all the trouble to set up HTTPS, it's probably best that you force users to use it!

18.1.3 Enforcing HTTPS for your whole app

Enforcing HTTPS across your whole website is practically required these days. Browsers are beginning to explicitly label HTTP pages as insecure, for security reasons you *must* use TLS any time you're transmitting sensitive data across the internet, and thanks to HTTP/2,[85] adding TLS can *improve* your app's performance.

There are multiple approaches to enforcing HTTPS for your application. If you're using a reverse proxy with SSL/TLS-offloading, then that might be handled for you anyway, without having to worry about it within your apps. Nevertheless, it doesn't hurt to enforce SSL/TLS in your applications too, regardless of what the reverse may be doing.

> **NOTE** If you're building a Web API, rather than a Razor Pages app, then it's common to just reject HTTP requests, without using the approaches described in this section. These protections apply primarily when

[85] HTTP/2 offers many performance improvements over HTTP/1.x, and all modern browsers require HTTPS to enable it. For a great introduction to HTTP/2, see https://developers.google.com/web/fundamentals/performance/http2/.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

building apps to be consumed in a browser. For more details, see https://docs.microsoft.com/aspnet/core/security/enforcing-ssl.

One approach to improving the security of your app is to use HTTP *security headers*. These are HTTP headers, sent as part of your HTTP response, which tell the browser how it should behave. There are many different headers available, most of which restrict the features your app can use in exchange for increased security[86]. In the next chapter, you'll see how to add your own custom headers to your HTTP responses by creating custom middleware.

One of these security headers, the HTTP Strict Transport Security (HSTS) header, can help ensure browsers use HTTPS where it's available, instead of defaulting to HTTP.

ENFORCING HTTPS WITH HTTP STRICT TRANSPORT SECURITY HEADERS

It's unfortunate, but by default, browsers always load apps over HTTP, unless otherwise specified. That means your apps typically must support both HTTP and HTTPS, even if you don't want to serve any traffic over HTTP. One mitigation for this (and a security best practice), is to add HTTP Strict Transport Security headers to your responses.

DEFINITION HTTP Strict Transport Security (HSTS) is a header that instructs the browser to use HTTPS for all *subsequent* requests to your application. The browser will no longer send HTTP requests to your app and will only use HTTPS instead. It can only be sent with responses to HTTPS requests.

HSTS headers are strongly recommended for *production* apps. You generally don't want to enable them for local development, as that would mean you can never run a non-HTTPS app locally! In a similar fashion, you should only use HSTS sites for which you *always* intend to use HTTPS, as it's hard (sometimes impossible) to "turn-off" HTTPS once it's enforced with HSTS!

ASP.NET Core comes with a built-in middleware for setting HSTS headers, which is included in some of the default templates automatically. The following listing shows how you can configure the HSTS headers for your application using the `HstsMiddleware` in Startup.cs.

Listing 18.2 Using HstsMiddleware to add HSTS headers to an application

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
        services.AddHsts(options =>                     #A
        {                                               #A
            options.MaxAge = TimeSpan.FromHours(1);     #A
        });                                             #A
    }
```

[86] Scott Helme has some great guidance on, this and other security headers you can add to your site: https://scotthelme.co.uk/hardening-your-http-response-headers/ .

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsProduction())                           #B
        {
            app.UseHsts();                                #C
        }

        app.UseStaticFiles();                             #D
        app.UseRouting();                                 #D
        app.UseAuthorization();                           #D
        app.UseEndpoints(endpoints =>                     #D
        {                                                 #D
            endpoints.MapRazorPages();                    #D
        });                                               #D
    }
}
```

#A Configure your HSTS header settings. This changes the MaxAge from the default of 30 days.
#B You shouldn't use HSTS in local environments
#C Adds the HstsMiddleware.
#D The HstsMiddleware should be very early in the middleware pipeline.

> **TIP** **The example above shows how to change the** `MaxAge` **sent in the HSTS header. It's a good idea to start with a small value initially. Once you're sure your app's HTTPS is functioning correctly, increase the age for greater security. For more details, on HSTS see https://scotthelme.co.uk/hsts-the-missing-link-in-tls/.**

HSTS is a great option for forcing users to use HTTPS on your website. But one problem with the header is that it can only be added to HTTPS requests. That means you must have *already* made an HTTPS request before HSTS kicks-in: if the *initial* request is HTTP, then no HSTS header is sent, and you *stay* on HTTP! That's unfortunate, but you can mitigate it by redirecting insecure requests to HTTPS immediately.

REDIRECTING FROM HTTP TO HTTPS WITH THE HTTPS REDIRECTION MIDDLEWARE

The `HstsMiddleware` should generally be used in conjunction with middleware that redirects all HTTP requests to HTTPS.

> **TIP** **It's possible to apply HTTPS redirection to only parts of your application, for example to specific Razor Pages, but I don't recommend that, as it's too easy to open up a security hole in your application.**

ASP.NET Core comes with `HttpsRedirectionMiddleware`, which you can use to enforce HTTPS across your whole app. You add it to the middleware pipeline in the `Configure` section of `Startup`, and it ensures that any requests that pass through it are secure. If an HTTP request reaches the `HttpsRedirectionMiddleware`, the middleware immediately short-circuits the pipeline with a redirect to the HTTPS version of the request. The browser will then repeat the request using HTTPS instead of HTTP.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

NOTE The eagle-eyed among you will notice that even with the HSTS and redirection middleware, there is still an inherent weakness. By default, browsers will always make an initial, *insecure*, request over HTTP to your app. The only way to avoid this is by HSTS-preloading, which tells browsers to *always* use HTTPS. You can find a great guide to HSTS, including preloading, here https://www.forwardpmx.com/insights/blog/the-ultimate-guide-to-hsts-protocol/.

The `HttpsRedirectionMiddleware` is added in the default ASP.NET Core templates. It is typically placed after the error handling and `HstsMiddleware`, as shown in the following listing. By default, the middleware redirects all HTTP requests to the secure endpoint, using an HTTP `307 Temporary Redirect` status code.

Listing 18.3 Using `RewriteMiddleware` to enforce HTTPS for an application

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
        app.UseExceptionHandler("/Error");
        if (env.IsProduction())
        {
            app.UseHsts();
        }

        app.UseHttpsRedirection();                    #A
        app.UseStaticFiles();
        app.UseRouting();
        app.UseAuthorization();
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }
}
```

#A Adds the HttpsRedirectionMiddleware to the pipeline. Redirects all HTTP requests to HTTPS.

The `HttpsRedirectionMiddleware` will automatically redirect HTTP requests to the first configured HTTPS endpoint for your application. If your application isn't configured for HTTPS, the middleware *won't* redirect, and instead will log a warning:

```
warn: Microsoft.AspNetCore.HttpsPolicy.HttpsRedirectionMiddleware[3]
      Failed to determine the https port for redirect.
```

If you want the middleware to redirect to a different port than Kestrel knows about, you can configure that by setting the `ASPNETCORE_HTTPS_PORT` environment variable. This is sometimes necessary if you're using a reverse-proxy, and can be set in alternative ways, as described in https://docs.microsoft.com/aspnet/core/security/enforcing-ssl.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

SSL/TLS offloading, header forwarding, and detecting secure requests

At the start of section 18.1, I encouraged you to consider terminating HTTPS requests at a reverse proxy. That way, the user uses HTTPS to talk to the reverse proxy, and the reverse proxy talks to your app using HTTP. With this setup, your users are protected but your app doesn't have to deal with TLS certificates itself.

In order for the `HttpsRedirectionMiddleware` to work correctly, Kestrel needs some way of knowing whether the *original* request that the reverse proxy received was over HTTP or HTTPS. The reverse proxy communicates to your app over HTTP, so Kestrel can't figure that out without extra help.

The standard approach used by most reverse proxies (such as IIS, NGINX, and HAProxy) is to add additional headers to the request before forwarding it to your app. Specifically, a header called `X-Forwarded-Proto` is added, indicating whether the original request protocol was HTTP or HTTPS.

ASP.NET Core includes the `ForwardedHeadersMiddleware` to look for this header (and others) and update the request accordingly, so your app treats a request that was *originally* secured by HTTPS as secure for all intents and purposes.

If you're using IIS with the `UseIisIntegration()` extension, the header forwarding is handled for you automatically. If you're using a different reverse proxy, such as NGINX or HAProxy, then you can enable the middleware by setting the environment variable `ASPNETCORE_FORWARDEDHEADERS_ENABLED=true`, as you saw in chapter 16. Alternatively, you can manually add the middleware to your application, as shown in section 16.3.2.

When the reverse proxy forwards a request, `ForwardedHeadersMiddleware` will look for the `X-Forwarded-Proto` header and will update the request details as appropriate. For all subsequent middleware, the request is considered secure. When adding the middleware manually, it's important you place `ForwardedHeadersMiddleware` before the call to `UseHsts()` or `UseHttpsRedirection()`, so that the forwarded headers are read and the request is marked secure, as appropriate.

HTTPS is one of the most basic requirements for adding security to your application these days. It can be tricky to set up initially, but once you're up and running, you can largely forget about it, especially if you're using SSL/TLS termination at a reverse proxy.

Unfortunately, most other security practices require rather more vigilance to ensure you don't accidentally introduce vulnerabilities into your app as it grows and develops. Many attacks are conceptually simple and have been known about for years, yet they're still commonly found in new applications. In the next section, we look at one such attack and see how to defend against it when building apps using Razor Pages.

18.2 Defending against cross-site scripting (XSS) attacks

In this section, I describe cross-site scripting attacks and how attackers can use them to compromise your users. I show how the Razor Pages framework protects you from these attacks, how to disable the protections when you need to, and what to look out for. I also discuss the difference between HTML-encoding and JavaScript-encoding, and the impact of using the wrong encoder.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Attackers can exploit a vulnerability in your app to create cross-site scripting (XSS) attacks[87] that execute code in another user's browser. Commonly, attackers submit content using a legitimate approach, such as an input form, which is later rendered somewhere to the page. By carefully crafting malicious input, the attacker can execute arbitrary JavaScript on a user's browsers, and so can steal cookies, impersonate the user, and generally do bad things.

Figure 18.5 shows a basic example of an XSS attack. Legitimate users of your app can send their name to your app by submitting a form. The app then adds the name to an internal list and renders the whole list to the page. If the names are not rendered safely, then a malicious user can execute JavaScript in the browser of every other user that views the list.

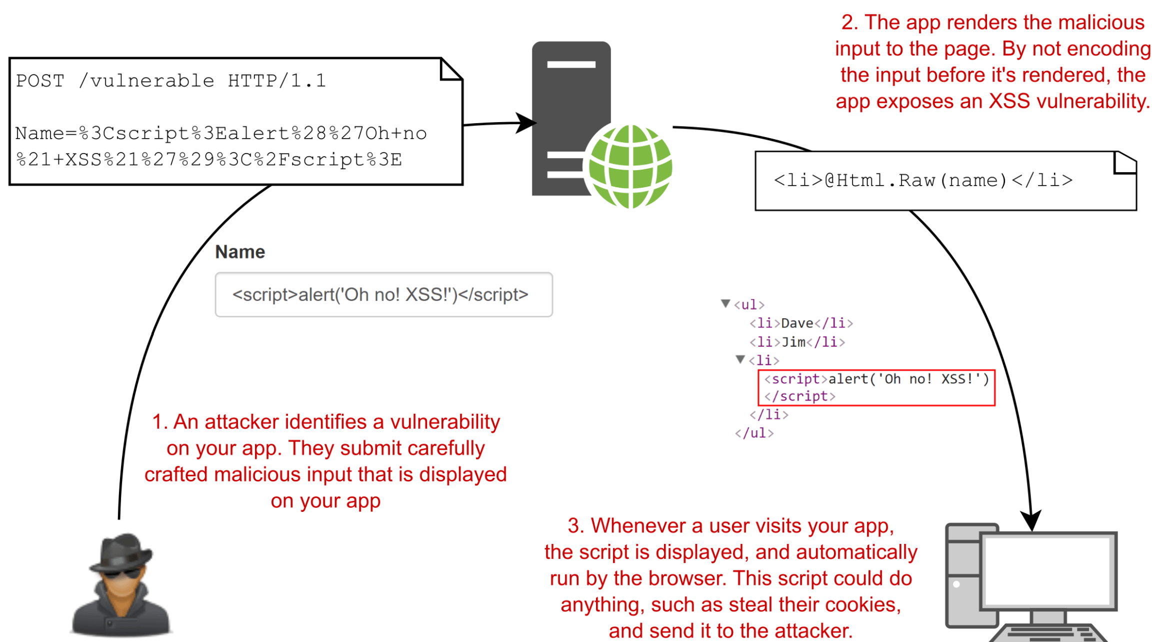


Figure 18.5 How an XSS vulnerability is exploited. An attacker submits malicious content to your app, which is displayed in the browser of other users. If the app doesn't encode the content when writing to the page, the input becomes part of the HTML of the page and can run arbitrary JavaScript.

In figure 18.5, the user entered a snippet of HTML such as their name. When users view the list of names, the Razor template renders the names using `@Html.Raw()`, which writes the `<script>` tag directly to the document. The user's input has become part of the page's HTML

[87] For a detailed discussion, see OWASP at https://owasp.org/www-community/attacks/xss/.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

structure. As soon as the page is loaded in the user's browser, the `<script>` tag executes, and the user is compromised. Once an attacker can execute arbitrary JavaScript on a user's browser, they can do pretty much anything.

The vulnerability here is due to rendering the user input in an unsafe way. If the data isn't encoded to make it safe before it's rendered, then you could open your users to attack. *By default, Razor protects against XSS attacks* by HTML-encoding any data written using Tag Helpers, HTML Helpers, or the `@` syntax. So, generally, you should be safe, as you saw in chapter 7.

Using `@Html.Raw()` is where the danger lies—if the HTML you're rendering contains user input (even indirectly), then you could have an XSS vulnerability. By rendering the user input with `@` instead, the content is encoded before it's written to the output, as shown in figure 18.6.

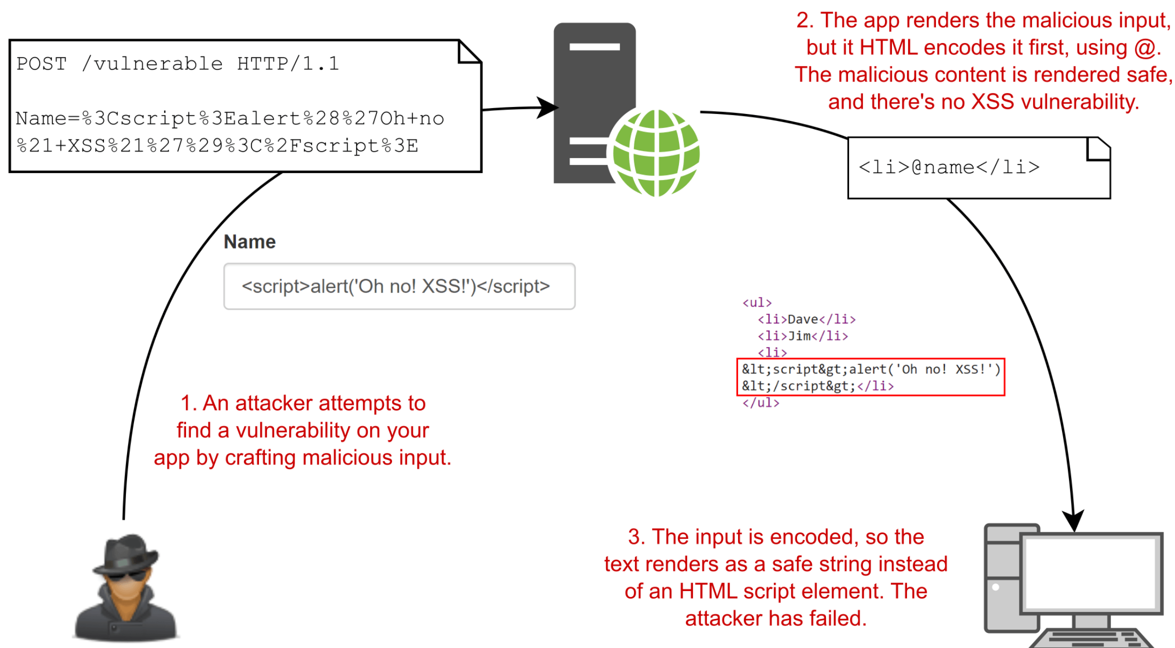


Figure 18.6 Protecting against XSS attacks by HTML encoding user input using `@` in Razor templates. The <script> tag is encoded so that it is no longer rendered as HTML, and can't be used to compromise your app.

This example demonstrates using HTML encoding to prevent elements being directly added to the HTML DOM, but it's not the only case you have to think about. If you're passing untrusted data to JavaScript, or using untrusted data in URL query values, you must make sure you encode the data correctly.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

A common scenario is when you're using jQuery or JavaScript with Razor pages, and you want to pass a value from the server to the client. If you use the standard `@` symbol to render the data to the page, then the output will be HTML-encoded. Unfortunately, if you HTML encode a string and inject it directly into JavaScript, you probably won't get what you expect.

For example, if you have a variable in your Razor file called `name`, and you want to make it available in JavaScript, you might be tempted to use something like

```
<script>var name = '@name'</script>
```

If the name contains special characters, Razor will encode them using HTML encoding, which probably isn't what you want in this JavaScript context. For example, if `name` was `Arnold "Arnie" Schwarzenegger`, then rendering it as you did previously would give:

```
<script>var name = 'Arnold &quot;Arnie&quot; Schwarzenegger';</script>
```

Note how the quotation marks `"` have been HTML-encoded to `"`. If you use this value in JavaScript directly, expecting it to be a "safe" encoded value, then it's going to look wrong, as shown in figure 18.7.

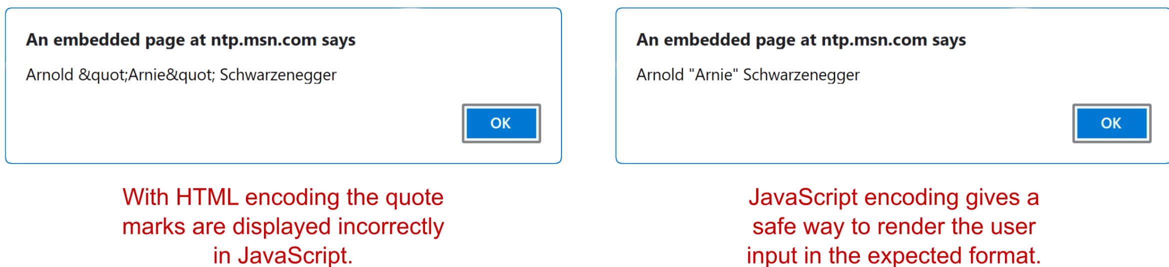


Figure 18.7 Comparison of alerts when using JavaScript encoding compared to HTML encoding

Instead, you should encode the variable using JavaScript encoding so that the `"` is rendered as a safe Unicode character, `\u0022`. You can achieve this by injecting a `Java-ScriptEncoder` into the view (as you saw in chapter 10) and calling `Encode()` on the `name` variable:

```
@inject System.Text.Encodings.Web.JavaScriptEncoder encoder;
<script>var name = '@encoder.Encode(name)'</script>
```

To avoid having to remember to use JavaScript encoding, I recommend you don't write values into JavaScript like this. Instead, write the value to an HTML element's attributes, and then read that into the JavaScript variable later. That avoids the need for the JavaScript encoder entirely.

Listing 18.4 Passing values to JavaScript by writing them to HTML attributes

```
<div id="data" data-name="@name"></div>        #A
<script>
```


Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
var ele = document.getElementById('data');      #B
var name = ele.getAttribute('data-name');       #C
</script>
```

#A Write the value you want in JavaScript to a data-* attribute. This will HTML encode the data.
#B Gets a reference to the HTML element
#C Reads the data-* attribute into JavaScript, which will convert it to JavaScript encoding

XSS attacks are still common, and it's easy to expose yourself to them whenever you allow users to input data. Validation of the incoming data can sometimes help, but it's often a tricky problem. For example, a naive name validator might require that you only use letters, which would prevent most attacks. Unfortunately, that doesn't account for users with hyphens or apostrophes in their name, let alone users with non-western names. People get (understandably) upset when you tell them their name is invalid, so be wary of this approach!

Whether or not you use strict validation, you should always encode the data when you render it to the page. Think carefully whenever you find yourself writing `@Html.Raw()`. Is there any way for a user to get malicious data into that field? If so, you'll need to find another way to display the data.

XSS vulnerabilities allow attackers to execute JavaScript on a user's browser. The next vulnerability we're going to consider lets them make requests to your API as though they're a different logged-in user, even when the user isn't using your app. Scared? I hope so!

18.3 Protecting from cross-site request forgery (CSRF) attacks

In this section you'll learn about cross-site request forgery attacks, how attackers can use them to impersonate a user on your site, and how to protect against them using anti-forgery tokens. Razor Pages protects you from these attacks by default, but you can disable these verifications, so it's important to understand the implications of doing so.

Cross-site request forgery (CSRF) attacks can be a problem for websites or APIs that use cookies for authentication. A CSRF attack involves a malicious website making an authenticated request to your API on behalf of the user, without the user initiating the request. In this section, we'll explore how these attacks work and how you can mitigate them with anti-forgery tokens.

The canonical example of this attack is a bank transfer/withdrawal. Imagine you have a banking application that stores authentication tokens in a cookie, as is common (especially in traditional server-side rendered applications). Browsers automatically send the cookies associated with a domain with every request, so the app knows whether a user is authenticated.

Now imagine your application has a page that lets a user transfer funds from their account to another account, using a POST request to the `Balance` Razor Page. You have to be logged in to access the form (you've protected the Razor Page with the `[Authorize]` attribute), but otherwise you post a form that says how much you want to transfer, and where you want to transfer it.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Imagine a user visits your site, logs in, and performs a transaction. They then visit a second website that the attacker has control of. The attacker has embedded a form on their website that performs a POST to your bank's website, identical to the transfer funds form on your banking website. This form does something malicious, such as transfer all the user's funds to the attacker, as shown in figure 18.8. Browsers automatically send the cookies for the application when the page does a full form post, and the banking app has no way of knowing that this is a malicious request. The unsuspecting user has given all their money to the attacker!

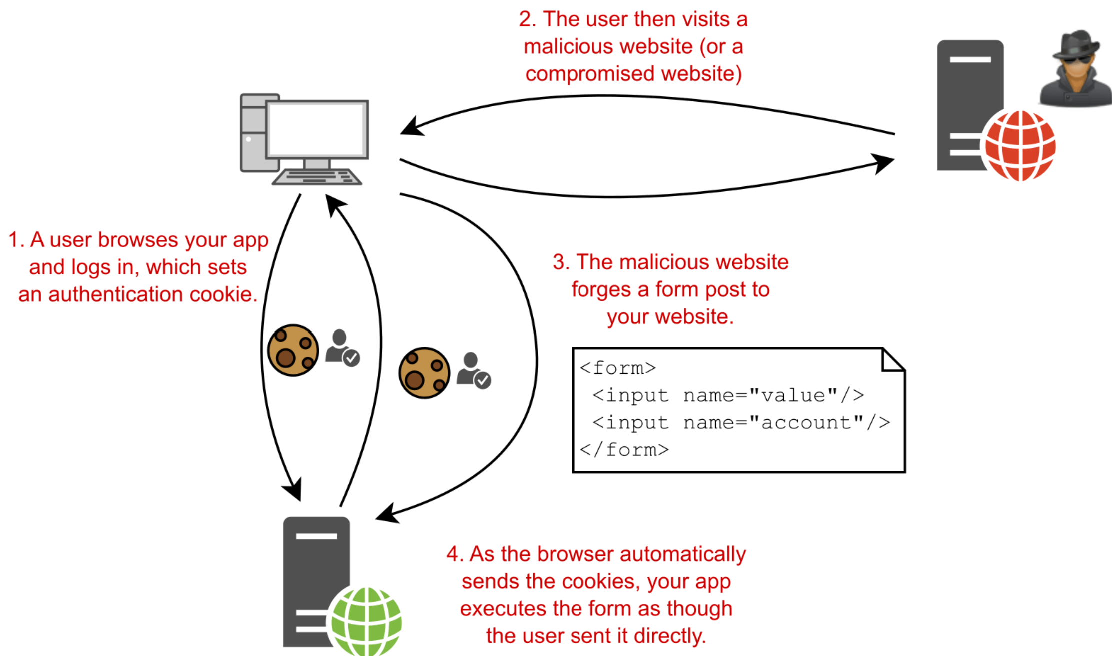


Figure 18.8 A CSRF attack occurs when a logged-in user visits a malicious site. The malicious site crafts a form that matches one on your app and POSTs it to your app. The browser sends the authentication cookie automatically, so your app sees the request as a valid request from the user.

The vulnerability here revolves around the fact that browsers automatically send cookies when a page is requested (using a GET request) or a form is POSTed. There's no difference between a legitimate POST of the form in your banking app and the attacker's malicious POST.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Unfortunately, this behavior is baked into the web; it's what allows you to navigate websites seamlessly after initially logging in.

A common solution to the attack is the *synchronizer token* pattern,[88] which uses user-specific, unique, anti-forgery tokens to enforce a difference between a legitimate POST and a forged POST from an attacker. One token is stored in a cookie and another is added to the form you wish to protect. Your app generates the tokens at runtime based on the current logged-in user, so there's no way for an attacker to create one for their forged form.

When the `Balance` Razor Page receives a form POST, it compares the value in the form with the value in the cookie. If either value is missing, or they don't match, then the request is rejected. If an attacker creates a POST, then the browser will post the cookie token as usual, but there won't be a token in the form itself, or the token won't be valid. The Razor Page will reject the request, protecting from the CSRF attack, as in figure 18.9.

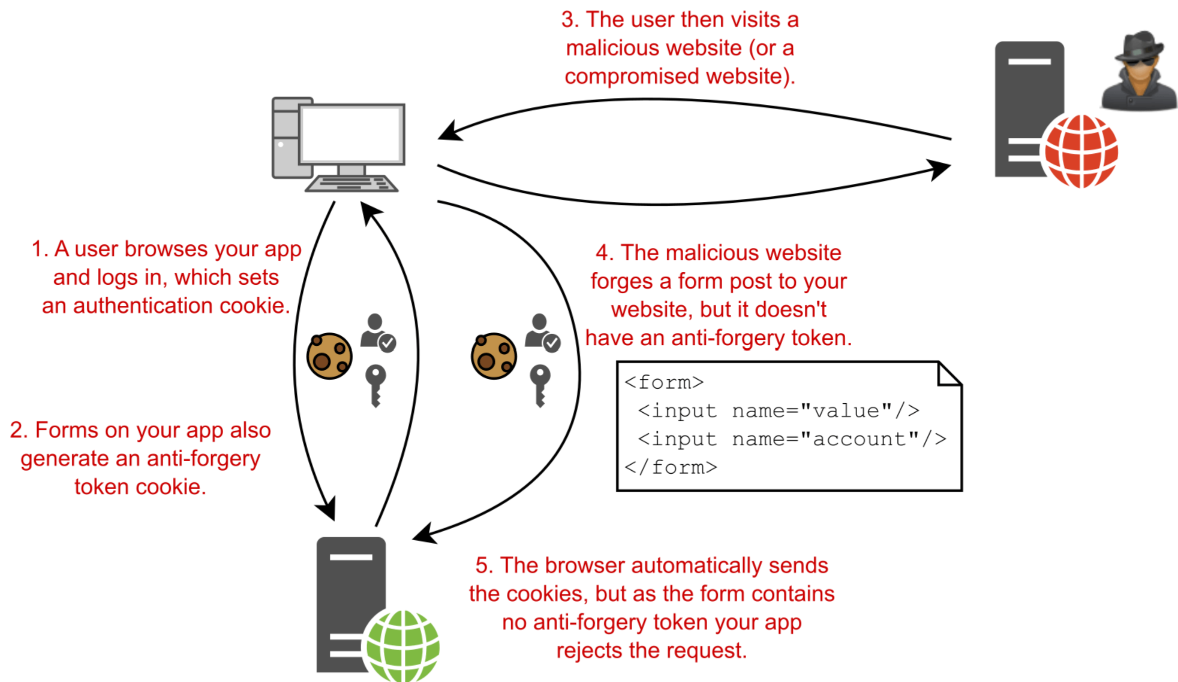


Figure 18.9 Protecting against a CSRF attack using anti-forgery tokens. The browser automatically forwards the cookie token, but the malicious site can't read it, and so can't include a token in the form. The app rejects the malicious request as the tokens don't match.

[88] The OWASP site gives a thorough discussion of the CSRF vulnerability, including the synchronizer token pattern: https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

The good news is that Razor Pages automatically protects you against CSRF attacks! The Form Tag Helper automatically sets an anti-forgery token cookie and renders the token to a hidden field called `__RequestVerificationToken` for every `<form>` element in your app (unless you specifically disable them). For example, take this simple Razor template that posts back to the same Razor Page:

```
<form method="post">
    <label>Amount</label>
    <input type="number" name="amount" />
    <button type="submit">Withdraw funds</button>
</form>
```

When rendered to HTML, the anti-forgery token is stored in the hidden field and posted back with a legitimate request:

```
<form method="post">
    <label>Amount</label>
    <input type="number" name="amount" />
    <button type="submit" >Withdraw funds</button>
    <input name="__RequestVerificationToken" type="hidden"
        value="CfDJ8Daz26qb0hBGsw7QCK">
</form>
```

ASP.NET Core automatically adds the anti-forgery tokens to every form, and Razor Pages automatically validates them. The framework ensures the anti-forgery tokens exist in both the cookie and the form data, ensures that they match, and will reject any requests where they don't.

If you're using MVC controllers with views instead of Razor Pages, ASP.NET Core still adds the anti-forgery tokens to every form. Unfortunately, it *doesn't* validate them for you. Instead, you have to decorate your controllers and actions with `[ValidateAntiForgeryToken]` attributes. These ensure the anti-forgery tokens exist in both the cookie and the form data, ensures they match, and will reject any requests where they don't.

> **WARNING** **ASP.NET Core *doesn't* automatically validate anti-forgery tokens if you're using MVC controllers with Views. You must make sure you mark all vulnerable methods with `[ValidateAntiForgeryToken]` attributes instead, as described in the documentation: https://docs.microsoft.com/aspnet/core/security/anti-request-forgery. Note that if you're using Web API controllers and are *not* using cookies for authentication, then you are not vulnerable to CSRF attacks.**

Generally, you only need to use anti-forgery tokens for POST, DELETE, and other dangerous request types that are used for modifying state. GET requests shouldn't be used for this purpose, so the framework doesn't require valid anti-forgery tokens to call them. Razor Pages validates anti-forgery tokens for dangerous verbs like POST and ignores safe verbs like GET. As long as you create your app following this pattern (and you should!), then the framework will do the right thing to keep you safe.

If you need to explicitly ignore anti-forgery tokens on a Razor Page for some reason, you can disable the validation by applying the `[IgnoreAntiforgeryToken]` attribute to a Razor

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Page's `PageModel`. This bypasses the framework protections, for those cases where you're doing something that you know is safe and doesn't need protecting, but in most cases it's better to play it safe and validate, just in case.

CSRF attacks can be a tricky thing to get your head around from a technical point of view, but for the most part everything *should* work without much effort on your part. Razor will add anti-forgery tokens to your forms, and the Razor Pages framework will take care of validation for you.

Where things get trickier is if you're making a lot of requests to an API using JavaScript and you're posting JSON objects rather than form data. In these cases, you won't be able to send the verification token as part of a form (as you're sending JSON), so you'll need to add it as a header in the request instead.[89]

> **TIP** If you're not using cookie authentication, and instead have an SPA that sends authentication tokens in a header, then good news—you don't have to worry about CSRF at all! Malicious sites can only send cookies, not headers, to your API, so they can't make authenticated requests.

Generating unique tokens with the Data Protection APIs

The anti-forgery tokens used to prevent CSRF attacks rely on the ability of the framework to use strong symmetric encryption to encrypt and decrypt data. Encryption algorithms typically rely on one or more keys, which are used to initialize the encryption and to make the process reproducible. If you have the key, you can encrypt and decrypt data; without it, the data is secure.

In ASP.NET Core, encryption is handled by the Data Protection APIs. They're used to create the anti-forgery tokens, to encrypt authentication cookies, and to generate secure tokens in general. Crucially, they also control the management of the *key files* that are used for encryption.

A key file is a small XML file that contains the random key value used for encryption in ASP.NET Core apps. It's critical that it's stored securely—if an attacker got hold of it, they could impersonate any user of your app, and generally do bad things!

The Data Protection system stores the keys in a safe location, depending on how and where you host your app. For example:

- *Azure Web App*—In a special synced folder, shared between regions.
- *IIS without user profile*—Encrypted in the registry.
- *Account with user profile*—In %LOCALAPPDATA%\ASP.NET\DataProtection-Keys on Windows, or ~/.aspnet/DataProtection-Keys on Linux or macOS.
- *All other cases*—In memory. When the app restarts, the keys will be lost.

So why do you care? In order for your app to be able to read your users' authentication cookies, it must decrypt them using the same key that was used to encrypt them. If you're running in a web-farm scenario, then, by default, each server will have its own key and won't be able to read cookies encrypted by other servers.

To get around this, you must configure your app to store its data protection keys in a central location. This could be a shared folder on a hard drive, a Redis instance, or an Azure blob storage instance, for example.

---

[89] Exactly how you do this varies depending on the JavaScript framework you're using. The documentation contains examples using JQuery and AngularJS, but you should be able to extend this to your JavaScript framework of choice: http://mng.bz/54Sl.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

The documentation on the data protection APIs is extremely detailed, but it can be overwhelming. I recommend reading the section on configuring data protection, (http://mng.bz/d40i) and configuring a key storage provider for use in a web-farm scenario (http://mng.bz/5pW6).

It's worth clarifying that the CSRF vulnerability discussed in this section requires that a malicious site does a full form POST to your app. The malicious site can't make the request to your API using *client-side only* JavaScript, as browsers will block JavaScript requests to your API that are from a different origin.

This is a safety feature, but it can often cause you problems. If you're building a client-side SPA, or even if you have a little JavaScript on an otherwise server-side rendered app, you may find you need to make such *cross-origin* requests. In the next section, I'll describe a common scenario you're likely to run into and show how you can modify your apps to work around it.

18.4 Calling your web APIs from other domains using CORS

In this section you'll learn about cross-origin resource sharing (CORS), a protocol to allow JavaScript to make requests from one domain to another. CORS is a frequent area of confusion for many developers, so this section describes why it's necessary, and how CORS headers work. You'll then learn how to add CORS to both your whole application and specific Web API actions, and how to configure multiple CORS policies for your application.

As you've already seen, CSRF attacks can be powerful, but they would be even more dangerous if it weren't for browsers implementing the *same-origin* policy. This policy blocks apps from using JavaScript to call a web API at a different location unless the web API explicitly allows it.

> **DEFINITION** *Origins* are deemed the same if they match the scheme (HTTP or HTTPS), domain (example.com), *and* port (80 by default for HTTP, and 443 for HTTPS). If an app attempts to access a resource using JavaScript and the origins aren't identical, the browser blocks the request.

The same-origin policy is strict—the origins of the two URLs must be identical for the request to be allowed. For example, the following origins are the same:

- http://example.com/home
- http://example.com/site.css

The paths are different for these two URLs (`/home` and `/sites.css`), but the scheme, domain, and port (80) are identical. So, if you were on the homepage of your app, you could request the `/sites.css` file using JavaScript without any issues.

In contrast, the origins of the following sites are all different, so you couldn't request any of these URLs using JavaScript from the http://example.com origin:

- *https://example.com*—Different scheme (https)
- *http://www.example.com*—Different domain (includes a subdomain)

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

- *http://example.com:5000*—Different port (default HTTP port is 80)

For simple apps, where you have a single web app handling all of your functionality, this limitation might not be a problem, but it's extremely common for an app to make requests to another domain.

For example, you might have an e-commerce site hosted at http://shopping.com, and you're attempting to load data from http://api.shopping.com to display details about the products available for sale. With this configuration, you'll fall foul of the same-origin policy. Any attempt to make a request using JavaScript to the API domain will fail, with an error similar to figure 18.10.

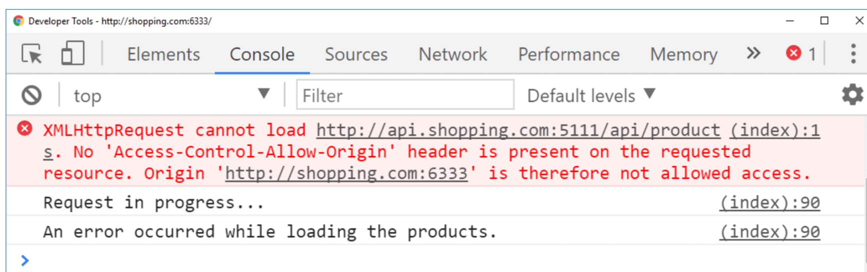


Figure 18.10 The console log for a failed cross-origin request. Chrome has blocked a cross-origin request from the app http://shopping.com:6333 to the API at http://api.shopping.com:5111.

The need to make cross-origin requests from JavaScript is increasingly common with the rise of client-side SPAs and the move away from monolithic apps. Luckily, there's a web standard that lets you work around this in a safe way; this standard is called cross-origin resource sharing (CORS). You can use CORS to control which apps can call your API, so you can enable scenarios like the one described earlier.

18.4.1 Understanding CORS and how it works

CORS is a web standard that allows your Web API to make statements about who can make cross-origin requests to it. For example, you could make statements such as

- Allow cross-origin requests from http://shopping.com and https://app.shopping.com
- Only allow GET cross-origin requests
- Allow returning the `Server` header in responses to cross-origin requests
- Allow credentials (such as authentication cookies or authorization headers) to be sent with cross-origin requests

You can combine these rules into a *policy* and apply different policies to different endpoints of your API. You could apply a policy to your entire application, or a different policy to every API action.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

CORS works using HTTP headers. When your Web API application receives a request, it sets special headers on the response to indicate whether cross-origin requests are allowed, which origins they're allowed from, and which HTTP verbs and headers the request can use—pretty much everything about the request.

In some cases, before sending a real request to your API, the browser sends a *preflight* request. This is a request sent using the OPTIONS verb, which the browser uses to check whether it's allowed to make the real request. If the API sends back the correct headers, the browser will send the true cross-origin request, as shown in figure 18.11.

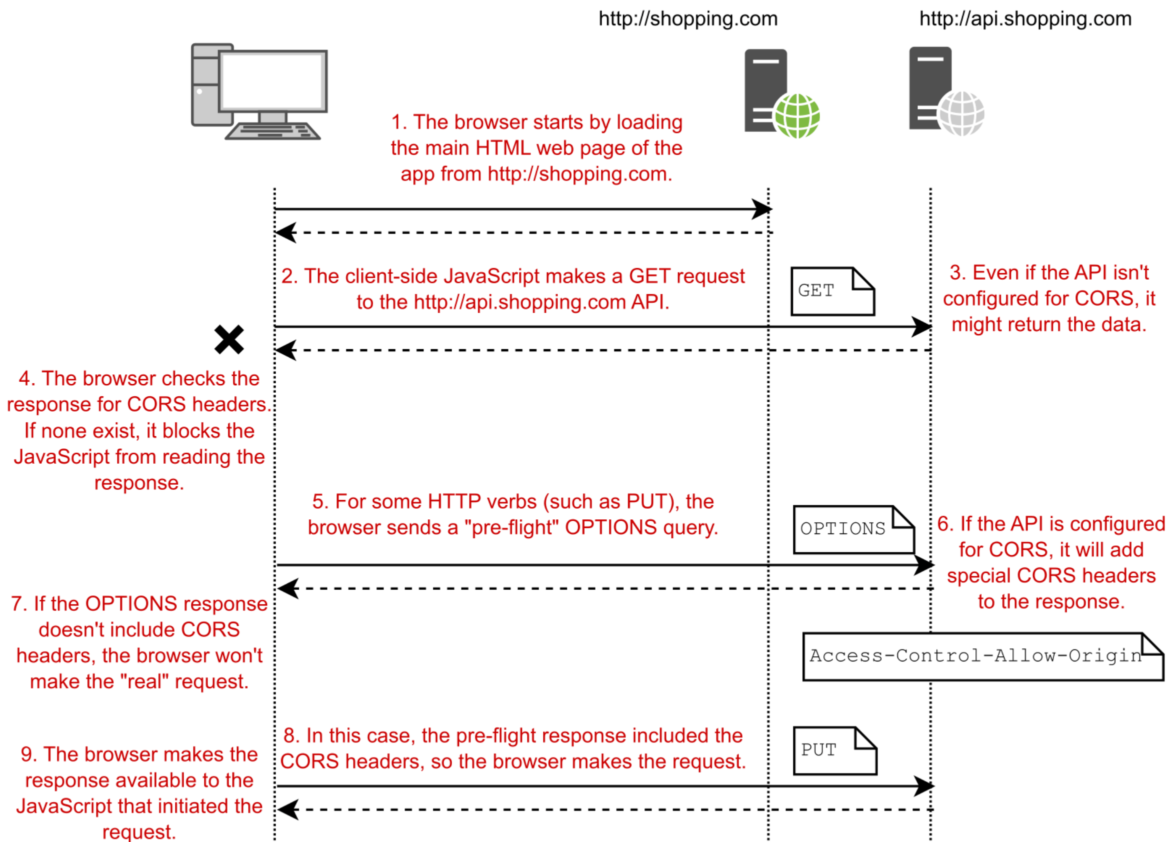


Figure 18.11 Two cross-origin-requests. The response to the first request doesn't contain any CORS headers, so the browser blocks the app from reading it. The second request requires a preflight OPTIONS request, to check if CORS is enabled. As the response contains CORS headers, the real request can be made, and the response provided to the JavaScript app.

TIP For a more detailed discussion of CORS, see *CORS in Action* by Monsur Hossain (Manning, 2014), available at http://mng.bz/aD41.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

The CORS specification, as with many technical documents, is pretty complicated,[90] with a variety of headers and processes to contend with. Thankfully, ASP.NET Core handles the details of the specification for you, so your main concern is working out exactly *who* needs to access your API, and under what circumstances.

18.4.2 Adding a global CORS policy to your whole app

Typically, you shouldn't set up CORS for your APIs until you need it. Browsers block cross-origin communication for a reason—it closes an avenue of attack—they're not being awkward! Wait until you have an API on a different domain to an app that needs to access it.

Adding CORS support to your application requires four things:

- Add the CORS services to your app.
- Configure at least one CORS policy.
- Add the CORS middleware to your middleware pipeline.
- Either set a default CORS policy for your entire app or decorate your Web API actions with the `[EnableCors]` attribute to selectively enable CORS for specific endpoints.

Adding the CORS services to your application involves calling `AddCors()` in your `Startup.ConfigureServices` method:

```
services.AddCors();
```

The bulk of your effort in configuring CORS will go into policy configuration. A CORS policy controls how your application will respond to cross-origin requests. It defines which origins are allowed, which headers to return, which HTTP methods to allow, and so on. You normally define your policies inline when you add the CORS services to your application.

For example, consider the previous e-commerce site example. You want your API that is hosted at http://api.shopping.com to be available from the main app via client-side JavaScript, hosted at http://shopping.com. You therefore need to configure the *API* to allow cross-origin requests.

> **NOTE** Remember, it's the main app that will get errors when attempting to make cross-origin requests, but it's the ***API*** you're accessing that you need to add CORS to, ***not*** the app making the requests.

The following listing shows how to configure a policy called `"AllowShoppingApp"` to enable cross-origin requests from http://shopping.com to the API. Additionally, we explicitly allow any HTTP verb type; without this call, only simple methods (GET, HEAD, and POST) are allowed. The policies are built up using the familiar "fluent builder" style you've seen throughout this book.

---

[90] If that's the sort of thing that floats your boat, you can read the spec here: https://fetch.spec.whatwg.org/#http-cors-protocol.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Listing 18.5 Configuring a CORS policy to allow requests from a specific origin

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddCors(options => {                          #A
        options.AddPolicy("AllowShoppingApp", policy =>    #B
            policy.WithOrigins("http://shopping.com")      #C
                .AllowAnyMethod());                        #D
    });
    // other service configuration
}
```

#A The AddCors method exposes an Action<CorsOptions> overload.
#B Every policy has a unique name.
#C The WithOrigins method specifies which origins are allowed. Note the URL has no trailing /.
#D Allows all HTTP verbs to call the API

> **WARNING** When listing origins in `WithOrigins()`, ensure that they don't have a trailing `"/"`, otherwise the origin will never match and your cross-origin requests will fail.

Once you've defined a CORS policy, you can apply it to your application. In the following listing, apply the `"AllowShoppingApp"` policy to the whole application using `CorsMiddleware` by calling `UseCors()` in the `Configure` method of Startup.cs.

Listing 18.6 Adding the CORS middleware and configuring a default CORS policy

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseRouting();                       #A

    app.UseCors("AllowShoppingApp");        #B
    app.UseAuthentication();
    app.UseAuthorization();

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

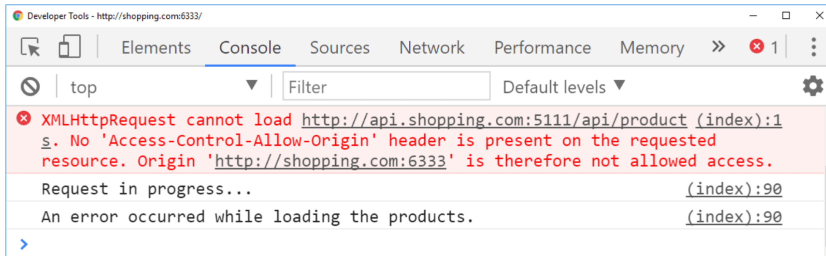
#A The CORS middleware must come after the call to UseRouting()
#B Adds the CORS middleware and uses AllowShoppingApp as the default policy
#B Place the CORS middleware before the endpoint middleware

> **NOTE** As with all middleware, the order of the CORS middleware is important. You must place the call to `UseCors()` *after* `UseRouting()` and before `UseEndpoints()`.The CORS middleware needs to intercept cross-origin requests to your Web API actions, so it can generate the correct responses to preflight requests and add the necessary headers. It's typical to place the CORS middleware before the call to `UseAuthentication()`.

With the CORS middleware in place for the API, the shopping app can now make cross-origin requests. You can call the API from the http://shopping.com site and the browser lets the

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

CORS request through, as shown in figure 18.12. If you make the same request from a domain other than http://shopping.com, the request continues to be blocked.



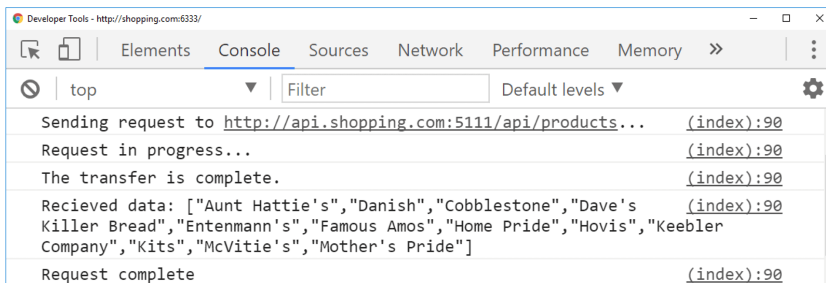


Figure 18.12 With CORS enabled, as in the lower image, cross-origin requests can be made and the browser will make the response available to the JavaScript. Compare this to the upper image, in which the request was blocked.

Applying a CORS policy globally to your application in this way may be overkill. If there's only a subset of actions in your API that need to be accessed from other origins, then it's prudent to only enable CORS for those specific actions. This can be achieved with the `[EnableCors]` attribute.

18.4.3 Adding CORS to specific Web API actions with EnableCorsAttribute

Browsers block cross-origin requests by default for good reason—they have the potential to be abused by malicious or compromised sites. Enabling CORS for your entire app may not be worth the risk if you know that only a subset of actions will ever need to be accessed cross-origin.

If that's the case, it's best to only enable a CORS policy for those specific actions. ASP.NET Core provides the `[EnableCors]` attribute, which lets you select a policy to apply to a given controller or action method.

This approach lets you apply different CORS policies to different action methods. For example, you could allow GET requests access to your entire API from the

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

http://shopping.com domain, but only allow other HTTP verbs for a specific controller, while allowing anyone to access your product list action method.

You define these policies in `ConfigureServices` using `AddPolicy()` and giving the policy a name, as you saw in listing 18.5. However, instead of calling `UseCors("AllowShoppingApp")` as you saw in listing 18.6, add the middleware without a default policy, by calling `UseCors()` only.

To apply a policy to a controller or an action method, apply the `[EnableCors]` attribute, as shown in the following listing. An `[EnableCors]` attribute on an action takes precedence over an attribute on a controller, or you can use the `[DisableCors]` attribute to disable cross-origin access to the method entirely.

Listing 18.7 Applying the `EnableCors` attribute to a controller and action

```
[EnableCors("AllowShoppingApp")]                                          #A
public class ProductController: Controller
{
    [EnableCors("AllowAnyOrigin")                                         #B
    public IActionResult GeteProducts() { /* Method */ }

    public IActionResult GeteProductPrice(int id) { /* Method */ }   #C

    [DisableCors]
    public IActionResult DeleteProduct(int id) { /* Method */ }      #D
}
```

#A Applies the AllowShoppingApp CORS policy to every action method
#B The AllowAnyOrigin policy is “closer” to the action, so it takes precedence.
#C The AllowShoppingApp policy (from the controller) will be applied.
#D The DisableCors attribute disables CORS for the action method completely.

If you want to apply a CORS policy to most of your actions, but want to use a different policy or disable CORS entirely for some actions, you can pass a default policy when configuring the middleware using `UseCors("AllowShoppingApp")` for example. Actions decorated with `[EnableCors("OtherPolicy")]` will apply `OtherPolicy` preferentially, and actions decorated with `[DisableCors]` will not have CORS enabled at all.

Whether you choose to use a single, a default CORS policy or multiple policies, you need to configure the CORS policies for your application in `ConfigureServies`. Many different options are available when configuring CORS. In the next section, I provide an overview of the possibilities.

18.4.4 Configuring CORS policies

Browsers implement the cross-origin policy for security reasons, so you should carefully consider the implications of relaxing any of the restrictions they impose. Even if you enable cross-origin requests, you can still control what data cross-origin requests can send, and what your API will return. For example, you can configure:

- The origins that may make a cross-origin request to your API

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

- The HTTP verbs (such as GET, POST, and DELETE) that can be used
- The headers the browser can send
- The headers that the browser can read from your app's response
- Whether the browser will send authentication credentials with the request

You define all of these options when creating a CORS policy in your call to `AddCors()` using the `CorsPolicyBuilder`, as you saw in listing 18.5. A policy can set all, or none of these options, so you can customize the results to your heart's content. Table 18.1 shows some of the options available, and their effects.

Table 18.1 The methods available for configuring a CORS policy, and their effect on the policy

`CorsPolicyBuilder` method example	Result
`WithOrigins("http://shopping.com")`	Allows cross-origin requests from http://shopping .com.
`AllowAnyOrigin()`	Allows cross-origin requests from any origin. This means *any* website can make JavaScript requests to your API.
`WithMethods()/AllowAnyMethod()`	Sets the allowed methods (such as GET, POST, and DELETE) that can be made to your API.
`WithHeaders()/AllowAnyHeader()`	Sets the headers that the browser may send to your API. If you restrict the headers, you must include at least `"Accept"`, `"Content-Type"`, and `"Origin"` to allow valid requests.
`WithExposedHeaders()`	Allows your API to send extra headers to the browser. By default, only the `Cache-Control`, `Content-Language`, `Content-Type`, `Expires`, `Last-Modified`, and `Pragma` headers are sent in the response.
`AllowCredentials()`	By default, the browser won't send authentication details with cross-origin requests unless you explicitly allow it. You must also enable sending credentials client-side in JavaScript when making the request.

One of the first issues in setting up CORS is realizing you have a cross-origin problem at all. Several times I've been stumped trying to figure out why a request won't work, until I realize the request is going cross-domain, or from HTTP to HTTPS, for example.

Whenever possible, I recommend avoiding cross-origin requests completely. You can end up with subtle differences in the way browsers handle them, which can cause more headaches. In particular, avoid HTTP to HTTPS cross-domain issues by running all of your applications behind HTTPS. As discussed in section 18.1, that's a best practice anyway, and it'll help avoid a whole class of CORS headaches.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Once I've established I definitely need a CORS policy, I typically start with the `WithOrigins()` method. I then expand or restrict the policy further, as need be, to provide cross-origin lock-down of my API, while still allowing the required functionality. CORS can be tricky to work around, but remember, the restrictions are there for your safety.

Cross-origin requests are only one of many potential avenues attackers could use to compromise your app. Many of these are trivial to defend against, but you need to be aware of them, and know how to mitigate them.[91] In the next section, we'll look at common threats and how to avoid them.

18.5 Exploring other attack vectors

So far in this chapter, I've described two potential ways attackers can compromise your apps—XSS and CSRF attacks—and how to prevent them. Both of these vulnerabilities regularly appear on the OWASP top ten list of most critical web app risks,[92] so it's important to be aware of them, and to avoid introducing them into your apps. In this section, I provide an overview of some of the other most common vulnerabilities and how to avoid them in your apps.

18.5.1 Detecting and avoiding open redirect attacks

A common OWASP vulnerability is due to *open redirect* attacks. An open redirect attack is where a user clicks a link to an otherwise safe app and ends up being redirected to a malicious website; for example, one that serves malware. The safe app contains no direct links to the malicious website, so how does this happen?

Open redirect attacks occur where the next page is passed as a parameter to an action method. The most common example is when logging in to an app. Typically, apps remember the page a user is on before redirecting them to a login page by passing the current page as a `returnUrl` query string parameter. After the user logs in, the app redirects the user to the `returnUrl` to carry on where they left off.

Imagine a user is browsing an e-commerce site. They click Buy on a product and are redirected to the login page. The product page they were on is passed as the `returnUrl`, so after they log in, they're redirected to the product page, instead of being dumped back to the home screen.

An open redirect attack takes advantage of this common pattern, as shown in figure 18.13. A malicious attacker creates a login URL where the `returnUrl` is set to the website they want to send the user and convinces the user to click the link to your web app. After the user logs in, a vulnerable app will then redirect the user to the malicious site.

---

[91] For an example of how incorrectly configured CORS policies could expose vulnerabilities in your app, see http://mng.bz/211b.
[92] OWASP publishes the list online, with descriptions of each attack and how to prevent those attacks. You can view the lists dating back to 2003 at http://mng.bz/yXd3, and a cheat sheet for staying safe here: https://cheatsheetseries.owasp.org/.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

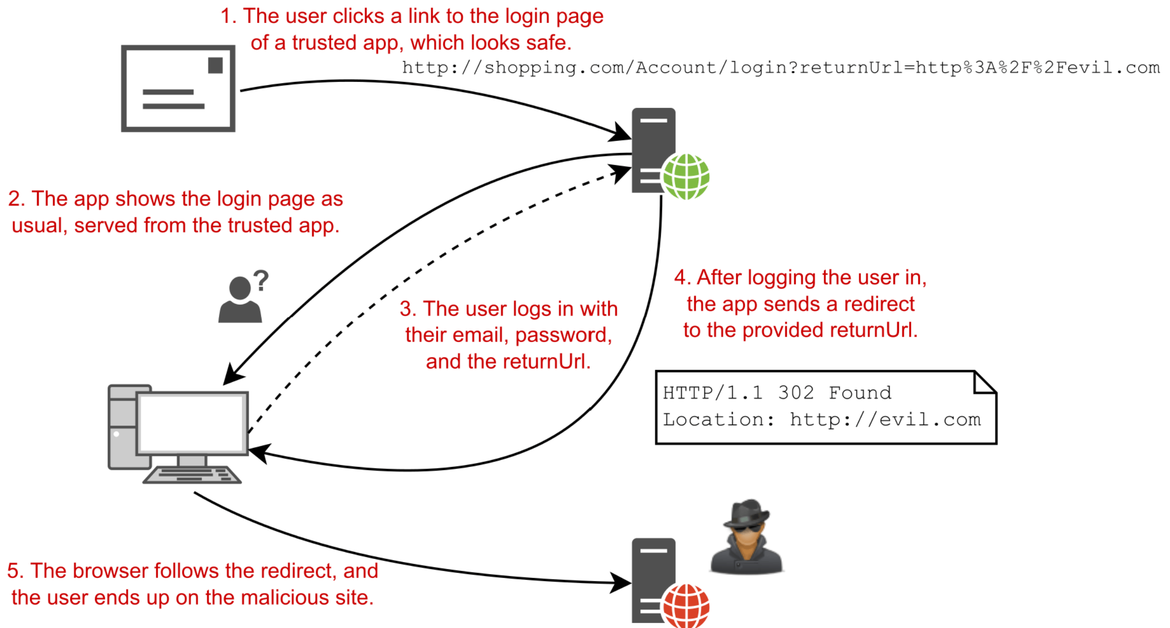


Figure 18.13 An open redirect makes use of the common return URL pattern. This is typically used for login pages but may be used in other areas of your app too. If your app doesn't verify the URL is safe before redirecting the user, it could redirect users to malicious sites.

The simple solution to this attack is to always validate that the `returnUrl` is a local URL that belongs to your app *before* redirecting users to it. The default Identity UI does this already, so you shouldn't have to worry about the login page if you're using Identity, as described in chapter 14.

If you have redirects in other parts of your app, ASP.NET Core provides a couple of helper methods for staying safe, the most useful of which is `Url.IsLocalUrl()`. The following listing shows how you could verify a provided return URL is safe, and if it isn't, redirect to the app's homepage. You can also use the `LocalRedirect()` helper method on the `ControllerBase` and Razor Page `PageModel` classes, which throws an exception if the provided URL isn't local.

Listing 18.8 Detecting open redirect attacks by checking for local return URLs

```
[HttpPost]
public async Task<IActionResult> Login(
    LoginViewModel model, string returnUrl = null)        #A
{
    // Verify password, and sign user in

    if (Url.IsLocalUrl(returnUrl))                        #B
    {
        return Redirect(returnUrl);                       #C
```


Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
    }
    else
    {
        return RedirectToAction("Index", "Home");         #D
    }
}
```

#A The return URL is provided as an argument to the action method.
#B Returns true if the return URL starts with / or ~/
#C The URL is local, so it's safe to redirect to it.
#D The URL was not local, could be an open redirect attack, so redirect to the homepage for safety.

This simple pattern protects against open redirect attacks that could otherwise expose your users to malicious content. Whenever you're redirecting to a URL that comes from a query string or other user input, you should use this pattern.

Open redirect attacks present a risk to your *users* rather than to your app directly. The next vulnerability represents a critical vulnerability in your app itself.

18.5.2 Avoiding SQL injection attacks with EF Core and parameterization

SQL injection attacks represent one of the most dangerous threats to your application. Attackers craft simple malicious input, which they send to your application as traditional form-based input or by customizing URLs and query strings to execute arbitrary code against your database. An SQL injection vulnerability could expose your entire database to attackers, so it's critical that you spot and remove any such vulnerabilities in your apps.

Hopefully, I've scared you a little with that introduction, so now for the good news—if you're using EF Core (or pretty much any other ORM) in a standard way, then you should be safe. EF Core has built-in protections against SQL injection, so as long as you're not doing anything funky, you should be fine.

SQL injection vulnerabilities occur when you build SQL statements yourself and include dynamic input that an attacker provides, even indirectly. EF Core provides the ability to create raw SQL queries using the `FromSqlRaw()` method, so you must be careful when using this method.

Imagine your recipe app has a search form that lets you search for a recipe by name. If you write the query using LINQ extension methods (as discussed in chapter 12), then you would have no risk of SQL injection attacks. However, if you decide to write your SQL query by hand, you open yourself up to such a vulnerability.

Listing 18.9 An SQL injection vulnerability in EF Core due to string concatenation

```
public IList<User> FindRecipe(string search)         #A
{
    return _context.Recipes                          #B
        .FromSqlRaw("SELECT * FROM Recipes" +        #C
                 "WHERE Name = '" + search + "'")    #D
        .ToList();
}
```

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

#A The search parameter comes from user input, so it's unsafe.
#B The current EF Core DbContext is held in the _context field.
#C You can write queries by hand using the FromSqlRaw extension method.
#D This introduces the vulnerability—including unsafe content directly in an SQL string.

In this listing, the user input held in `search` is included *directly* in the SQL query. By crafting malicious input, users can potentially perform any operation on your database. Imagine an attacker searches your website using the text

```
'; DROP TABLE Recipes; --
```

Your app assigns this to the `search` parameter, and the SQL query executed against your database becomes

```
SELECT * FROM Recipes WHERE Name = ''; DROP TABLE Recipes; --'
```

By simply entering text into the search form of your app, the attacker has deleted the entire Recipes table from your app! That's catastrophic, but an SQL injection vulnerability provides more or less unfettered access to your database. Even if you've set up database permissions correctly to prevent this sort of destructive action, attackers will likely be able to read all the data from your database, including your users' details.

The simple way to avoid this happening is to avoid creating SQL queries by hand like this. If you do need to write your own SQL queries, then don't use string concatenation, as you did in listing 18.9. Instead, use parameterized queries, in which the (potentially unsafe) input data is separate from the query itself, as shown here.

Listing 18.10 Avoiding SQL injection by using parameterization

```
public IList<User> FindRecipe(string search)
{
    return _context.Recipes
        .FromSqlRaw("SELECT * FROM Recipes WHERE Name = '{0}'",     #A
                 search)                                          #B
        .ToList();
}
```

#A The SQL query uses a placeholder {0} for the parameter.
#B The dangerous input is passed as a parameter, separate from the query.

Parameterized queries are not vulnerable to SQL injection attacks, so the same attack presented earlier won't work. If you use EF Core (or other ORMs) to access data using standard LINQ queries, you won't be vulnerable to injection attacks. EF Core will automatically create all SQL queries using parameterized queries to protect you.

> **NOTE** I've only talked about SQL injection attacks in terms of a relational database, but this vulnerability can appear in NoSQL and document databases too. Always use parameterized queries (or equivalent) and don't craft queries by concatenating strings with user input.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Injection attacks have been the number one vulnerability on the web for over a decade, so it's crucial that you're aware of them and how they arise. Whenever you need to write raw SQL queries, make sure you always use parameterized queries.

The next vulnerability is also related to attackers accessing data they shouldn't be able to. It's a little subtler than a direct injection attack but is trivial to perform—the only skill the attacker needs is the ability to count.

18.5.3 Preventing insecure direct object references

Insecure direct object reference is a bit of a mouthful, but it means users accessing things they shouldn't by noticing patterns in URLs. Let's revisit our old friend the recipe app. As a reminder, the app shows you a list of recipes. You can view any of them, but you can only edit recipes you created yourself. When you view someone else's recipe, there's no Edit button visible.

For example, a user clicks the edit button on one of their recipes and notices the URL is `/Recipes/Edit/120`. That "120" is a dead giveaway as the underlying database ID of the entity you're editing. A simple attack would be to change that ID to gain access to a *different* entity, one that you wouldn't normally have access to. The user could try entering `/Recipes/Edit/121`. If that lets them edit or view a recipe that they shouldn't be able to, you have an insecure direct object reference vulnerability.

The solution to this problem is simple—you should have resource-based authentication and authorization in your action methods. If a user attempts to access an entity they're not allowed to access, they should get a permission denied error. They shouldn't be able to bypass your authorization by typing a URL directly into the search bar of their browser.

In ASP.NET Core apps, this vulnerability typically arises when you attempt to restrict users by hiding elements from your UI, for example, by hiding the Edit button. Instead, you should use resource-based authorization, as discussed in chapter 15.

> **WARNING** You must always use resource-based authorization to restrict which entities a user can access. Hiding UI elements provides an improved user experience, but it isn't a security measure.

You can sidestep this vulnerability somewhat by avoiding integer IDs for your entities in the URLs; for example, using a pseudorandom GUID (for instance, `C2E296BA-7EA8-4195-9CA7-C323304CCD12`) instead. This makes the process of guessing other entities harder as you can't just add one to an existing number, but it's only masking the problem rather than fixing it. Nevertheless, using GUIDs can be useful when you want to have publicly accessible pages (that don't require authentication), but you don't want their IDs to be easily discoverable.

The final section in this chapter doesn't deal with a single vulnerability. Instead, I discuss a separate, but related, issue: protecting your users' data.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

18.5.4 Protecting your users' passwords and data

For many apps, the most sensitive data you'll be storing is the personal data of your users. This could include emails, passwords, address details, or payment information. You should be careful when storing any of this data. As well as presenting an inviting target for attackers, you may have legal obligations for how you handle it, such as data protection laws and PCI compliance requirements.

The easiest way to protect yourself is to not store data that you don't need. If you don't *need* your user's address, don't ask for it. That way, you can't lose it! Similarly, if you use a third-party identity service to store user details, as described in chapter 14, then you won't have to work as hard to protect your users' personal information.

If you store user details in your own app, or build your own identity provider, then you need to make sure to follow best practices when handling user information. The new project templates that use ASP.NET Core Identity follow most of these practices by default, so I highly recommend you start from one of these. You need to consider many different aspects—too many to go into detail here[93]—but they include:

- Never store user passwords anywhere directly. You should only store cryptographic hashes, computed using an expensive hashing algorithm, such as BCrypt or PBKDF2.
- Don't store more data than you need. You should never store credit card details.
- Allow users to use two-factor authentication (2FA) to sign in to your site.
- Prevent users from using passwords that are known to be weak or compromised.
- Mark authentication cookies as "http" (so they can't be read using JavaScript) and "secure" so they'll only be sent over an HTTPS connection, never over HTTP.
- Don't expose whether a user is already registered with your app or not. Leaking this information can expose you to enumeration attacks.[94]

These are all guidelines, but they represent the minimum you should be doing to protect your users. The most important thing is to be aware of potential security issues as you're building your app. Trying to bolt on security at the end is always harder than thinking about it from the start, so it's best to think about it earlier rather than later.

This chapter has been a whistle-stop tour of things to look out for. We've touched on most of the big names in security vulnerabilities, but I strongly encourage you to check out the other resources mentioned in this chapter. They provide a more exhaustive list of things to consider complementing the defenses mentioned in this chapter. On top of that, don't forget about input validation and mass assignment/over-posting, as discussed in chapter 6. ASP.NET

[93] The NIST (National Institute of Standards and Technology) recently released their Digital Identity Guidelines on how to handle user details: https://pages.nist.gov/800-63-3/sp800-63-3.html.

[94] You can learn more about website enumeration in this video tutorial from Troy Hunt: http://mng.bz/PAAA.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Core includes basic protections against some of the most common attacks, but you can still shoot yourself in the foot. Make sure it's not your app making headlines for being breached!

18.6 Summary

- HTTPS is used to encrypt your app's data as it travels from the server to the browser and back. This prevents third parties from seeing or modifying it.
- HTTPS is virtually mandatory for production apps, as modern browsers like Chrome and Firefox mark non-HTTPS apps as explicitly "not secure."
- In production, you can avoid handling the TLS in your app by using SSL/TLS offloading. This is where a reverse proxy uses HTTPS to talk to the browser, but the traffic is unencrypted between your app and the reverse proxy. The reverse proxy could be on the same or a different server, such as IIS or NGINX, or it could be a third-party service, such as Cloudflare.
- You can use the .NET Core developer certificate or the IIS express developer certificate to enable HTTPS during development. This can't be used for production, but it's sufficient for testing locally. You must run `dotnet dev-certs https --trust` when you first install .NET Core to trust the certificate.
- You can configure an HTTPS certificate for Kestrel in production using the `Kestrel:Certificates:Default` configuration section. This does not require any changes to your application—Kestrel will automatically load the certificate when your app starts and use it to serve HTTPS requests.
- You can use the `HstsMiddleware` to set HTTP Strict Transport Security (HSTS) headers for your application, to ensure the browser sends HTTPS requests to your app instead of HTTP requests. This can only be enforced once an HTTPS request is made to your app, so is best used in conjunction with HTTP to HTTPS redirection.
- You can enforce HTTPS for your whole app using the `HttpsRedirectionMiddleware`. This will redirect HTTP requests to HTTPS endpoints.
- Cross-site scripting (XSS) attacks involve malicious users injecting content into your app, typically to run malicious JavaScript when users browse your app. You can avoid XSS injection attacks by always encoding unsafe input before writing it to a page. Razor Pages do this automatically unless you use the `@Html.Raw()` method, so use it sparingly and carefully.
- Cross-site request forgery (CSRF) attacks are a problem for apps that use cookie-based authentication, such as ASP.NET Core Identity. It relies on the fact that browsers automatically send cookies to a website. A malicious website could create a form that POSTs to your API, and the browser will send the authentication cookie with the request. This allows malicious websites to send requests as though they're the logged-in user.
- You can mitigate CSRF attacks using anti-forgery tokens. These involve writing a hidden field in every form that contains a random string based on the current user. A similar token is stored in a cookie. A legitimate request will have both parts, but a

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

forged request from a malicious website will only have the cookie half; they cannot recreate the hidden field in the form. By validating these tokens, your API can reject forged requests.

- The Razor Pages framework automatically adds anti-forgery tokens to any forms you create using Razor and validates the tokens for inbound requests. You can disable the validation check if necessary, using the `[IgnoreAntiForgeryToken]` attribute.
- Browsers won't allow websites to make JavaScript AJAX requests from one app to others at different origins. To match the origin, the app must have the same scheme, domain, and port. If you wish to make cross-origin requests like this, you must enable cross-origin resource sharing (CORS) in your API.
- CORS uses HTTP headers to communicate with browsers and defines which origins can call your API. In ASP.NET Core, you can define multiple policies, which can be applied either globally to your whole app, or to specific controllers and actions.
- You can add the CORS middleware by calling `UseCors()` in `Startup.Configure` and optionally providing the name of the default CORS policy to apply. You can also apply CORS to a Web API action or controller by adding the `[EnableCors]` attribute and providing the name of the policy to apply.
- Open redirect attacks use the common `returnURL` mechanism after logging in to redirect users to malicious websites. You can prevent this attack by ensuring you only redirect to local URLs, URLs that belong to your app.
- Insecure direct object references are a common problem where you expose the ID of database entities in the URL. You should always verify that users have permission to access or change the requested resource by using resource-based authorization in your action methods.
- SQL injection attacks are a common attack vector when you build SQL requests manually. Always use parameterized queries when building requests, or instead use a framework like EF Core, which isn't vulnerable to SQL injection.
- The most sensitive data in your app is often the data of your users. Mitigate this risk by only storing data that you need. Ensure you only store passwords as a hash, protect against weak or compromised passwords, and provide the option for 2FA. ASP.NET Core Identity provides all of this out of the box, so it's a great choice if you need to create an identity provider.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

19 Building custom components

This chapter covers

- **Building custom middleware**
- **Creating simple endpoints that generate a response using middleware**
- **Using configuration values to set up other configuration providers**
- **Replacing the built-in DI container with a third-party container**

When you're building apps with ASP.NET Core, most of your creativity and specialization goes into the services and models that make up your business logic, and the Razor Pages and controllers that expose them through views or APIs. Eventually, however, you're likely to find that you can't quite achieve a desired feature using the components that come out of the box. At that point, you may need to build a custom component.

This chapter shows how to create some ASP.NET Core components that you're likely to need as your app grows. You probably won't need to use all of them, but each solves a specific problem you may run into.

We start by looking at the middleware pipeline. You saw how to build pipelines by piecing together existing middleware in chapter 3, but in this chapter, you'll create your own custom middleware. You'll explore the basic middleware constructs of the `Map`, `Use`, and `Run` methods, and how to create standalone middleware classes. You'll use these to build middleware components that can add headers to all your responses as well as middleware that return responses.

In section 19.2, you'll see how to use your custom middleware to create simple endpoints using endpoint routing. By using endpoint routing, you can take advantage of the power of the routing and authorization systems that you learned about in chapters 5 and 15, without needing the additional complexity that comes with using API controllers.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Chapter 11 described the configuration provider system used by ASP.NET Core, but in section 19.3, we look at more complex scenarios. In particular, I show you how to handle the situation where a configuration provider itself needs some configuration values. For example, a configuration provider that reads values from a database might need a connection string. You'll also see how to use DI when configuring strongly typed `IOptions` objects, something not possible using the methods you've seen so far.

We stick with DI in section 19.4 where I show how to replace the built-in DI container with a third-party alternative. The built-in container is fine for most small apps, but your `ConfigureServices` function can quickly get bloated as your app grows and you register more services. I show you how to integrate the third-party library, Lamar, into an existing app, so you can make use of extra features such as automatic service registration by convention.

The components and techniques shown in this chapter are common across all ASP.NET Core applications. For example, I use the subject of the first topic—custom middleware—in almost every project I build. In chapter 20 we look at some additional components that are specific to Razor Pages and API controllers.

19.1 Customizing your middleware pipeline

In this section you'll learn how to create custom middleware. You'll learn how to use the `Map`, `Run`, and `Use` extension methods to create simple middleware using lambda expressions. You'll then see how to create equivalent middleware components using dedicated classes. You'll also learn how to split the middleware pipeline into branches, and find out when this is useful.

The middleware pipeline is one of the fundamental building blocks of ASP.NET Core apps, so we covered it in depth in chapter 3. Every request passes through the middleware pipeline, and each middleware component in turn gets an opportunity to modify the request, or to handle it and return a response.

ASP.NET Core includes middleware out of the box for handling common scenarios. You'll find middleware for serving static files, for handling errors, for authentication, and many more. However, you'll spend most of your time during development working with Razor Pages and Web API controllers. These are exposed as the endpoints for most of your app's business logic and call methods on your app's various business services and models.

Sometimes, however, you don't need all the power (and associated complexity) that comes with Razor Pages and API controllers. You might want to create a very simple app that, when called, returns the current time. Or you might want to add a health-check URL to an existing app, where calling the URL doesn't do any significant processing, but checks that the app is running. Although you *could* use API controllers for these, you could also create small, dedicated middleware components to handle these requirements.

Other times, you might have requirements that lie outside the remit of Razor Pages and API controllers. For example, you might want to ensure all responses generated by your app include a specific header. This sort of cross-cutting concern is a perfect fit for custom middleware. You could add the custom middleware early in your middleware pipeline to ensure

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

that every response from your app includes the required header, whether it comes from the static file middleware, the error-handling middleware, or a Razor Page.

In this section, I show three ways to create custom middleware components, as well as how to create branches in your middleware pipeline where a request can flow down either one branch or another. By combining the methods demonstrated in this section, you'll be able to create custom solutions to handle your specific requirements.

We'll start by creating a middleware component that returns the current time as plain text, whenever the app receives a request. From there, we'll look at branching the pipeline, creating general-purpose middleware components, and finally, how to encapsulate your middleware into standalone classes. In section 19.2. you'll see an alternative approach to exposing response-generating middleware using endpoint routing.

19.1.1 Creating simple endpoints with the Run extension

As you've seen in previous chapters, you define the middleware pipeline for your app in the `Configure` method of your `Startup` class. You add middleware to a provided `IApplicationBuilder` object, typically using extension methods. For example:

```
public void Configure(IApplicationBuilder)
{
    app.UseDeveloperExceptionPage();
    app.UseStaticFiles();
}
```

When your app receives a request, the request passes through each middleware, which gets a chance to modify the request, or to handle it by generating a response. If a middleware component generates a response, it effectively short-circuits the pipeline; no subsequent middleware in the pipeline will see the request. The response passes back through the earlier middleware components on its way back to the browser.

You can use the `Run` extension method to build a simple middleware component that always generates a response. This extension takes a single lambda function that runs whenever a request reaches the component. The `Run` extension always generates a response, so no middleware placed after it will ever execute. For that reason, you should always place the `Run` middleware last in a middleware pipeline.

> **TIP** Remember, middleware runs in the order you add them to the pipeline. If a middleware handles a request and generates a response, later middleware will never see the request.

The `Run` extension method provides access to the request in the form of the `HttpContext` object you saw in chapter 3. This contains all the details of the request via the `Request` property, such as the URL path, the headers, and the body of the request. It also contains a `Response` property you can use to return a response.

The following listing shows how you could build a simple middleware that returns the current time. It uses the provided `HttpContext` context object and the `Response` property to

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

set the `Content-Type` header of the response and writes the body of the response using `WriteAsync(text)`.

Listing 19.1 Creating simple middleware using the `Run` extension

```
public void Configure(IApplicationBuilder app)
{
    app.Run(async (HttpContext context) =>             #A
    {
        context.Response.ContentType = "text/plain";     #B
        await context.Response.WriteAsync(              #C
            DateTime.UtcNow.ToString());                #C
    });

    app.UseStaticFiles();                               #D
}
```

#A Uses the Run extension to create a simple middleware that always returns a response
#B You should set the content-type of the response you're generating.
#C Returns the time as a string in the response. The 200 OK status code is used if not explicitly set.
#D Any middleware added after the Run extension will never execute.

The `Run` extension is useful for building simple middleware. You can use it to create very basic endpoints that always generate a response. But as the component always generates some sort of response, you must always place it at the end of the pipeline, as no middleware placed after it will execute.

A more common scenario is where you want your middleware component to only respond to a specific URL path. In the next section, you'll see how you can combine `Run` with the `Map` extension method to create simple branching middleware pipelines.

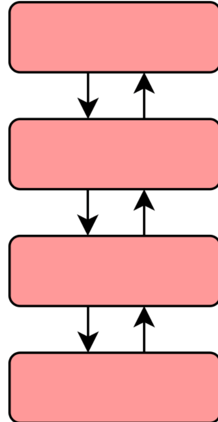
19.1.2 Branching middleware pipelines with the Map extension

So far, when discussing the middleware pipeline, we've always considered it as a single pipeline of sequential components. Each request passes through every middleware, until a component generates a response, which passes back through the previous middleware.

The `Map` extension method lets you change that simple pipeline into a branching structure. Each branch of the pipeline is independent; a request passes through one branch or the other, but not both, as shown in figure 19.1.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>



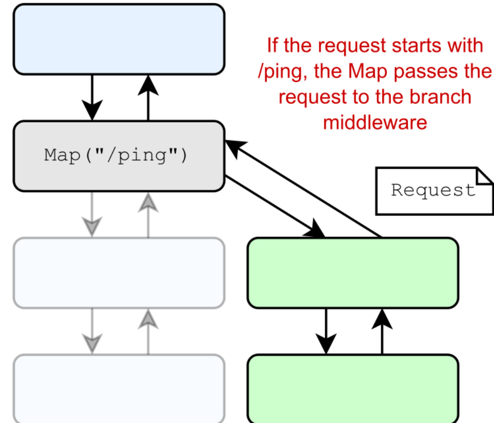


Figure 19.1 A sequential middleware pipeline compared to a branching pipeline created with the `Map` extension. In branching middleware, requests only pass through one of the branches at most. Middleware on the other branch never see the request and aren't executed.

The `Map` extension method looks at the path of the request's URL; if the path matches the required pattern, the request travels down the branch of the pipeline, otherwise it remains on the main trunk. This lets you have completely different behavior in different branches of your middleware pipeline.

> **NOTE** The URL matching used by `Map` is conceptually similar to the routing you've seen since chapter 6, but it is much more basic, with many limitations. For example, it uses a simple string-prefix match, and you can't use route parameters. Generally you should favor creating *endpoints* instead of branching using `Map`, as you'll see in section 19.2.

For example, imagine you want to add a simple health-check endpoint to your existing app. This endpoint is a simple URL you can call that indicates whether your app is running correctly. You could easily create a health-check middleware using the `Run` extension, as you saw in listing 19.1, but then that's *all* your app can do. You only want the health-check to respond to a specific URL, `/ping`; your Razor Pages should handle all other requests as normal.

> **TIP** The health-check scenario is a simple example to demonstrate the `Map` method, but ASP.NET Core includes built-in support for health-check endpoints which you should use instead of creating your own. You

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

can learn more about creating health checks at https://docs.microsoft.com/aspnet/core/host-and-deploy/health-checks.

One solution to this would be to create a branch using the `Map` extension method and to place the health-check middleware on that branch, as shown next. Only those requests that match the `Map` pattern `/ping` will execute the branch, all other requests will be handled by the standard routing middleware and Razor Pages on the main trunk instead.

Listing 19.2 Using the `Map` extension to create branching middleware pipelines

```
public void Configure(IApplicationBuilder app)
{
    app.UseDeveloperExceptionPage();                        #A

    app.Map("/ping", (IApplicationBuilder branch) =>        #B
    {
        branch.UseExceptionHandler();                       #C

        branch.Run(async (HttpContext context) =>           #D
        {                                                   #D
            context.Response.ContentType = "text/plain";    #D
            await context.Response.WriteAsync("pong");      #D
        });                                                 #D
    });

    app.UseStaticFiles();                                   #E
    app.UseRouting();                                       #E
    app.UseEndpoints(endpoints =>                           #E
    {                                                       #E
        endpoints.MapRazorPages();                          #E
    });                                                     #E
}
```

#A Every request will pass though this middleware.
#B The Map extension method will branch if a request starts with /ping.
#C This middleware will only run for requests matching the /ping branch.
#D The Run extension always returns a response, but only on the /ping branch.
#E The rest of the middleware pipeline will run for requests that don't match the /ping branch.

The `Map` middleware creates a completely new `IApplicationBuilder` (called `branch` in the listing), which you can customize as you would your main `app` pipeline. Middleware added to the `branch` builder are only added to the branch pipeline, not the main trunk pipeline.

In this example, you add the `Run` middleware to the branch, so it will only execute for requests that start with `/ping`, such as `/ping`, `/ping/go`, or `/ping?id=123`. Any requests that don't start with `/ping` are ignored by the `Map` extension. Those requests stay on the main trunk pipeline and execute the next middleware in the pipeline after `Map` (in this case, the `StaticFilesMiddleware`).

If you need to, you can create sprawling branched pipelines using `Map`, where each branch is independent of every other. You could also nest calls to `Map`, so you have branches coming off branches.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

The `Map` extension can be useful, but if you try to get too elaborate, it can quickly get confusing. Remember, you should use middleware for implementing cross-cutting concerns or very simple endpoints. The endpoint routing mechanism of controllers and Razor Pages is better suited to more complex routing requirements, so don't be afraid to use it.

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

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.

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.

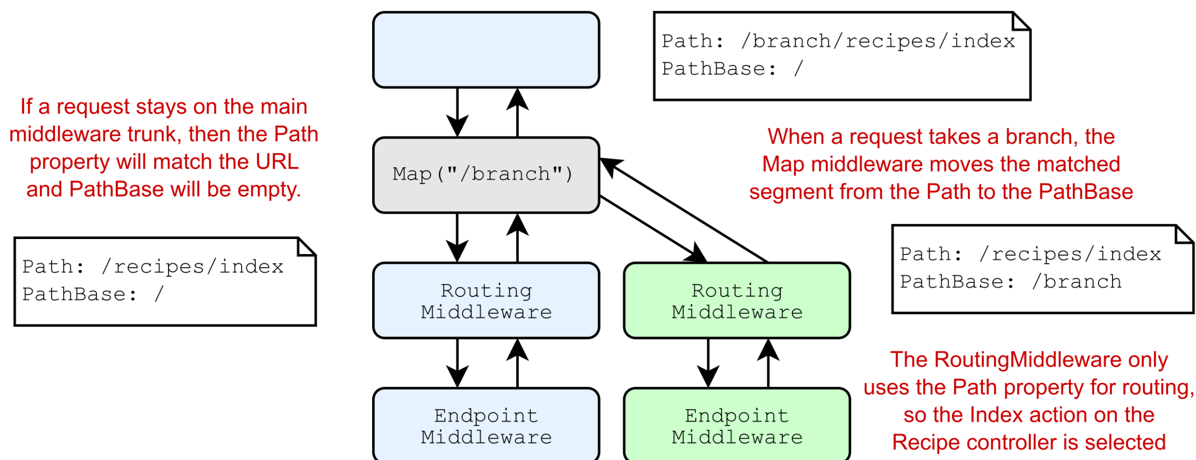


Figure 19.2 When the `Map` extension diverts a request to a branch, it removes the matched segment from the `Path` property and adds it to the `PathBase` property.

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.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

19.1.3 Adding to the pipeline with the Use extension

You can use the `Use` extension method to add a general-purpose piece of middleware. You can use it to view and modify requests as they arrive, to generate a response, or to pass the request on to subsequent middleware in the pipeline.

Similar to the `Run` extension, when you add the `Use` extension to your pipeline, you specify a lambda function that runs when a request reaches the middleware. The app passes two parameters to this function:

- The `HttpContext` representing the current request and response. You can use this to inspect the request or generate a response, as you saw with the `Run` extension.
- A pointer to the rest of the pipeline as a `Func<Task>`. By executing this task, you can execute the rest of the middleware pipeline.

By providing a pointer to the rest of the pipeline, you can use the `Use` extension to control exactly how and when the rest of the pipeline executes, as shown in figure 19.3. If you don't call the provided `Func<Task>` at all, then the rest of the pipeline doesn't execute for the request, so you have complete control.

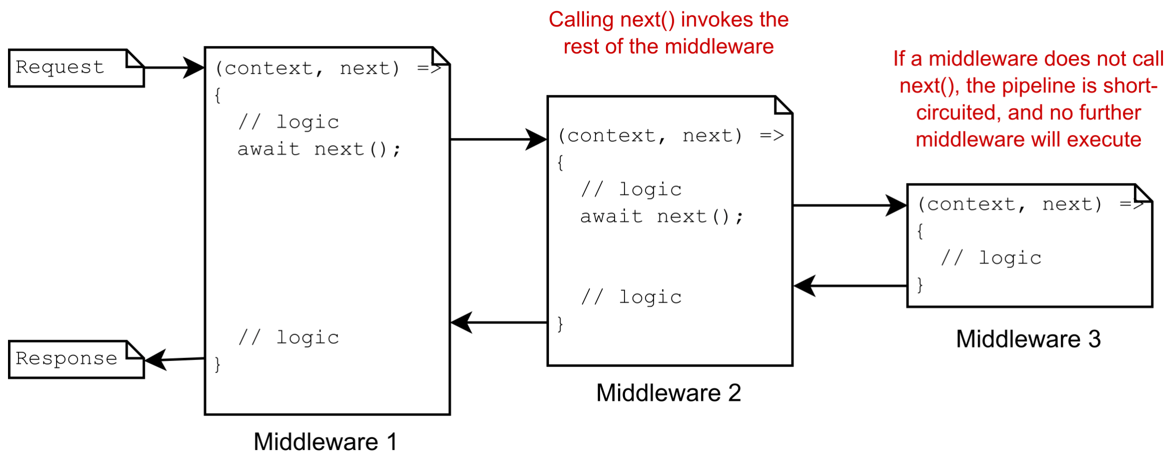


Figure 19.3 Three pieces of middleware, created with the `Use` extension. Invoking the provided `Func<Task>` using `next()` invokes the rest of the pipeline. Each middleware can run code before and after calling the rest of the pipeline, or it can choose to not call `next()` at all to short-circuit the pipeline.

Exposing the rest of the pipeline as a `Func<Task>` makes it easy to conditionally short-circuit the pipeline, which opens up many different scenarios. Instead of branching the pipeline to implement the health-check middleware with `Map` and `Run`, as you did in listing 19.2, you could use a single instance of the `Use` extension. This provides the same required functionality as before but does so without branching the pipeline.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Listing 19.3 Using the `Use` extension method to create a health-check middleware

```
public void Configure(IApplicationBuilder app)
{
    app.Use(async (HttpContext context, Func<Task> next) =>      #A
    {
        if (context.Request.Path.StartsWithSegments("/ping"))     #B
        {
            context.Response.ContentType = "text/plain";          #C
            await context.Response.WriteAsync("pong");            #C
        }
        else
        {
            await next();                                         #D
        }
    });

    app.UseStaticFiles();
}
```

#A The Use extension takes a lambda with HttpContext (context) and Func<Task> (next) parameters.
#B The StartsWithSegments method looks for the provided segment in the current path.
#C If the path matches, generate a response, and short-circuit the pipeline
#D If the path doesn't match, call the next middleware in the pipeline, in this case UseStaticFiles().

If the incoming request starts with the required path segment (`/ping`), then the middleware responds and doesn't call the rest of the pipeline. If the incoming request doesn't start with `/ping`, then the extension calls the next middleware in the pipeline, no branching necessary.

With the `Use` extension, you have control over when, and if, you call the rest of the middleware pipeline. But it's important to note that you generally shouldn't modify the `Response` object after calling `next()`. Calling `next()` runs the rest of the middleware pipeline, so a subsequent middleware may start streaming the response to the browser. If you try to modify the response *after* executing the pipeline, you may end up corrupting the response or sending invalid data.

> **WARNING** **Don't *modify* the `Response` object after calling `next()`. Also, don't call `next()` if you've written to the `Response.Body`: writing to this `Stream` can trigger Kestrel to start streaming the response to the browser and you could cause invalid data to be sent. You can generally *read* from the `Response` object safely, to inspect the final `StatusCode` or `ContentType` of the response, for example.**

Another common use for the `Use` extension method is to modify every request or response that passes through it. For example there are various HTTP headers that you should send with all your applications for security reasons. These headers often disable old, insecure, legacy

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

behaviors by browsers, or restrict the features enabled by the browser. You learned about the HSTS header in chapter 18, but there are other headers you can add for additional security.[95]

Imagine you've been tasked with adding one such header, `X-Content-Type-Options: nosniff` (which provides added protection against XSS attacks), to every response generated by your app. This sort of cross-cutting concern is perfect for middleware. You can use the `Use` extension method to intercept every request, set the response header, and then execute the rest of the middleware pipeline. No matter what response the pipeline generates, whether it's a static file, an error, or a Razor Page, the response will always have the security header.

Listing 19.4 shows a robust way to achieve this. When the middleware receives a request, it registers a callback that runs before Kestrel starts sending the response back to the browser. It then calls `next()` to run the rest of the middleware pipeline. When the pipeline generates a response, likely in some later middleware, Kestrel executes the callback and adds the header. This approach ensures the header isn't accidentally removed by other middleware in the pipeline and also that you don't try to modify the headers after the response has started streaming to the browser.

Listing 19.4 Adding headers to a response with the `Use` extension

```
public void Configure(IApplicationBuilder app)
{
    app.Use(async (HttpContext context, Func<Task> next) =>         #A
    {
        context.Response.OnStarting(() =>                           #B
        {
            context.Response.Headers["X-Content-Type-Options"] =    #C
                "nosniff";                                          #C
            return Task.CompletedTask;                              #D
        });
        await next();                                               #E
    }

    app.UseStaticFiles();                                           #F
    app.UseRouting();                                               #F
    app.UseEndpoints(endpoints =>                                   #F
    {                                                               #F
        endpoints.MapControllers();                                 #F
    });                                                             #F
}
```

#A Adds the middleware at the start of the pipeline
#B Sets a function that should be called before the response is sent to the browser
#C Adds an HSTS header. For 60 seconds the browser will only send HTTPS requests to your app
#D The function passed to OnStarting must return a Task
#E Invokes the rest of the middleware pipeline
#F No matter what response is generated, it'll have the security header added

[95] You can test the security headers for your app using https://securityheaders.com/, which also provides information about what headers you should add to your application and why.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Simple cross-cutting middleware like the security header example are common, but they can quickly clutter your `Configure` method, and make it difficult to understand the pipeline at a glance. Instead, it's common to encapsulate your middleware into a class that's functionally equivalent to the `Use` extension, but which can be easily tested and reused.

19.1.4 Building a custom middleware component

Creating middleware with the `Use` extension, as you did in listings 19.3 and 19.4, is convenient but it's not easy to test, and you're somewhat limited in what you can do. For example, you can't easily use DI to inject scoped services inside of these basic middleware components. Normally, rather than calling the `Use` extension directly, you'll encapsulate your middleware into a class that's functionally equivalent.

Custom middleware components don't have to derive from a specific base class or implement an interface, but they have a certain shape, as shown in listing 19.5. ASP.NET Core uses reflection to execute the method at runtime. Middleware classes should have a constructor that takes a `RequestDelegate` object, which represents the rest of the middleware pipeline, and they should have an `Invoke` function with a signature similar to

```
public Task Invoke(HttpContext context);
```

The `Invoke()` function is equivalent to the lambda function from the `Use` extension, and is called when a request is received. Here's how you could convert the headers middleware from listing 19.4 into a standalone middleware class.[96]

Listing 19.5 Adding headers to a `Response` using a custom middleware component

```
public class HeadersMiddleware
{
    private readonly RequestDelegate _next;          #A
    public HeadersMiddleware(RequestDelegate next)   #A
    {                                                #A
        _next = next;                                #A
    }                                                #A

    public async Task Invoke(HttpContext context)    #B
    {
        context.Response.OnStarting(() =>                          #C
        {                                                          #C
            context.Response.Headers["X-Content-Type-Options"] =   #C
                "nosniff";                                         #C
            return Task.CompletedTask;                             #C
        });                                                        #C

        await _next(context);        #D
```

[96] Using this "shape" approach makes the middleware more flexible. In particular, it means you can easily use DI to inject services into the `Invoke` method. This wouldn't be possible if the `Invoke` method were an overridden base class method or an interface. However, if you prefer, you can implement the `IMiddleware` interface, which defines the basic Invoke method.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
    }
}
```

#A The RequestDelegate represents the rest of the middleware pipeline
#B The Invoke method is called with HttpContext when a request is received
#C Adds the HSTS header response as before
#D Invokes the rest of the middleware pipeline. Note that you must pass in the provided HttpContext

This middleware is effectively identical to the example in listing 19.4 but encapsulated in a class called `HeadersMiddleware`. You can add this middleware to your app in `Startup.Configure` by calling

```
app.UseMiddleware<HeadersMiddleware>();
```

A common pattern is to create helper extension methods to make it easy to consume your extension method from `Startup.Configure` (so that IntelliSense reveals it as an option on the `IApplicationBuilder` instance). Here's how you could create a simple extension method for `HeadersMiddleware`.

Listing 19.6 Creating an extension method to expose `HeadersMiddleware`

```
public static class MiddlewareExtensions
{
    public static IApplicationBuilder UseSecurityHeaders(     #A
        this IApplicationBuilder app)                         #A
    {
        return app.UseMiddleware<HeadersMiddleware>();        #B
    }
}
```

#A By convention, the extension method should return an IApplicationBuilder to allow chaining
#B Adds the middleware to the pipeline

With this extension method, you can now add the headers middleware to your app using

```
app.UseSecurityHeaders();
```

> **TIP** There is a NuGet package available that makes it easy to add security headers using middleware, without having to write your own. The package provides a fluent interface for adding the recommend security headers to your app. You can find instructions on how to install it at https://github.com/andrewlock/NetEscapades.AspNetCore.SecurityHeaders.

Listing 19.5 is a simple example, but you can create middleware for many different purposes. In some cases, you may need to use DI to inject services and use them to handle a request. You can inject singleton services into the constructor of your middleware component, or you can inject services with any lifetime into the `Invoke` method of your middleware, as demonstrated in the following listing.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Listing 19.7 Using DI in middleware components

```
public class ExampleMiddleware
{
    private readonly RequestDelegate _next;
    private readonly ServiceA _a;                              #A
    public HeadersMiddleware(RequestDelegate next, ServiceA a)    #A
    {
        _next = next;
        _a = a;                                                #A
    }

    public async Task Invoke(
        HttpContext context, ServiceB b, ServiceC c)             #B
    {
        // use services a, b, and c
        // and/or call _next.Invoke(context);
    }
}
```

#A You can inject additional services in the constructor. These must be singletons
#B You can inject services into the Invoke method. These may have any lifetime

> **WARNING** ASP.NET Core creates the middleware only once for the lifetime of your app, so any dependencies injected in the constructor must be singletons. If you need to use scoped or transient dependencies, inject them into the `Invoke` method.

That covers pretty much everything you need to start building your own middleware components. By encapsulating your middleware into custom classes, you can easily test their behavior, or distribute them in NuGet packages, so I strongly recommend taking this approach. Apart from anything else, it will make your `Startup.Configure()` method less cluttered and easier to understand.

19.2 Creating custom endpoints with endpoint routing

In this section you'll learn how to create custom endpoints from your middleware using endpoint routing. We take the simple middleware branches used in section 19.1 and convert them to use endpoint routing, and demonstrate the additional features this enables, such as routing and authorization.

In section 19.1 I described creating a simple "endpoint" using the `Map` and `Run` extension methods, that returns a plain-text `pong` response whenever a `/ping` request is received, by branching the middleware pipeline. This is fine because it's so simple, but what if you have more complex requirements?

Consider a basic enhancement of the ping-pong example: how would you add authorization to the request? The `AuthorizationMiddleware` added to your pipeline by `UseAuthorization()` looks for metadata on endpoints like Razor Pages to see if there's an `[Authorize]` attribute, but it doesn't know how to work with your ping-pong `Map` extension.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Similarly, what if you wanted to use more complex routing? Maybe you want to be able to call `/ping/3` and have your ping-pong middleware reply `pong-pong-pong` (no, I can't think why you would either!). You now have to try and parse that integer from the URL, make sure it's valid and so on. That's sounding like a lot more work!

When your requirements start ramping up like this, one option is to move to using Web API controllers or Razor Pages. These provide the greatest flexibility in your app and have the most features, but they're also comparatively heavy weight compared to middleware. What if you want something in-between?

In ASP.NET Core 3.0, the routing system was re-written to use *endpoint routing*, to provide exactly this balance. Endpoint routing allows you to create endpoints that can use the same routing and authorization framework as you get with Web API controllers and Razor Pages, but with the simplicity of middleware.

REMINDER I discussed endpoint routing in detail in chapter 5.

In this section you'll see how to convert the simple branch-based middleware from the previous section to a custom endpoint. You'll see how taking this approach makes it easy to apply authorization to the endpoint, using the declarative approaches you're already familiar with from chapter 15.

19.2.1 Creating a custom endpoint routing component

As I described in chapter 5, endpoint routing splits the process of executing an endpoint into two steps, implemented by two separate pieces of middleware:

1. `RoutingMiddleware`. Uses the incoming request to *select* an endpoint to execute. Exposes the metadata about the selected endpoint on `HttpContext`, such as authorization requirements applied using the `[Authorize]` attribute.
2. `EndpointMiddleware`. *Executes* the selected endpoint to generate a response.

The advantage of using a two-step process is that you can place middleware *between* the middleware that selects the endpoint and the middleware that executes it to generate a response. For example, the `AuthorizationMiddleware` uses the selected endpoint to determine whether to short-circuit the pipeline, *before* the endpoint is executed.

Let's imagine that you need to apply authorization to the simple ping-pong endpoint you created in section 19.1.2. This is much easier to achieve with endpoint routing than using simple middleware branches like `Map` or `Use`. The first step is to create a middleware component for the functionality, using the approach you saw in section 19.1.4, as shown in the following listing.

Listing 19.8 The `PingPongMiddleware` implemented as a middleware component

```
public class PingPongMiddleware
{
    public PingPongMiddleware(RequestDelegate next)         #A
```


Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
    {
    }

    public async Task Invoke(HttpContext context)          #B
    {
        context.Response.ContentType = "text/plain";       #C
        await context.Response.WriteAsync("pong");         #C
    }
}
```

#A Even though it isn't used in this case, you must inject a RequestDelegate in the constructor
#B Invoke is called to execute the middleware
#C The middleware always returns a "pong" response

Note that this middleware always returns a `"pong"` response, regardless of the request URL—we will configure the `"/ping"` path later. We can use this class to convert a middleware pipeline from the "branching" version shown in figure 19.1, to the "endpoint" version shown in figure 19.4.

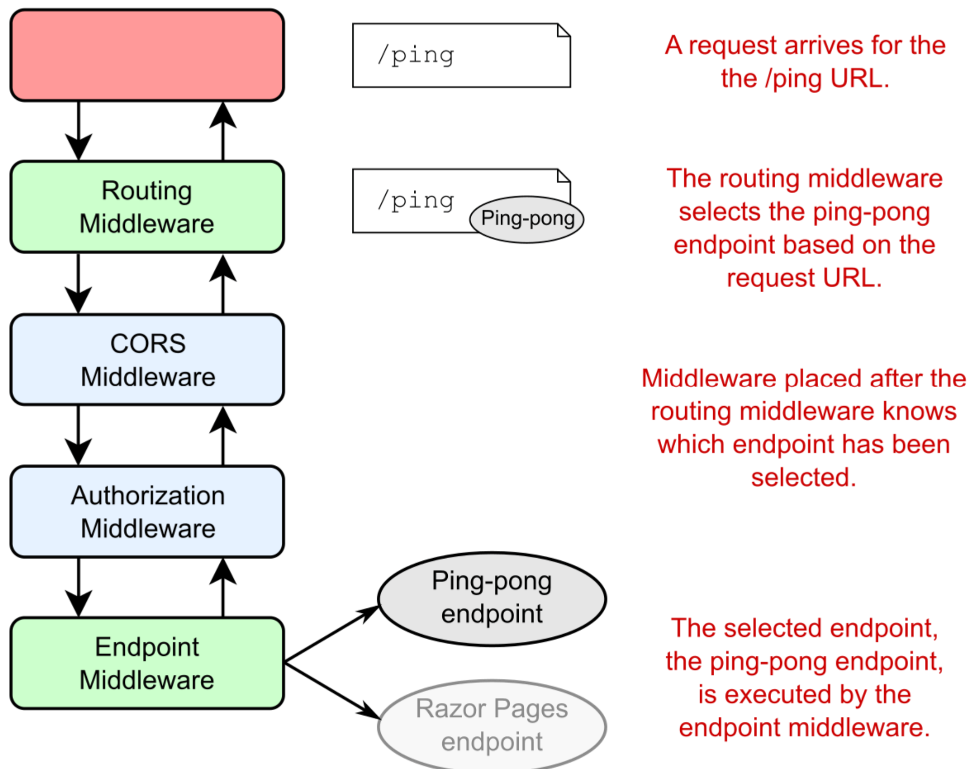


Figure 19.4. Endpoint routing separates the selection of an endpoint from the execution of an endpoint. The routing middleware selects an endpoint based on the incoming request and exposes metadata about the

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

endpoint. Middleware placed before the endpoint middleware can act based on the selected endpoint, such as short-circuiting unauthorized requests. If the request is authorized, the endpoint middleware executes the selected endpoint and generates a response.

Converting the ping-pong middleware to an endpoint doesn't require any changes to the middleware itself. Instead, you need to create a "mini" middleware pipeline, containing your ping-pong middleware only.

> **TIP** Converting a response-generating middleware to an endpoint essentially requires converting it into its own mini-pipeline, so you can include additional middleware in the "endpoint pipeline" if you wish.

You must create your endpoint pipeline *inside* the `UseEndpoints()` lambda argument as shown in the following listing. Use `CreateApplicationBuilder()` to create a new `IApplicationBuilder`, add your middleware that makes up your endpoint, and then call `Build()` to create the pipeline.

Listing 19.9 Mapping the ping-pong endpoint in UseEndpoints

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseRouting();

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

    app.UseEndpoints(endpoints =>
    {
        var endpoint = endpoints                          #A
            .CreateApplicationBuilder()                   #A
            .UseMiddleware<PingPongMiddleware>()          #B
            .Build();                                     #B

        endpoints.Map("/ping", endpoint);                 #C
        endpoints.MapRazorPages();
        endpoints.MapHealthChecks("/healthz");
    });
}}
```

#A Create a miniature, standalone, IApplicationBuilder to build your endpoint
#B Add the middleware and build the final endpoint. This is executed when the endpoint is executed
#C Add the new endpoint to the endpoint collection associated with the route template "/ping"

Once you have a pipeline, you can associate it with a given route by calling `Map()` on the `IEndpointRouteBuilder` instance, and passing in a route template.

> **TIP** Note that the `Map()` function on `IEndpointRouteBuilder` creates a new endpoint (consisting of your mini-pipeline) with an associated route. Although it has the same name, this is conceptually different to the `Map` function on `IApplicationBuilder` from section 19.1.2 which is used to *branch* the middleware pipeline.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

If you have many custom endpoints, the `UseEndpoints()` method can quickly get cluttered. I like to extract this functionality into an extension method, to make the `UseEndpoints()` method cleaner and easier to read. The following listing extracts the code to create an endpoint from listing 19.9 into a separate class, taking the route template to use as a method parameter.

Listing 19.10 An extension method for using the PingPongMiddleware as an endpoint

```
public static class EndpointRouteBuilderExtensions
{
    public static IEndpointConventionBuilder MapPingPong(         #A
        this IEndpointRouteBuilder endpoints,                     #A
        string route)                                             #B
    {
        var pipeline = endpoints.CreateApplicationBuilder()       #C
            .UseMiddleware<PingPongMiddleware>()                  #C
            .Build();                                             #C

        return endpoints                                  #D
            .Map(route, pipeline)                         #D
            .WithDisplayName("Ping-pong");                #E
    }
}
```

#A Create an extension method for registering the PingPongMiddleware as an endpoint
#B Allows the caller to pass in a route template for the endpoint
#C Create the endpoint pipeline
#D Add the new endpoint to the provided endpoint collection, using the provide route template
#E You can add additional metadata here directly, or the caller can add metadata themselves

Now that you have an extension method, `MapPingPong()`, you can update your `UseEndpoints()` method in `Startup.Configure()` to be simpler and easier to understand.

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapPingPong("/ping");
    endpoints.MapRazorPages();
    endpoints.MapHealthChecks("/healthz");
});
```

Congratulations, you've created your first custom endpoint! You haven't added any additional *functionality* yet, but by using the endpoint routing system it's now much easier to satisfy the additional authorization requirements, as you'll see in section 19.2.2.

> **TIP** This example used a very basic route template, "`/ping`", but you can also use templates that contain route parameters, for example "`/ping/{count}`", using the same routing framework you learned in chapter 5. For examples of how to access this data from your middleware, as well as best practice advice, see https://andrewlock.net/accessing-route-values-in-endpoint-middleware-in-aspnetcore-3/.

Converting a branching middleware to use endpoint routing can be useful for taming a middleware pipeline with lots of branches, but you won't necessarily always want to use it.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Using simple branches can be faster than using the routing infrastructure, so in some cases it may be best to avoid endpoint routing.

A good example of this trade-off is the built-in `StaticFileMiddleware`. This middleware serves static files based on the request's URL, but it *doesn't* use endpoint routing due to the performance impact of adding many (potentially hundreds) of routes for each static file in your application. The downside to that choice is that adding authorization to static files is not easy to achieve: if endpoint routing were used, adding authorization would be simple.

19.2.2 Applying authorization to endpoints

One of the main advantages of endpoint routing is the ability to easily apply authorization to your endpoint. For Razor Pages and API controllers, this is achieved by adding the `[Authorize]` attribute, as you saw in chapter 15.

For other endpoints, such as the ping-pong endpoint you created in section 19.2.1, you can apply authorization declaratively when you add the endpoint to your application, by calling `RequireAuthorization()` on the `IEndpointConventionBuilder`, as shown in the following listing.

Listing 19.11 Applying authorization to an endpoint using `RequireAuthorization()`

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseRouting();

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

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapPingPong("/ping")                      #A
            .RequireAuthorization();                        #A
        endpoints.MapRazorPages();
        endpoints.MapHealthChecks("/healthz")               #B
            .RequireAuthorization("HealthCheckPolicy")      #B
    });
}}
```

#A Require authorization. This is equivalent to applying the [Authorize] attribute.
#B Require authorization using a specific policy, HealthCheckPolicy.

Listing 19.11 shows two examples of applying authorization to endpoints

- `RequireAuthorization()`. If you don't provide a method argument, this applies the default authorization policy to the endpoint. It is equivalent to applying the `[Authorize]` attribute to a Razor Page or API controller endpoint.
- `RequireAuthorization(policy)`. If you provide a policy name, the chosen authorization policy will be used. The policy must be configured in `ConfigureServices`, as you saw in chapter 15. This is equivalent to applying `[Authorize("HealthCheckPolicy")]` to a Razor Page or API controller endpoint.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

If you are globally applying authorization to your application (as described in chapter 15), then you can "punch a hole" in the global policy with the complimentary `AllowAnonymous()` method, for example:

```
endpoints.MapPingPong("/ping").AllowAnonymous();
```

This is equivalent to using the `[AllowAnonymous]` attribute on your Razor Pages and actions.

> **NOTE** The `AllowAnonymous()` method for endpoints is new in .NET 5.

Authorization is the canonical example of adding metadata to endpoints to add functionality, but there are other options available too. Out of the box you can use the following methods:

- `RequireAuthorization()`. Applies authorization policies to the endpoint, as you've already seen.
- `AllowAnonymous()`. Overrides a global authorization policy to allow anonymous access to an endpoint
- `RequireCors(policy)`. Apply a CORS policy to the endpoint, as described in chapter 18.
- `RequireHost(hosts)`. Only allow routing to the endpoint if the incoming request matches one of the provided hostnames.
- `WithDisplayName(name)`. Sets the friendly name for the endpoint. Used primarily in logging to describe the endpoint.
- `WithMetadata(items)`. Add arbitrary values as metadata to the endpoint. You can access these values in middleware after an endpoint is selected by the routing middleware.

These features allow various functionality, such as CORS and authorization, to work seamlessly across Razor Pages, API controllers, built in endpoints like the health check endpoints, and custom endpoints like your ping-pong middleware. They should allow you to satisfy most requirements you get around custom endpoints. And if you find you need something more complex, like model-binding for example, then you can always fall back to using API controllers instead. The choice is yours!

In the next section we move away from the middleware pipeline and look at how to handle complex configuration requirements. In particular, you'll see how to set up complex configuration providers that require their own configuration values, and how to use DI services to build your strongly typed `IOptions` objects.

19.3 Handling complex configuration requirements

In this section I describe how to handle two complex configuration requirements: configuration providers that need configuring themselves; and using services to configure `IOptions` objects. In the first scenario, you will see how to partially build your configuration to allow building the

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

provider. In the second scenario, you will see how to use the `IConfigureOptions` interface to allow accessing services when configuring your options objects.

In chapter 11, we discussed the ASP.NET Core configuration system in depth. You saw how an `IConfiguration` object is built from multiple layers, where subsequent layers can add to or replace configuration values from previous layers. Each layer is added by a configuration provider, which can read values from a file, from environment variables, from User Secrets, or from any number of possible locations.

You also saw how to *bind* configuration values to strongly typed POCO objects, using the `IOptions` interface. You can inject these strongly typed objects into your classes to provide easy access to your settings, instead of having to read configuration using `string` keys.

In this section, we'll look at two scenarios that are slightly more complex. In the first scenario, you're trying to use a configuration provider that requires some configuration itself.

As an example, imagine you want to store some configuration in a database table. In order to load these values, you'd need some sort of connection string, which would most likely also come from your `IConfiguration`. You're stuck with a chicken-and-egg situation: you need to build the `IConfiguration` object to add the provider, but you can't add the provider without building the `IConfiguration` object!

In the second scenario, you want to configure a strongly typed `IOptions` object with values returned from a service. But the service won't be available until after you've configured all of the `IOptions` objects. In section 19.3.2, you'll see how to handle this by implementing the `IConfigureOptions` interface.

19.3.1 Partially building configuration to configure additional providers

ASP.NET Core includes many configuration providers, such as file and environment variable providers, that don't require anything more than basic details to set up. All you need to read a JSON file, for example, is the path to that file.

But the configuration system is highly extensible, and more complex configuration providers may require some degree of configuration themselves. For example, you may have a configuration provider that loads configuration values from a database, a provider that loads values from a remote API, or a provider that loads secrets from Azure Key Vault.[97]

Each of these providers require some sort of configuration themselves: a connection string for the database, a URL for the remote service, or a key to decrypt the data from Key Vault. Unfortunately, this leaves you with a circular problem: you need to add the provider to build your configuration object, but you need a configuration object to add the provider!

The solution is to use a two-stage process to build your final `IConfiguration` configuration object, as shown in figure 19.5. In the first stage, you load the configuration values that are

[97] Azure Key Vault is a service that lets you securely store secrets in the cloud. Your app retrieves the secrets from Azure Key Vault at runtime by calling an API and providing a client ID and a secret. The client ID and secret must come from local configuration values, so that you can retrieve the rest from Azure Key Vault. Read more, including how to get started, at https://docs.microsoft.com/aspnet/core/security/key-vault-configuration.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

available locally, such as JSON files and environment variables, and build a temporary `IConfiguration`. You can use this object to configure the complex providers, add them to your configuration builder, and build the final `IConfiguration` for your app.

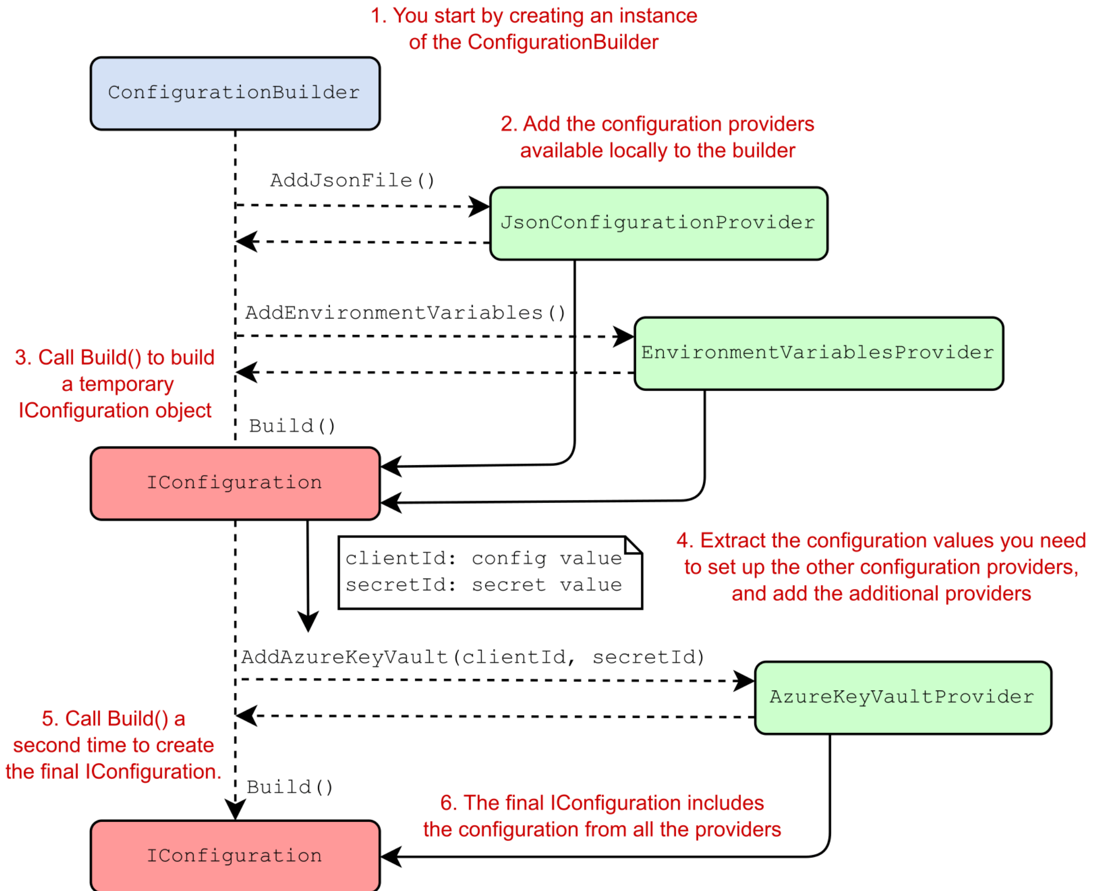


Figure 19.5 Adding a configuration provider that requires configuration. Start by adding configuration providers that you have the details for and build a temporary `IConfiguration` object. You can use this configuration object to load the settings required by the complex provider, add the provider to your builder, and build the final `IConfiguration` object using all three providers.

You can use this two-phase process whenever you have configuration providers that need configuration themselves. Building the configuration object twice means that the values are loaded from each of the initial configuration providers twice, I've never found that to be a problem.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

As an example of this process, in listing 19.12, you'll create a temporary `IConfiguration` object built using the contents of an XML file. This contains a configuration property called `"SettingsFile"` with the filename of a JSON file.

In the second phase of configuration, you add the JSON file provider (using the filename from the partial `IConfiguration`) and the environment variable provider. When you finally call `Build()` on the `IHostBuilder`, a new `IConfiguration` object will be built, containing the configuration values from the XML file, the JSON file, and the environment variables.

Listing 19.12 Using multiple `IConfiguration` objects to configure providers

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureAppConfiguration((context, config) =>
        {
            config.Sources.Clear();                           #A
            config.AddXmlFile("baseconfig.xml");              #B

            IConfiguration partialConfig = config.Build();    #C
            string filename = partialConfig["SettingsFile"];  #D

            config.AddJsonFile(filename)                      #E
                .AddEnvironmentVariables();                   #F
        })
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

#A Remove the default configuration sources
#B Adds an XML file to the configuration, which contains configuration for other providers
#C Builds the IConfiguration to read the XML file
#D Extracts the configuration required by other providers
#E Uses the extracted configuration to configure other providers
#F Remember, values from subsequent providers will overwrite values from previous providers

This is a somewhat contrived example—it's unlikely that you'd need to load configuration values to read a JSON file—but the principle is the same no matter which provider you use. A good example of this is the Azure Key Value provider. To load configuration values from Azure Key Vault, you need a URL, a client ID, and a secret. These must be loaded from other configuration providers, so you have to use the same two-phase process as shown in the previous listing.

Once you've loaded the configuration for your app, it's common to *bind* this configuration to strongly typed objects using the `IOptions` pattern. In the next section, we look at other ways to configure your `IOptions` objects and how to build them using DI services.

19.3.2 Using services to configure IOptions with IConfigureOptions

A common and encouraged practice is to bind your configuration object to strongly typed `IOptions<T>` objects, as you saw in chapter 11. Typically, you configure this binding in

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Startup.ConfigureServices by calling services.Configure<T>() and providing an IConfiguration object or section to bind.

To bind a strongly typed object, called CurrencyOptions, to the "Currencies" section of an IConfiguration object, you'd use

```
public void ConfigureServices(IServiceCollection services)
{
   services.Configure<CurrencyOptions>(
        Configuration.GetSection("Currencies"));
}
```

This sets the properties of the CurrencyOptions object, based on the values in the "Currencies" section of your IConfiguration object. Simple binding like this is common, but sometimes you might want to customize the configuration of your IOptions<T> objects, or you might not want to bind them to configuration at all. The IOptions pattern only requires you to configure a strongly typed object before it's injected into a dependent service, it doesn't mandate that you *have* to bind it to an IConfiguration section.

TIP **Technically, even if you don't configure an** IOptions<T> **at all, you can still inject it into your services. In that case, the** T **object will always be created using the default constructor.**

The services.Configure<T>() method has an overload that lets you provide a lambda function that the framework uses to configure the CurrencyOptions object. For example, in the following snippet, we use a lambda function to set the Currencies property on the configured CurrencyOptions object to a fixed array of strings:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CurrencyOptions>(
        Configuration.GetSection("Currencies"));
    services.Configure<CurrencyOptions>(options =>
        options.Currencies = new string[] { "GBP", "USD"});
}
```

Each call to Configure<T>(), both the binding to IConfiguration and the lambda function, adds another configuration step to the CurrencyOptions object. When the DI container first requires an instance of IOptions<CurrencyOptions>, each of the steps run in turn, as shown in figure 19.6.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

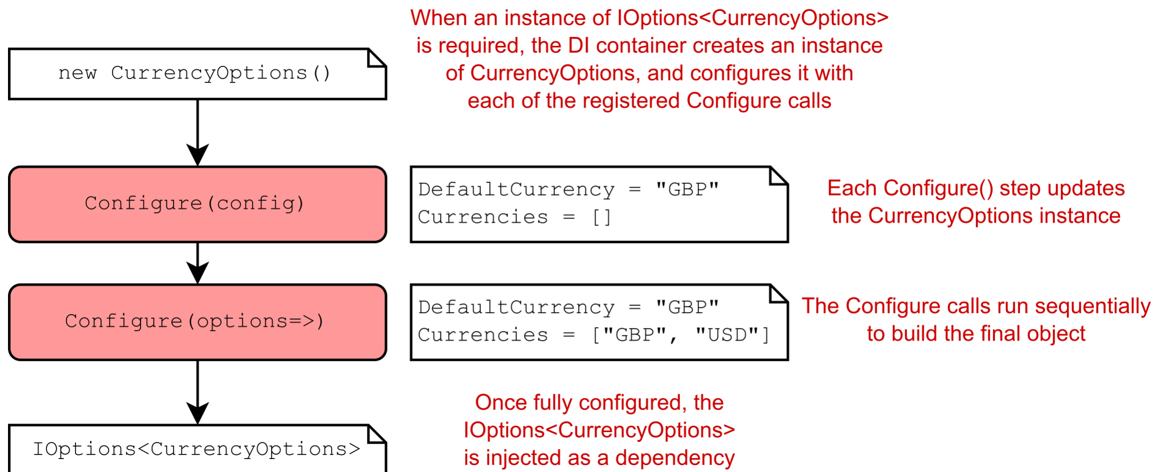


Figure 19.6 Configuring a `CurrencyOptions` object. When the DI container needs an `IOptions<>` instance of a strongly typed object, the container creates the object, and then uses each of the registered `Configure()` methods to set the object's properties.

In the example, you set the `Currencies` property to a static array of strings in a lambda function. But what if you don't know the correct values ahead of time? You might need to load the available currencies from a database, or from some remote service, for example.

Unfortunately, this situation, where you need a configured service to configure your `IOptions<T>` is hard to resolve. Remember, you declare your `IOptions<T>` configuration inside `ConfigureServices`, as part of the DI configuration. How can you get a fully configured instance of a currency service if you haven't registered it with the container yet?

The solution is to defer the configuration of your `IOptions<T>` object until the last moment, just before the DI container needs to create it to handle an incoming request. At that point, the DI container will be completely configured and will know how to create the currency service.

ASP.NET Core provides this mechanism with the `IConfigureOptions<T>` interface. You implement this interface in a configuration class and use it to configure the `IOptions<T>` object in any way you need. This class can use DI, so you can easily use any other required services.

Listing 19.13 Implementing `IConfigureOptions<T>` to configure an options object

```
public class ConfigureCurrencyOptions : IConfigureOptions<CurrencyOptions>
{
    private readonly ICurrencyProvider _currencyProvider;             #A
    public ConfigureCurrencyOptions(ICurrencyProvider currencyProvider)
    {
        _currencyProvider = currencyProvider;                         #A
    }
```

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
    public void Configure(CurrencyOptions options)                    #B
    {
        options.Currencies = _currencyProvider.GetCurrencies();       #C
    }
}
```

#A You can inject services that are only available after the DI is completely configured.
#B Configure is called when an instance of IOptions <CurrencyOptions> is required.
#C You can use the injected service to load the values from a remote API, for example.

All that remains is to register this implementation in the DI container. As always, order is important, so if you want `ConfigureCurrencyOptions` to run *after* binding to configuration, you must add it *after* the first call to `services.Configure<T>()`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CurrencyOptions>(
        Configuration.GetSection("Currencies"));
    services.AddSingleton
        <IConfigureOptions<CurrencyOptions>, ConfigureCurrencyOptions>();
}
```

> **WARNING** The `CurrencyConfigureOptions` **object is registered as a singleton, so it will capture any injected services of scoped or transient lifetimes.**[98]

With this configuration, when `IOptions<CurrencyOptions>` is injected into a controller or service, the `CurrencyOptions` object will first be bound to the `"Currencies"` section of your `IConfiguration` and will then be configured by the `ConfigureCurrencyOptions` class.

One piece of configuration not yet shown is `ICurrencyProvider` used by `ConfigureCurrencyOptions`. In the sample code for this chapter, I created a simple `CurrencyProvider` service and registered it with the DI container using

```
services.AddSingleton<ICurrencyProvider, CurrencyProvider>();
```

As your app grows, and you add extra features and services, you'll probably find yourself writing more and more of these simple DI registrations, where you register a `Service` that implements `IService`. The built-in ASP.NET Core DI container requires you to explicitly register each of these services manually. If you find this requirement frustrating, it may be time to look at third-party DI containers that can take care of some of the boilerplate for you.

---

[98] If you inject a scoped service into your configuration class (for example, a DbContext), you need to do a bit more work to ensure it's disposed of correctly. I describe how to achieve that here: http://mng.bz/6m17.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

19.4 Using a third-party dependency injection container

In this section I show how to replace the default dependency injection container with a third-party alternative, Lamar. Third-party containers often provide additional features compared to the built-in container, such as assembly scanning, automatic service registration, and property injection. Replacing the built-in container can also be useful when porting an existing app that uses a third-party DI container to ASP.NET Core.

The .NET community had been using DI containers for years before ASP.NET Core decided to include one that is built-in. The ASP.NET Core team wanted a way to use DI in their own framework libraries, and to create a common abstraction[99] that allows you to replace the built-in container with your favorite third-party alternative, such as

- Autofac
- StructureMap/Lamar
- Ninject
- Simple Injector
- Unity

The built-in container is intentionally limited in the features it provides, and it won't be getting many more realistically. In contrast, third-party containers can provide a host of extra features. These are some of the features available in Lamar (https://jasperfx.github.io/lamar/documentation/ioc/), the spiritual successor to StructureMap (https://structuremap.github.io/):

- Assembly scanning for interface/implementation pairs based on conventions
- Automatic concrete class registration
- Property injection and constructor selection
- Automatic `Lazy<T>`/`Func<T>` resolution
- Debugging/testing tools for viewing inside your container

None of these features are a requirement for getting an application up and running, so using the built-in container makes a lot of sense if you're building a small app or you're new to DI containers in general. But if, at some undefined tipping point, the simplicity of the built-in container becomes too much of a burden, it may be worth replacing it.

> **TIP** A middle-of-the-road approach is to use the Scrutor NuGet package, which adds some features to the built-in DI container, without replacing it entirely. For an introduction and examples, see https://andrewlock.net/using-scrutor-to-automatically-register-your-services-with-the-asp-net-core-di-container/.

[99] Although the promotion of DI as a core practice has been applauded, this abstraction has seen some controversy. This post from one of the maintainers of the SimpleInjector DI library describes many of the arguments and concerns: https://blog.simpleinjector.org/2016/06/whats-wrong-with-the-asp-net-core-di-abstraction/. You can also read more about the decisions here: https://github.com/aspnet/DependencyInjection/issues/433.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

In this section, I show how you can configure an ASP.NET Core app to use Lamar for dependency resolution. It won't be a complex example, or an in-depth discussion of Lamar itself. Instead, I'll cover the bare minimum to get up and running.

Whichever third-party container you choose to install into an existing app, the overall process is pretty much the same:

1. Install the container NuGet package.
2. Register the third-party container with the `IHostBuilder` in Program.cs
3. Add a `ConfigureContainer` method in `Startup`.
4. Configure the third-party container in `ConfigureContainer` to register your services.

Most of the major .NET DI containers have been ported to work on .NET Core and include an adapter that lets you add them to ASP.NET Core apps. For details, it's worth consulting the specific guidance for the container you're using. For Lamar, the process looks like this:

1. Install the Lamar.Microsoft.DependencyInjection NuGet package using the NuGet package manager, by running dotnet add package

   ```
   dotnet add package Lamar.Microsoft.DependencyInjection
   ```

   or by adding a `<PackageReference>` to your csproj file

   ```
   <PackageReference
       Include="Lamar.Microsoft.DependencyInjection" Version="4.3.0" />
   ```

2. Call `UseLamar()` on your `IHostBuilder` in Program.cs

   ```
   public static IHostBuilder CreateHostBuilder(string[] args) =>
       Host.CreateDefaultBuilder(args)
           .UseLamar()
           .ConfigureWebHostDefaults(webBuilder =>
           {
               webBuilder.UseStartup<Startup>();
           }
   ```

3. Add a `ConfigureContainer` method to your `Startup` class, with the following signature:

   ```
   public void ConfigureContainer(ServiceRegistry services) { }
   ```

4. Configure the Lamar `ServiceRegistry` in `ConfigureContainer`, as shown in the following listing. This is a basic configuration, but you can see a more complex example in the source code for this chapter.

Listing 19.14 Configuring StructureMap as a third-party DI container

```
public void ConfigureContainer(ServiceRegistry services)        #A
{
    services.AddAuthorization();                                #B
    services.AddControllers()                                   #B
        .AddControllersAsServices();                            #C
```


Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
    config.Scan(_ => {                                          #D
        _.AssemblyContainingType(typeof(Startup));              #D
        _.WithDefaultConventions();                             #D
    });                                                         #D
}
```

#A Configure your services in ConfigureContainer instead of ConfigureServices
#B You can (and should) add ASP.NET Core framework services to the ServiceRegistry as usual.
#C Required so that Lamar is used to build your API controllers.
#D Lamar can automatically scan your assemblies for services to register.

In this example, I've used the default conventions to register services. This will automatically register concrete classes and services that are named following expected conventions (for example, `Service` implements `IService`). You can change these conventions or add other registrations in the `ConfigureContainer` method.

The `ServiceRegistry` passed into `ConfigureContainer` implements `IServiceCollection`, which means you can use all the built-in extension methods, such as `AddControllers()` and `AddAuthorization()`, to add framework services to your container.

> **WARNING** If you're using DI in your MVC controllers (almost certainly!) and you register those dependencies with Lamar, rather than the built-in container, you may need to call `AddControllersAsServices()`, as shown in listing 19.14. This is due to an implementation detail in the way the your MVC controllers are created by the framework. For details, see https://andrewlock.net/controller-activation-and-dependency-injection-in-asp-net-core-mvc/.

With this configuration in place, whenever your app needs to create a service, it will request it from the Lamar container, which will create the dependency tree for the class and create an instance. This example doesn't show off the power of Lamar, so be sure to check out the documentation (https://jasperfx.github.io/lamar/) and the associated source code for this chapter for more examples. Even in modestly sized applications, Lamar can greatly simplify your service registration code, but its party trick is showing all the services you have registered, and any associated issues.

That brings us to the end of this chapter on custom components. In this chapter, I focused on some of the most common components you will build for the configuration, dependency injection, and middleware systems of ASP.NET Core. In the next chapter, you'll learn about more custom components, with a focus on Razor Pages and API controllers.

19.5 Summary

- Use the `Run` extension method to create middleware components that always return a response. You should always place the `Run` extension at the end of a middleware pipeline or branch, as middleware placed after it will never execute.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

- You can create branches in the middleware pipeline with the `Map` extension. If an incoming request matches the specified path prefix, the request will execute the pipeline branch, otherwise it will execute the trunk.
- When the `Map` extension matches a request path segment, it removes the segment from the request's `HttpContext.Path` and moves it to the `PathBase` property. This ensures that routing in branches works correctly.
- You can use the `Use` extension method to create generalized middleware components that can generate a response, modify the request, or pass the request on to subsequent middleware in the pipeline. This is useful for cross-cutting concerns, like adding a header to all responses.
- You can encapsulate middleware in a reusable class. The class should take a `RequestDelegate` object in the constructor, and should have a public `Invoke()` method that takes an `HttpContext` and returns a `Task`. To call the next middleware in the pipeline, invoke the `RequestDelegate` with the provided `HttpContext`.
- To create endpoints that generate a response, build a miniature pipeline containing the response-generating middleware, and call `endpoints.Map(route, pipeline)`. Endpoint routing will be used to map incoming requests to your endpoint.
- You can attach metadata to endpoints which is made available to any middleware placed between the calls to `UseRouting()` and `UseEndpoints()`. This metadata enables functionality such as authorization and CORS.
- To add authorization to an endpoint, call `RequireAuthorization()` after mapping the endpoint. This is equivalent to using the `[Authorize]` attribute on Razor Pages and API controllers. You can optionally provide an authorization policy name, instead of using the default policy.
- Some configuration providers require configuration values themselves. For example, a configuration provider that loads settings from a database might need a connection string. You can load these configuration providers by partially building an `IConfiguration` object using the other providers and reading the required configuration from it. You can then configure the database provider and add it to the `ConfigurationBuilder` before rebuilding to get the final `IConfiguration`.
- If you need to use services from the DI container to configure an `IOptions<T>` object, then you should create a separate class that implements `IConfigureOptions<T>`. This class can use DI in the constructor and is used to lazily build a requested `IOptions<T>` object at runtime.
- You can replace the built-in DI container with a third-party container. Third-party containers often provide additional features, such as convention-based dependency registration, assembly scanning, or property injection.
- To use a third-party container such as Lamar, install the NuGet package, enable the container on `IHostBuilder`, and implement `ConfigureContainer()` in Startup. Configure the third-party container in this method by registering both the required ASP.NET Core framework services and your app specific services.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

20 Building custom MVC and Razor Pages components

This chapter covers

- **Creating custom Razor Tag Helpers**
- **Using view components to create complex Razor views**
- **Creating a custom `DataAnnotations` validation attribute**
- **Replacing the `DataAnnotations` validation framework with an alternative**

In the previous chapter, you learned how to customize and extend some of the core systems in ASP.NET Core: configuration, dependency injection, and your middleware pipeline. These components form the basis of all ASP.NET Core apps. In this chapter we're focusing on Razor Pages and MVC/API controllers. You'll learn how to build custom components that work with Razor views, as well as the validation framework used by both Razor Pages and API controllers.

We start by looking at Tag Helpers. In section 20.1, I show you how to create two different Tag Helpers: one that generates HTML to describe the current machine, and one that lets you write `if` statements in Razor templates without having to use C#. These will give you the details you need to create your own custom Tag Helpers in your own apps if the need arises.

In section 20.2, you'll learn about a new Razor concept: view components. View components are a bit like partial views, but they can contain business logic and database access. For example, on an e-commerce site, you might have a shopping cart, a dynamically populated menu, and a login widget, all on one page. Each of those sections is independent of the main page content and has its own logic and data-access needs. In an ASP.NET Core app using Razor Pages, you'd implement each of those as a View Component.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

In section 20.3, I show how to create a custom validation attribute. As you saw in chapter 6, validation is a key responsibility of Razor Page handlers and action methods, and the `DataAnnotations` attributes provide a clean, declarative way for doing so. We previously only looked at the built-in attributes, but you'll often find you need to add attributes tailored to your app's domain. In section 20.3, you'll see how to create a simple validation attribute, and how to extend it to use services registered with the DI container.

Throughout this book I've mentioned that you can easily swap out core parts of the ASP.NET Core framework if you wish. In section 20.4 you'll do just that, by replacing the built-in attribute-based validation framework with a popular alternative, FluentValidation. This open source library allows you to separate your binding models from the validation rules, which makes building certain validation logic easier. Many people prefer this approach of separating concerns to the declarative approach of `DataAnnotations`.

When you're building pages with Razor Pages, one of the best productivity features is Tag Helpers, and in the next section you'll see how you can create your own.

20.1 Creating a custom Razor Tag Helper

In this section you'll learn how to create your own Tag Helpers, which allow you to customize your HTML output. You'll learn how to create Tag Helpers that add new elements to your HTML markup, as well as Tag Helpers that can be used to remove or customize existing markup. You'll also see that your custom Tag Helpers integrate with the tooling of your IDE to provide rich IntelliSense in the same way as the built-in Tag Helpers.

In my opinion, Tag Helpers are one of the best additions to the venerable Razor template language in ASP.NET Core. They allow you to write Razor templates that are easier to read, as they require less switching between C# and HTML, and they *augment* your HTML tags, rather than replacing them (as opposed to the HTML Helpers used extensively in the previous version of ASP.NET).

ASP.NET Core comes with a wide variety of Tag Helpers (see chapter 8), which will cover many of your day-to-day requirements, especially when it comes to building forms. For example, you can use the Input Tag Helper by adding an `asp-for` attribute to an `<input>` tag and passing a reference to a property on your `PageModel`, in this case `Input.Email`:

```
<input asp-for="Input.Email" />
```

The Tag Helper is activated by the presence of the attribute and gets a chance to augment the `<input>` tag when rendering to HTML. The Input Tag Helper uses the name of the property to set the `<input>` tag's `name` and `id` properties, the value of the model to set the `value` property, and the presence of attributes such as `[Required]` or `[EmailAddress]` to add attributes for validations:

```
<input type="email" id="Input_Email" name="Input.Email"
    value="test@example.com" data-val="true"
    data-val-email="The Email Address field is not a valid e-mail address."
    data-val-required="The Email Address field is required."
    />
```

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Tag Helpers help reduce the duplication in your code, or they can simplify common patterns. In this section, I show how you can create your own custom Tag Helpers.

In section 20.1.1, you create a system information Tag Helper, which prints details about the name and operating system of the server your app is running on. In section 20.1.2, you create a Tag Helper that you can use to conditionally show or hide an element based on a C# Boolean property. In section 20.1.3 you create a Tag Helper that reads the Razor content written *inside* the Tag Helper and transforms it.

20.1.1 Printing environment information with a custom Tag Helper

A common problem you may run into when you start running your web applications in production, especially if you're using a server-farm setup, is working out which machine rendered the page you're currently looking at. Similarly, when deploying frequently, it can be useful to know which *version* of the application is running. When I'm developing and testing, I sometimes like to add a little "info dump" at the bottom of my layouts, so I can easily work out which server generated the current page, which environment it's running in, and so on.

In this section, I'm going to show you how to build a custom Tag Helper to output system information to your layout. You'll be able to toggle the information it displays, but by default, it will display the machine name and operating system on which the app is running, as shown in figure 20.1.

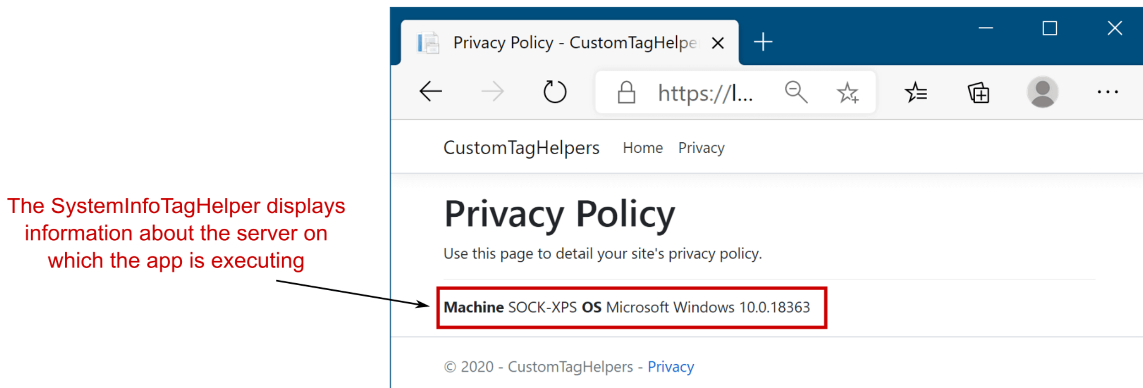


Figure 20.1 The `SystemInfoTagHelper` displays the machine name and operating system on which the application is running. It can be useful for identifying which instance of your app handled the request when running in a web farm scenario.

You can call this Tag Helper from Razor by creating a `<system-info>` element in your template, for example:

```
<footer>
  <system-info></system-info>
</footer>
```

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

TIP You probably don't want to expose this sort of information in production, so you could also wrap it in an `<environment>` Tag Helper, as you saw in chapter 8.

The easiest way to create a custom Tag Helper is to derive from the `TagHelper` base class and override the `Process()` or `ProcessAsync()` function that describes how the class should render itself. The following listing shows your complete custom Tag Helper, the `SystemInfoTagHelper`, which renders the system information to a `<div>`. You could easily extend this class if you wanted to display additional fields or add additional options.

Listing 20.1 `SystemInfoTagHelper` to render system information to a view

```
public class SystemInfoTagHelper : TagHelper      #A
{
    private readonly HtmlEncoder _htmlEncoder;                #B
    public SystemInfoTagHelper(HtmlEncoder htmlEncoder)       #B
    {
        _htmlEncoder = htmlEncoder;
    }

    [HtmlAttributeName("add-machine")]        #C
    public bool IncludeMachine { get; set; } = true;

    [HtmlAttributeName("add-os")]         #C
    public bool IncludeOS { get; set; } = true;

    public override void Process(        #D
        TagHelperContext context, TagHelperOutput output)       #D
    {
        output.TagName = "div";            #E
        output.TagMode = TagMode.StartTagAndEndTag;      #F
        var sb = new StringBuilder();

        if (IncludeMachine)      #G
        {            #G
           sb.Append(" <strong>Machine</strong> ");    #G
           sb.Append(_htmlEncoder.Encode(Environment.MachineName));      #G
        }             #G

        if (IncludeOS)           #H
        {                #H
           sb.Append(" <strong>OS</strong> ");     #H
           sb.Append(         #H
              _htmlEncoder.Encode(RuntimeInformation.OSDescription));    #H
        }          #H
        output.Content.SetHtmlContent(sb.ToString());         #I
    }
}
```

#A Derives from the TagHelper base class
#B An HtmlEncoder is necessary when writing HTML content to the page.
#C Decorating properties with HtmlAttributeName allows you to set their value from Razor markup
#D The main function called when an element is rendered.
#E Replaces the <system-info> element with a <div> element
#F Renders both the <div> </div> start and end tag

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

#G If required, adds a <strong> element and the HTML-encoded machine name
#H If required, adds a <strong> element and the HTML-encoded OS name
#I Sets the inner content of the <div> tag with the HTML-encoded value stored in the string builder

There's a lot of new code in this sample, so we'll work through it line by line. First, the class name of the Tag Helper defines the name of the element you must create in your Razor template, with the suffix removed and converted to kebab-case. As this Tag Helper is called `SystemInfoTagHelper`, you must create a `<system-info>` element.

> **TIP** **If you want to customize the name of the element, for example to `<env-info>`, but want to keep the same class name, you can apply the `[HtmlTargetElement]` with the desired name, such as `[HtmlTargetElement("Env-Info")]`. HTML tags are not case sensitive, so you could use `"Env-Info"` or `"env-info"`.**

Inject an `HtmlEncoder` into your Tag Helper so you can HTML-encode any data you write to the page. As you saw in chapter 18, you should always HTML-encode data you write to the page to avoid XSS vulnerabilities (and to ensure the data is displayed correctly).

You've defined two properties on your Tag Helper, `IncludeMachine` and `IncludeOS`, which you'll use to control which data is written to the page. These are decorated with a corresponding `[HtmlAttributeName]`, which enables setting the properties from the Razor template. In Visual Studio, you'll even get IntelliSense and type checking for these values, as shown in figure 20.2.

```
<footer>
    <system-info add-os="false" add-/>
</footer>
```

add-machine
bool CustomTagHelpers.SystemInfoTagHelper.IncludeMachine
Show the current System.Environment.MachineName. true by default

Figure 20.2 In Visual Studio, Tag Helpers are shown in a purple font and you get IntelliSense for properties decorated with `[HtmlAttributeName]`.

Finally, we come to the `Process()` method. The Razor engine calls this method to execute the Tag Helper when it identifies the target element in a view template. The `Process()` method defines the type of tag to render (`<div>`), whether it should render a start and end tag (or a self-closing tag—it depends on the type of tag you're rendering), and the HTML content of the `<div>`. You set the HTML content to be rendered inside the tag by calling `Content.SetHtmlContent()` on the provided instance of `TagHelperOutput`.

> **WARNING** **Always HTML-encode your output before writing to your tag with `SetHtmlContent()`. Alternatively, pass unencoded input to `SetContent()` and the output will be automatically HTML-encoded for you.**

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Before you can use your new Tag Helper in a Razor template, you need to register it. You can do this in the _ViewImports.cshtml file, using the `@addTagHelper` directive and specifying the fully qualified name of the Tag Helper and the assembly. For example,

```
@addTagHelper CustomTagHelpers.SystemInfoTagHelper, CustomTagHelpers
```

Alternatively, you can add all the Tag Helpers from a given assembly by using the wildcard syntax, `*`, and specifying the assembly name:

```
@addTagHelper *, CustomTagHelpers
```

With your custom Tag Helper created and registered, you're now free to use it in any of your Razor views, partial views, or layouts.

> **TIP** **If you're not seeing IntelliSense for your Tag Helper in Visual Studio, and the Tag Helper isn't rendered in the bold font used by Visual Studio, then you probably haven't registered your Tag Helpers correctly in __ViewImports.cshtml using** `@addTagHelper`**.**

The `SystemInfoTagHelper` is an example of a Tag Helper that generates content, but you can also use Tag Helpers to control how existing elements are rendered. In the next section, you'll create a simple Tag Helper that can control whether or not an element is rendered, based on an HTML attribute.

20.1.2 Creating a custom Tag Helper to conditionally hide elements

If you want to control whether an element is displayed in a Razor template based on some C# variable, then you'd typically wrap the element in a C# `if` statement:

```
@{
    var showContent = true;
}
@if(showContent)
{
    <p>The content to show</p>
}
```

Falling back to C# constructs like this can be useful, as it allows you to generate any markup you like. Unfortunately, it can be mentally disruptive having to switch back and forth between C# and HTML, and it makes it harder to use HTML editors that don't understand Razor syntax.

In this section, you'll create a simple Tag Helper to avoid the cognitive dissonance problem. You can apply this Tag Helper to existing elements to achieve the same result as shown previously, but without having to fall back to C#:

```
@{
    var showContent = true;
}
<p if="showContent">
    The content to show
</p>
```

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Instead of creating a new *element*, as you did for SystemInfoTagHelper (<system-info>), you'll create a Tag Helper that you apply as an *attribute* to existing HTML elements. This Tag Helper does one thing: it controls the visibility of the element it's attached to. If the value passed in the if attribute is true, the element and its content is rendered as normal. If the value passed is false, the Tag Helper removes the element and its content from the template. Here's how you could achieve this.

Listing 20.2 Creating an IfTagHelper to conditionally render elements

```
[HtmlTargetElement(Attributes = "if")]        #A
public class IfTagHelper : TagHelper
{
    [HtmlAttributeName("if")]            #B
    public bool RenderContent { get; set; } = true;

    public override void Process(                           #C
        TagHelperContext context, TagHelperOutput output)    #C
    {
        if(RenderContent == false)            #D
        {
            output.TagName = null;         #E
            output.SuppressOutput();        #F
        }
    }

    public override int Order => int.MinValue;      #G
}
```

#A Setting the Attributes property ensures the Tag Helper is triggered by an if attribute.
#B Binds the value of the if attribute to the RenderContent property
#C The Razor engine calls Process() to execute the Tag Helper.
#D If the RenderContent property evaluates to false, removes the element
#E Sets the element the Tag Helper resides on to null, removing it from the page
#F Doesn't render or evaluate the inner content of the element
#G Ensures this Tag Helper runs before any others attached to the element

Instead of a standalone <if> element, the Razor engine executes the IfTagHelper whenever it finds an element with an if attribute. This can be applied to any HTML element: <p>, <div>, <input>, whatever you need. Define a Boolean property for whether you should render the content, which is bound to the value in the if attribute.

The Process() function is much simpler here. If RenderContent is false, then it sets the TagHelperOutput.TagName to null, which removes the element from the page. You also call SuppressOutput(), which prevents any content *inside* the attributed element from being rendered. If RenderContent is true, then you skip these steps and the content is rendered as normal.

One other point of note is the overridden Order property. This controls the order in which Tag Helpers run when multiple Tag Helpers are applied to an element. By setting Order to int.MinValue, you ensure IfTagHelper will run first, removing the element if required,

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

before other Tag Helpers execute. There's generally no point running other Tag Helpers if the element is going to be removed from the page anyway!

> **NOTE** Remember to register your custom Tag Helpers in _ViewImports .cshtml with the `@addTagHelper` directive.

With a simple HTML attribute, you can now conditionally render elements in Razor templates, without having to fall back to C#. This tag helper can show and hide content without needing to know what the content is. In the next section, we'll create a Tag Helper that does *need* to know the content.

20.1.3 Creating a Tag Helper to convert Markdown to HTML

The two Tag Helpers shown so far are agnostic to the content written *inside* the Tag Helper, but it can also be useful to create Tag Helpers that inspect, retrieve, and modify this content. In this section you'll see an example of one such Tag Helper that converts Markdown content written inside it into HTML.

> **DEFINITION** Markdown is a commonly used text-based markup language that is easy to read but can also be converted into HTML. It is the common format used by README files on GitHub, and I use it to write blog posts, for example. For an introduction to Markdown, see https://guides.github.com/features/mastering-markdown/.

We'll use the popular Markdig library (https://github.com/lunet-io/markdig/) to create the Markdown Tag Helper. This library converts a `string` containing markdown into an HTML `string`. You can install Markdig using Visual Studio, by running `dotnet add package Markdig`, or by adding a `<PackageReference>` to your csproj file:

```
<PackageReference Include="Markdig" Version="0.20.0" />
```

The Markdown Tag Helper that we'll create shortly can be used by adding `<markdown>` elements to your Razor Page, as shown in the following listing.

Listing 20.3 Using a Markdown Tag Helper in a Razor Page

```
@page
@model IndexModel

<markdown>                                                    #A
## This is a markdown title                                   #B

This is a markdown list:                                      #C

* Item 1                                                      #C
* Item 2                                                      #C

<div if="showContent">                                        #D
  Content is shown when showContent is true                   #D
</div>                                                        #D
```

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
</markdown>
```

#A The Markdown Tag Helper is added using the <markdown> element
#B Titles can be created in Markdown using # to denote h1, ## to denote h2, and so on
#C Markdown converts simple lists to HTML <ul> elements
#D Razor content can be nested inside other Tag Helpers

The Markdown Tag Helper renders content by:

1. Rendering any Razor content inside the Tag Helper. This includes executing any *nested* Tag Helpers and C# code inside the Tag Helper. The above example uses the `IfTagHelper`, for example.
2. Converting the resulting `string` to HTML using the Markdig library.
3. Replacing the content with the rendered HTML and removing the Tag Helper `<markdown>` element.

The following listing shows a simple approach to implementing a Markdown Tag Helper using Markdig. Markdig supports many additional extensions and features that you could enable, but the overall pattern of the Tag Helper would be the same.

Listing 20.4 Implementing a Markdown Tag Helper using Markdig

```
public class MarkdownTagHelper: TagHelper                            #A
{
    public override async Task ProcessAsync(
        TagHelperContext context, TagHelperOutput output)
    {
        TagHelperContent markdownRazorContent = await                #B
            output.GetChildContentAsync(NullHtmlEncoder.Default);    #B
        string markdown =                                            #C
            markdownRazorContent.GetContent(NullHtmlEncoder.Default);    #C

        string html = Markdig.Markdown.ToHtml(markdown);             #D

        output.Content.SetHtmlContent(html);                         #E
        output.TagName = null;                                       #F
    }
}
```

#A The Markdown Tag Helper will use the <markdown> element
#B Retrieve the contents of the <markdown> element
#C Render the Razor contents to a string
#D Convert the markdown string to HTML using Markdig
#E Write the HTML content to the output
#F Remove the <markdown> element from the content

When rendered to HTML, the Markdown content in listing 20.3 (when the `showContent` variable is `true`) becomes:

```
<h2>This is a markdown title</h2>
<p>This is a markdown list:</p>
<ul>
<li>Item 1</li>
```

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
<li>Item 2</li>
</ul>
<div>
  Content is shown when showContent is true
</div>
```

NOTE In Listing 20.4 we implemented `ProcessAsync()` instead of `Process()`. That is because we call the `async` method, `GetChildContentAsync()`. You must only call `async` methods from other `async` methods, as otherwise you can get issues such as thread starvation. For more details, see https://docs.microsoft.com/aspnet/core/performance/performance-best-practices.

The Tag Helpers in this section represent a small sample of possible avenues you could explore,[100] but they cover the two broad categories: Tag Helpers for rendering new content, and Tag Helpers for controlling the rendering of other elements.

Tag Helpers can be useful for providing small pieces of isolated, reusable functionality like this, but they're not designed to provide larger, application-specific sections of an app or to make calls to business-logic services. Instead, you should use view components, as you'll see in the next section.

20.2 View components: adding logic to partial views

In this section you'll learn about view components. View components operate independently of the main Razor Page and can be used to encapsulate complex business logic. You can use view components to keep your main Razor Page focused on a single task, rendering the main content, instead of also being responsible for other sections of the page.

If you think about a typical website, you'll notice that they often have multiple independent dynamic sections, in addition to the main content. Consider Stack Overflow, shown in figure 20.3, for example. As well as the main body of the page showing questions and answers, there's a section showing the current logged-in user, a panel for blog posts and related items, and a section for job suggestions.

[100] For further details and examples, see the documentation at http://mng.bz/Idb0.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Figure 20.3 The Stack Overflow website has multiple sections that are independent of the main content, but which contain business logic and complex rendering logic. Each of these sections could be rendered as a view component in ASP.NET Core.

Each of these sections is effectively independent of the main content. Each section contains business logic (deciding which posts or ads to show), database access (loading the details of the posts), and rendering logic for how to display the data. In chapter 7, you saw that you can use layouts and partial views to split up the rendering of a view template into similar sections, but partial views aren’t a good fit for this example. Partial views let you encapsulate *view rendering* logic, but not *business* logic that’s independent of the main page content.

Instead, *view components* provide this functionality, encapsulating both the business logic and rendering logic for displaying a small section of the page. You can use DI to provide access to a database context, and you can test them independently of the view they generate, much like MVC and API controllers. Think of them a bit like mini-MVC controllers, or mini-Razor Pages, but you invoke them directly from a Razor view, instead of in response to an HTTP request.

> **TIP** View components are comparable to *child actions* from the previous version of ASP.NET, in that they provide similar functionality. Child actions don’t exist in ASP.NET Core.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

View components versus Razor Components and Blazor

In this book I am focusing on server-side rendered applications using Razor Pages and API applications using API controllers. .NET Core 3.0 introduced a completely new approach to building ASP.NET Core applications: Blazor. I don't cover Blazor in this book, so I recommend reading *Blazor in Action* by Chris Sainty (Manning, 2021).

Blazor has two programming models, client-side and server-side, but both approaches use *Blazor components* (confusingly, officially called *Razor components*). Blazor components have a lot of parallels with view components, but they live in a fundamentally different world. Blazor components can interact with each other easily, but you can't use them with Tag Helpers or view components, and it's hard to combine them with Razor Page form posts.

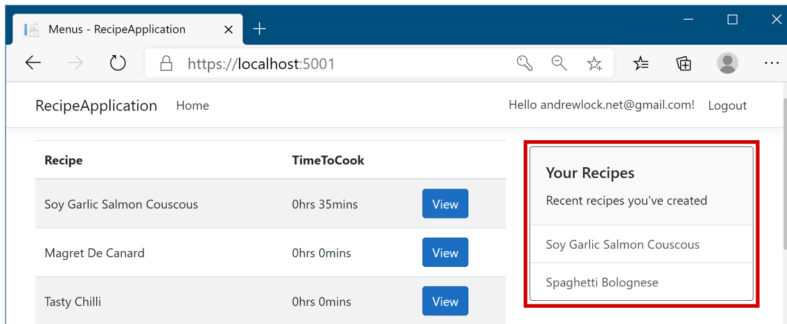
Nevertheless, if you need an "island" of rich client-side interactivity in a single Razor Page, you can embed a Blazor component inside a Razor Page, as shown in the documentation: https://docs.microsoft.com/aspnet/core/blazor/components/integrate-components. You could also use Blazor components as a way to replace AJAX calls in your Razor Pages, as I show in https://andrewlock.net/replacing-ajax-calls-in-razor-pages-using-razor-components-and-blazor/.

If you don't need the client-side interactivity of Blazor, then view components are still the best option for isolated sections in Razor Pages. They interoperate cleanly with your Razor Pages, have no additional operational overhead, and use familiar concepts like layouts, partial views and Tag Helpers. For more details on why you should continue to use view components see https://andrewlock.net/dont-replace-your-view-components-with-razor-components/.

In this section, you'll see how to create a custom view component for the recipe app you built in previous chapters, as shown in figure 20.4. If the current user is logged in, the view component displays a panel with a list of links to the user's recently created recipes. For unauthenticated users, the view component displays links to the login and register actions.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>



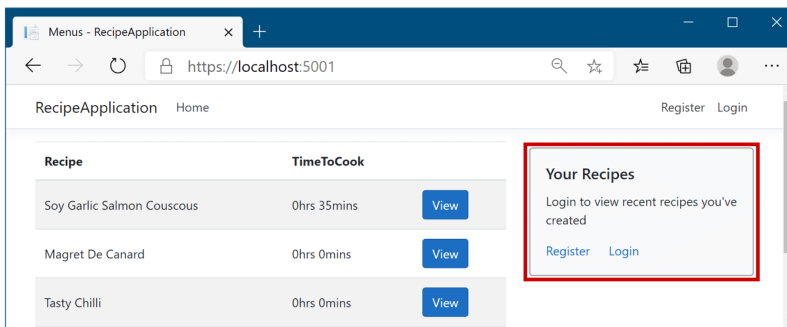


Figure 20.4 The view component displays different content based on the currently logged-in user. It includes both business logic (which recipes to load from the database) and rendering logic (how to display the data).

This component is a great candidate for a view component as it contains database access and business logic (choosing which recipes to display), as well as rendering logic (how the panel should be displayed).

> **TIP** Use partial views when you want to encapsulate the rendering of a specific view model, or part of a view model. When you have rendering logic that requires business logic or database access, or where the section is logically distinct from the main page content, consider using a view component.

You invoke view components directly from Razor views and layouts using a Tag Helper-style syntax, using a `vc:` prefix:

```
<vc:my-recipes number-of-recipes="3">
</vc:my-recipes>
```

Custom view components typically derive from the `ViewComponent` base class, and implement an `InvokeAsync()` method, as shown in listing 20.5. Deriving from this base class allows access to useful helper methods, in much the same way that deriving from the

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

ControllerBase class does for API controllers. Unlike API controllers, the parameters passed to InvokeAsync don't come from model binding. Instead, you pass the parameters to the view component using properties on the Tag Helper element in your Razor view.

Listing 20.5 A custom view component to display the current user's recipes

```
public class MyRecipesViewComponent : ViewComponent              #A
{
    private readonly RecipeService _recipeService;               #B
    private readonly UserManager<ApplicationUser> _userManager;   #B
    public MyRecipesViewComponent(RecipeService recipeService,    #B
        UserManager<ApplicationUser> userManager)                 #B
    {                                                             #B
        _recipeService = recipeService;                           #B
        _userManager = userManager;                               #B
    }                                                             #B

    public async Task<IViewComponentResult> InvokeAsync(          #C
        int numberOfRecipes)                                      #D
    {
        if(!User.Identity.IsAuthenticated)
        {
            return View("Unauthenticated");                       #E
        }

        var userId = _userManager.GetUserId(HttpContext.User);    #F
        var recipes = await _recipeService.GetRecipesForUser(     #F
            userId, numberOfRecipes);

        return View(recipes);                                     #G
    }
}
```

#A Deriving from the ViewComponent base class provides useful methods like View().
#B You can use DI in a view Component.
#C InvokeAsync renders the view component. It should return a Task<IViewComponentResult>
#D You can pass parameters to the component from the view.
#E Calling View() will render a partial view with the provided name.
#F You can use async external services, allowing you to encapsulate logic in your business domain.
#G You can pass a view model to the partial view. Default.cshtml is used by default.

This custom view component handles all the logic you need to render a list of recipes when the user is logged in, or a different view if the user isn't authenticated. The name of the view component is derived from the class name, like Tag Helpers. Alternatively, you can apply the [ViewComponent] attribute to the class and set a different name entirely.

The InvokeAsync method must return a Task<IViewComponentResult>. This is similar to the way you can return IActionResult from an action method or a page handler, but it's more restrictive; view components *must* render some sort of content, so you can't return status codes or redirects. You'll typically use the View() helper method to render a partial view template (as in the previous listing) though you can also return a string directly using the

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

`Content()` helper method, which will HTML-encode the content and render it to the page directly.

You can pass any number of parameters to the `InvokeAsync` method. The name of the parameters (in this case, `numberOfRecipes`) is converted to kebab-case and exposed as a property in the view component's Tag Helper (`<number-of-recipes>`). You can provide these parameters when you invoke the view component from a view, and you'll get IntelliSense support, as show in figure 20.5.

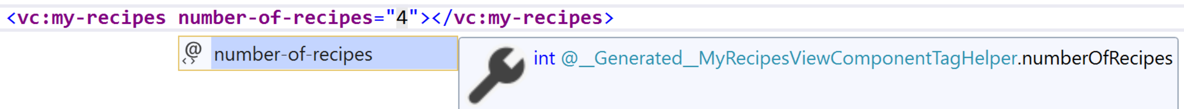


Figure 20.5 Visual Studio provides IntelliSense support for the method parameters of a view component's `InvokeAsync` method. The parameter name, in this case `numberOfRecipes`, is converted to kebab-case for use as an attribute in the Tag Helper.

View components have access to the current request and `HttpContext`. In listing 20.5, you can see that we're checking whether the current request was from an authenticated user. You can also see that we've used some conditional logic: if the user isn't authenticated, render the "Unauthenticated" Razor template, if they're authenticated, render the default Razor template and pass in the view models loaded from the database.

> **NOTE** If you don't specify a specific Razor view template to use in the `View()` function, view components use the template name, "Default.cshtml."

The partial views for view components work similarly to other Razor partial views that you learned about in chapter 7, but they're stored separately from them. You must create partial views for view components at either:

- Views/Shared/Components/ComponentName/TemplateName, or
- Pages/Shared/Components/ComponentName/TemplateName

Both locations work, so for Razor Pages app, I typically use the Pages/ folder. For the view component in listing 20.5, for example, you'd create your view templates at

- Pages/Shared/Components/MyRecipes/Default.cshtml
- Pages/Shared/Components/MyRecipes/Unauthenticated.cshtml

This was only a quick introduction to view components, but it should get you a long way. View components are a simple method to embed pockets of isolated, complex logic in your Razor layouts. Having said that, be mindful of these caveats:

- View component classes must be public, non-nested, and non-abstract classes.
- Although similar to MVC controllers, you can't use filters with view components.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

- You can use Layouts in your view components views, to extract rendering logic common to a specific view component. This layout may contain `@sections`, as you saw in chapter 7, but these sections are independent of the "main" Razor view's layout.
- View components are isolated from the Razor Page they're rendered in, so you can't, for example, define a `@section` in a Razor Page layout, and then add that content from a view component; the contexts are completely separate
- When using the `<vc:my-recipes>` Tag Helper syntax to invoke your view component, you must import it as a custom Tag Helper, as you saw in section 20.1.
- Instead of using the Tag Helper syntax, you may invoke the view component from a view directly by using `IViewComponentHelper Component`, though I don't recommend using this syntax. For example:

```
@await Component.InvokeAsync("MyRecipes", new { numberOfRecipes = 3 })
```

We've covered Tag Helpers and view components, which are both features of the Razor engine in ASP.NET Core. In the next section, you'll learn about a different, but related, topic: how to create a custom `DataAnnotations` attribute. If you've used previous versions of ASP.NET, then this will be familiar, but ASP.NET Core has a couple of tricks up its sleeve to help you out.

20.3 Building a custom validation attribute

In this section you'll learn how to create a custom `DataAnnotations` validation attribute that specifies specific values a `string` property may take. You'll then learn how you can expand the functionality to be more generic by delegating to a separate service that is configured in your DI controller. This will allow you to create custom domain-specific validations for your apps.

We looked at model binding in chapter 6, where you saw how to use the built-in `DataAnnotations` attributes in your binding models to validate user input. These provide several built-in validations, such as

- `[Required]`—The property isn't optional and must be provided.
- `[StringLength(min, max)]`—The length of the `string` value must be between `min` and `max` characters.
- `[EmailAddress]`—The value must be a valid email address format.

But what if these attributes don't meet your requirements? Consider the following listing, which shows a binding model from a currency converter application. The model contains three properties: the currency to convert from, the currency to convert to, and the quantity.

Listing 20.6 Currency converter initial binding model

```
public class CurrencyConverterModel
{
    [Required]                                  #A
    [StringLength(3, MinimumLength = 3)]        #B
    public string CurrencyFrom { get; set; }
```

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
    [Required]                                  #A
    [StringLength(3, MinimumLength = 3)]        #B
    public string CurrencyTo { get; set; }

    [Required]                                  #A
    [Range(1, 1000)]                            #C
    public decimal Quantity { get; set; }
}
```

#A All the properties are required.
#B The strings must be exactly 3 characters.
#C The quantity can be between 1 and 1000.

There's some basic validation on this model, but during testing you identify a problem: users can enter any three-letter string for the `CurrencyFrom` and `CurrencyTo` properties. Users should only be able to choose a valid currency code, like `"USD"` or `"GBP"`, but someone attacking your application could easily send `"XXX"` or `"£$%"`!

Assuming you support a limited set of currencies, say GBP, USD, EUR, and CAD, you could handle the validation in a few different ways. One way would be to validate the `CurrencyFrom` and `CurrencyTo` values within the Razor Page handler method, after model binding and attribute validation has already occurred.

Another way would be to use a `[RegularExpresssion]` attribute to look for the allowed strings. The approach I'm going to take here is to create a custom `ValidationAttribute`. The goal is to have a custom validation attribute you can apply to the `CurrencyFrom` and `CurrencyTo` attributes, to restrict the range of valid values. This will look something like the following example.

Listing 20.7 Applying custom validation attributes to the binding model

```
public class CurrencyConverterModel
{
    [Required]
    [StringLength(3, MinimumLength = 3)]
    [CurrencyCode("GBP", "USD", "CAD", "EUR")]      #A
    public string CurrencyFrom { get; set; }

    [Required]
    [StringLength(3, MinimumLength = 3)]
    [CurrencyCode("GBP", "USD", "CAD", "EUR")]      #A

    [Required]
    [Range(1, 1000)]
    public decimal Quantity { get; set; }
}
```

#A CurrencyCodeAttribute validates that the property has one of the provided values

Creating a custom validation attribute is simple, as you can start with the `ValidationAttribute` base class and you only have to override a single method. The next

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

listing shows how you could implement `CurrencyCodeAttribute` to ensure that the currency codes provided match the expected values.

Listing 20.8 Custom validation attribute for currency codes

```
public class CurrencyCodeAttribute : ValidationAttribute          #A
{
    private readonly string[] _allowedCodes;                      #B
    public CurrencyCodeAttribute(params string[] allowedCodes)    #B
    {                                                             #B
        _allowedCodes = allowedCodes;                             #B
    }                                                             #B

    protected override ValidationResult IsValid(                  #C
        object value, ValidationContext context)                  #C
    {
        var code = value as string;
        if(code == null || !_allowedCodes.Contains(code))              #D
        {
            return new ValidationResult("Not a valid currency code");    #D
        }
        return ValidationResult.Success;        #E
    }
}
```

#A Derives from ValidationAttribute to ensure your attribute is used during validation
#B The attribute takes in an array of allowed currency codes.
#C The IsValid method is passed the value to validate and a context object.
#D If the value provided isn't a string, is null, or isn't an allowed code, then return an error . . .
#E . . . otherwise, return a success result

Validation occurs in the action filter pipeline after model binding, before the action or Razor Page handler is executed. The validation framework calls `IsValid()` for each instance of `ValidationAttribute` on the model property being validated. The framework passes in `value` (the value of the property being validated) and the `ValidationContext` to each attribute in turn. The context object contains details that you can use to validate the property.

Of particular note is the `ObjectInstance` property. You can use this to access the top-level model being validated when you validate a sub-property. For example, if the `CurrencyFrom` property of the `CurrencyConvertModel` is being validated, you can access the top-level object from the `ValidationAttribute` using:

```
var model = validationContext.ObjectInstance as CurrencyConverterModel;
```

This can be useful if the validity of a property depends on the value of another property of the model. For example, you might want a validation rule that says that GBP is a valid value for `CurrencyTo`, *except* when `CurrencyFrom` is *also* GBP. The `ObjectInstance` makes these sorts of comparison validations easy.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

NOTE **Although using `ObjectInstance` makes it easy to make "model-level" comparisons like these, it reduces the portability of your validation attribute. In this case, you would only be able to use the attribute in the application that defines `CurrencyConverterModel`.**

Within the `IsValid` method, you can cast the `value` provided to the required data type (in this case, `string`) and check against the list of allowed codes. If the code isn't allowed, then the attribute returns a `ValidationResult` with an error message indicating there was a problem. If the code *is* allowed, then `ValidationResult.Success` is returned, and the validation succeeds.

Putting your attribute to the test in figure 20.6 shows that when `CurrencyTo` is an invalid value (`£$%`), the validation for the property fails and an error is added to the `ModelState`. You could do some tidying up of this attribute to let you set a custom message, to allow nulls, or to display the name of the property that's invalid, but the important features are all there.

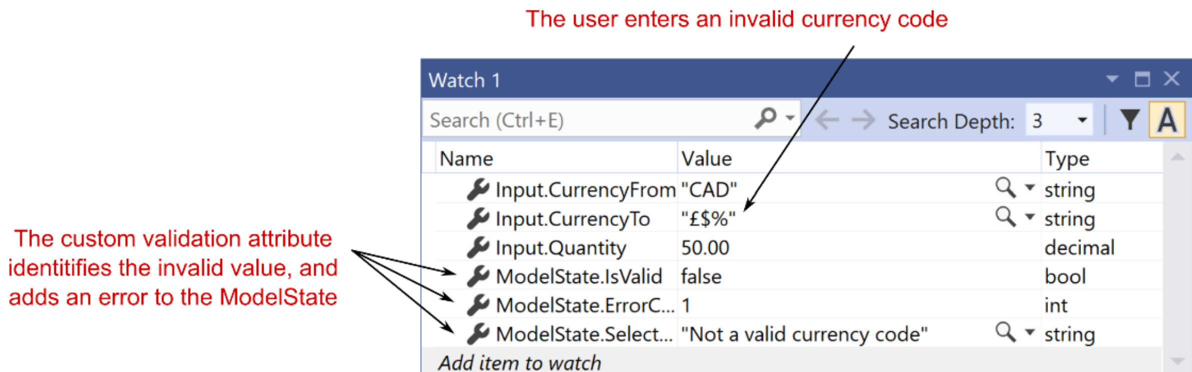


Figure 20.6 The Watch window of Visual Studio showing the result of validation using the custom `ValidationAttribute`. The user has provided an invalid `currencyTo` value, `£$%`. Consequently, `ModelState` isn't valid and contains a single error with the message "Not a valid currency code."

The main feature missing from this custom attribute is client-side validation. You've seen that the attribute works well on the server side, but if the user entered an invalid value they wouldn't be informed until after the invalid value had been sent to the server. That's safe, and so is as much as you *need* to do for security and data-consistency purposes, but client-side validation can improve the user experience by providing immediate feedback.

You can implement client-side validation in several ways, but it's heavily dependent on the JavaScript libraries you use to provide the functionality. Currently ASP.NET Core Razor templates rely on jQuery for client-side validation. See the documentation for an example of creating a jQuery Validation adapter for your attributes: https://docs.microsoft.com/aspnet/core/mvc/models/validation#custom-client-side-validation.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Another improvement to your custom validation attribute would be to load the list of currencies from a DI service, such as an `ICurrencyProvider`. Unfortunately, you can't use constructor DI in your `CurrencyCodeAttribute` as you can only pass *constant* values to the constructor of an `Attribute` in .NET. In chapter 13, we worked around this limitation for filters by using `[TypeFilter]` or `[ServiceFilter]`, but there's no such solution for `ValidationAttribute`.

Instead, for validation attributes, you must use the service locator pattern. As I discussed in chapter 10, this antipattern is best avoided where possible, but unfortunately, it's necessary in this case. Instead of declaring an explicit dependency via a constructor, you must ask the DI container directly for an instance of the required service.

Listing 20.9 shows how you could rewrite listing 20.8 to load the allowed currencies from an instance of `ICurrencyProvider`, instead of hardcoding the allowed values in the attribute's constructor. The attribute calls the `GetService<T>()` method on `ValidationContext` to resolve an instance of `ICurrencyProvider` from the DI container. Note that `ICurrencyProvider` is a hypothetical service, that would need to be registered in your application's `ConfigureServices()` method in Startup.cs.

Listing 20.9 Using the service-locator pattern to access services

```
public class CurrencyCodeAttribute : ValidationAttribute
{
    protected override ValidationResult IsValid(
        object value, ValidationContext context)
    {
        var provider = context.GetService<ICurrencyProvider>();          #A
        var allowedCodes = provider.GetCurrencies();                          #B

        var code = value as string;                                      #C
        if(code == null || !_allowedCodes.Contains(code))                #C
        {                                                                #C
            return new ValidationResult("Not a valid currency code");    #C
        }                                                                #C
        return ValidationResult.Success;                                 #C
    }
}
```

#A Retrieves an instance of ICurrencyProvider directly from the DI container
#B Fetches the currency codes using the provider
#C Validates the property as before

> **TIP** **The generic `GetService<T>` method is an extension method available in the `Microsoft.Extensions.DependencyInjection` namespace. As an alternative, you can use the `GetService(Type type)` method.**

The default `DataAnnotations` validation system can be convenient due to its declarative nature, but this has trade-offs, as shown by the dependency injection problem above. Luckily, you can completely replace the validation system your application uses, as shown in the following section.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

20.4 Replacing the validation framework with FluentValidation

In this section you'll learn how to replace the `DataAnnotations`-based validation framework that's used by default in ASP.NET Core. You'll see the arguments for why you might want to do this and learn how to use a third-party alternative: FluentValidation. This open source project allows you to define the validation requirements of your models separately from the models themselves. This separation can make some types of validation easier and ensures each class in your application has a single responsibility.

Validation is an important part of the model-binding process in ASP.NET Core. Up to now, we've been using `DataAnnotation` attributes applied to properties of your binding model, to define your requirements. In section 20.3 we even created a custom validation attribute.

By default, ASP.NET Core is configured to use these attributes to drive the validation portion of model-binding. But the ASP.NET Core framework is very flexible and allows you to replace whole chunks of the framework if you like. The validation system is one such area that many people choose to replace.

FluentValidation (https://fluentvalidation.net/) is a popular alternative validation framework for ASP.NET Core. It is a mature library, with its roots going back well before ASP.NET Core was conceived of. With FluentValidation you write your validation code *separately* from your binding model code. This gives several advantages:

- You're not restricted to the limitations of `Attributes`, such as the dependency injection problem we had to work around in listing 20.9.
- It's much easier to create validation rules that apply to multiple properties, for example to ensure an `EndDate` property contains a later value than a `StartDate` property. Achieving this with `DataAnnotation` attributes is possible, but difficult.
- It's generally easier to test FluentValidation validators than `DataAnnotation` attributes.
- The validation is strongly-typed, compared to DataAnnotations attributes where it's possible to apply attributes in ways that don't make sense, such as applying an `[EmailAddress]` attribute to an `int` property, for example.
- Separating your validation logic from the model itself arguably better conforms to the single-responsibility-principle (SRP) [101].

That final point is often given as a reason *not* to use FluentValidation: FluentValidation separates a binding model from its validation rules. Some people are happy to accept the limitations of `DataAnnotations` to keep the model and validation rules together. Before I show how to add FluentValidation to your application, let's see what FluentValidation validators look like.

---

[101] **The SRP is one of the SOLID design principles: https://en.wikipedia.org/wiki/SOLID.**

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

20.4.1 Comparing FluentValidation to DataAnnotation attributes

To better understand the difference between the `DataAnnotations` approach and FluentValidation, we'll convert the binding models from section 20.3 to use FluentValidation. The following listing shows what the binding model from listing 20.7 would look like when used with FluentValidation. It is structurally identical but has no validation attributes.

Listing 20.10 Currency converter initial binding model for use with FluentValidation

```
public class CurrencyConverterModel
{
    public string CurrencyFrom { get; set; }
    public string CurrencyTo { get; set; }
    public decimal Quantity { get; set; }
}
```

In FluentValidation you define your validation rules in a separate class, with a class per model to be validated. Typically, these derive from the `AbstractValidator<>` base class, which provides a set of extension methods for defining your validation rules.

The following listing shows a validator for the `CurrencyConverterModel`, which matches the validations added using attributes in listing 20.7. You create a set of validation rules for a property by calling `RuleFor()`, and chaining method calls such as `NotEmpty()` from it. This style of method chaining is called a "fluent" interface, hence the name.

Listing 20.11 Currency converter initial binding model for use with FluentValidation

```
public class CurrencyConverterModelValidator                    #A
    : AbstractValidator<CurrencyConverterModel>                 #A
{
    private readonly string[] _allowedValues                    #B
        = new []{ "GBP", "USD", "CAD", "EUR" };                 #B

    public InputValidator()                                     #C
    {
        RuleFor(x => x.CurrencyFrom)                            #D
            .NotEmpty()                                         #E
            .Length(3)                                          #E
            .Must(value => _allowedValues.Contains(value))      #F
            .WithMessage("Not a valid currency code");          #F

        RuleFor(x => x.CurrencyTo)
            .NotEmpty()
            .Length(3)
            .Must(value => _allowedValues.Contains(value))
            .WithMessage("Not a valid currency code");

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

#A The validator inherits from AbstractValidator
#B Defines the static list of currency codes that are supported

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

#C You define validation rules in the validator's constructor
#D RuleFor is used to add a new validation rule. The lambda syntax allows for strong typing
#E There are equivalent rules for common DataAnnotation validation attributes
#F You can easily add custom validation rules, without having to create separate classes
#G 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>                        #A
        MustBeCurrencyCode<T>(                                          #A
            this IRuleBuilder<T, string> ruleBuilder)                   #A
    {
        return ruleBuilder                                              #B
            .Must(value => _allowedValues.Contains(value))              #B
            .WithMessage("Not a valid currency code");                  #B
    }

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

#A Creates an extension method that can be chained from RuleFor() for string properties
#B Applies the same validation logic as before
#C 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:

```
RuleFor(x => x.CurrencyTo)
    .NotEmpty()
    .Length(3)
    .MustBeCurrencyCode();
```

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Another advantage of the FluentValidation approach of using standalone validation classes, is that they are created using dependency injection, so you can inject services into them. As an example, consider the `[CurrencyCode]` validation attribute from listing 20.9 which used a service, `ICurrencyProvider` from the DI container. This requires using service location to obtain an instance of `ICurrencyProvider` using an injected "context" object.

With the FluentValidation library, you can just inject the `ICurrencyProvider` directly into your validator, as shown in the following listing. This requires fewer gymnastics to get the desired functionality and makes your validator's dependencies explicit.

Listing 20.13 Currency converter validator using dependency injection

```
public class CurrencyConverterModelValidator
    : AbstractValidator< CurrencyConverterModel>
{
    public InputValidator(ICurrencyProvider provider)         #A
    {
        RuleFor(x => x.CurrencyFrom)
            .NotEmpty()
            .Length(3)
            .Must(value => provider                           #B
                .GetCurrencies()                              #B
                .Contains(value))                             #B
            .WithMessage("Not a valid currency code");

        RuleFor(x => x.CurrencyTo)
            .NotEmpty()
            .Length(3)
            .MustBeCurrencyCode(provider.GetCurrencies());    #C

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

#A Injecting the service using standard constructor dependency injection
#B Using the injected service in a Must() rule
#C Using the injected service with an extension method

The final feature I'll show demonstrates how much easier it is to write validators that span multiple properties with FluentValidation. For example, imagine we want to validate that the value of `CurrencyTo` is different to `CurrencyFrom`. Using FluentValidation you can implement this with an overload of `Must()`, which provides both the model and the property being validated, as shown in the following listing.

Listing 20.14 Using `Must()` to validate that two properties are different

```
RuleFor(x => x.CurrencyTo)                                      #A
    .NotEmpty()
    .Length(3)
    .MustBeCurrencyCode()
    .Must((InputModel model, string currencyTo)                 #B
        => currencyTo != model.CurrencyFrom)                    #C
```


Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
    .WithMessage("Cannot convert currency to itself");        #D
```

#A The error message will be associated with the CurrencyTo property
#B The Must function passes the top-level model being validated and the current property
#C Perform the validation—the currencies must be different
#D Use the provided message as the error message

Creating a validator like this is certainly possible with `DataAnnotation` attributes, but it requires far more ceremony than the FluentValidation equivalent, and is generally harder to test. FluentValidation has many more features for making it easier to write and test your validators too, for example:

- *Complex property validations*. Validators can be applied to complex types, as well as primitive types like `string` and `int` shown here in this section.
- *Custom property validators*. In addition to simple extension methods, you can create your own property validators for complex validation scenarios.
- *Collection rules*. When types contain collections, such as `List<T>`, you can apply validation to each item in the list, as well as the overall collection.
- *RuleSets*. Create multiple collections of rules that can be applied to an object in different circumstances. These can be especially useful if you're using FluentValidation in additional areas of your application.
- *Client-side validation*. FluentValidation is a server-side framework, but it emits the same attributes as `DataAnnotation` attributes to enable client-side validation using jQuery.

There are many more features in addition to these, so be sure to browse the documentation at https://docs.fluentvalidation.net/ for details. In the next section you'll see how to add FluentValidation to your ASP.NET Core application.

20.4.2 Adding FluentValidation to your application

Replacing the whole validation system of ASP.NET Core sounds like a big step, but the FluentValidation library makes it easy to add to your application. Simply follow these steps:

1. Install the FluentValidation.AspNetCore NuGet package using Visual Studio's NuGet package manager, using the CLI by running `dotnet add package FluentValidation.AspNetCore` or by adding a `<PackageReference>` to your csproj file:

```
<PackageReference Include="FluentValidation.AspNetCore" Version="9.0.1" />
```

2. Configure the FluentValidation library in the `ConfigureServices` method of your `Startup` class by calling `AddFluentValidation()`. You can further configure the library, as shown in listing 20.15 below.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

3. Register your validators (such as the `CurrencyConverterModelValidator` from listing 20.13) with the DI container. These can be registered manually, using any scope you choose, for example:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages()
        .AddFluentValidation();

    services.AddScoped<
        IValidator<CurrencyConverterModelValidator>,
        CurrencyConverterModelValidator>();
}
```

Alternatively, you can allow FluentValidation to automatically register all your validators using the options shown in listing 20.15.

For such a mature library, FluentValidation has relatively few configuration options to decipher. The following listing shows some of the options available, in particular it shows how to automatically register all the custom validators in your application, and how to disable `DataAnnotation` validation entirely.

Listing 20.15 Configuring FluentValidation in an ASP.NET Core application

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddRazorPages()
    .AddFluentValidation(opts =>
    {
      opts.RegisterValidatorsFromAssemblyContaining<Startup>();    #A
      opts.ImplicitlyValidateChildProperties = true;               #B
      opts.LocalizationEnabled = false;                            #C
      opts.RunDefaultMvcValidationAfterFluentValidationExecutes    #D
        = false;                                                   #D
    });
}
```

#A Instead of manually registering validators, FluentValidation can auto-register them for you
#B Ensure that complex (nested) properties are validated, not just "top-level" properties
#C FluentValidation has full localization support, but you can disable it if you don't need it
#D Setting to false disables DataAnnotation validation completely for model binding

It's important to understand that final point. If you don't set it to `false`, ASP.NET Core will run validation with *both* `DataAnnotations` and with FluentValidation. That may be useful if you're in the process of migrating from one system to the other, but otherwise I recommend disabling it. Having your validation split between both places seems like it would be the worst of both worlds!

One final thing to consider is where to put your validators in your solution. There are no technical requirements on this—if you've registered your validator with the DI container it will be used correctly—so the choice is up to you. Personally, I prefer to place validators close to the models they're validating.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

For Razor Pages binding model validators, I create the validator as a nested class of the `PageModel`, in the same place as I create the `InputModel`, as I described in chapter 6. That gives a class hierarchy in the Razor Page similar to the following:

```
public class IndexPage : PageModel
{
    public class InputModel { }
    public class InputModelValidator: AbstractValidator<InputModel> { }
}
```

That's just my preference of course, you're free to adopt another approach if you prefer!

That brings us to the end of this chapter on custom Razor Pages components. When combined with the components in the previous chapter, you've got a great base for extending your ASP.NET Core applications to meet your needs. It's a testament to ASP.NET Core's design that you can swap out whole sections like the Validation framework entirely. If you don't like how some part of the framework works, see if someone has written an alternative!

20.5 Summary

- With Tag Helpers, you can bind your data model to HTML elements, making it easier to generate dynamic HTML. Tag Helpers can customize the elements they're attached to, add additional attributes, and customize how they're rendered to HTML. This can greatly reduce the amount of markup you need to write.
- The name of a Tag Helper class dictates the name of the element in the Razor templates, so the `SystemInfoTagHelper` corresponds to the `<system-info>` element. You can choose a different element name by adding the `[HtmlTargetElement]` attribute to your Tag Helper.
- You can set properties on your Tag Helper object from Razor syntax by decorating the property with an `[HtmlAttributeName("name")]` attribute and providing a name. You can set these properties from Razor using HTML attributes, `<system-info name="value">`, for example.
- The `TagHelperOutput` parameter passed to the `Process` or `ProcessAsync` methods control the HTML that's rendered to the page. You can set the element type with the `TagName` property, and set the inner content using `Content.SetContent()` or `Content.SetHtmlContent()`.
- You can prevent inner Tag Helper content being processed by calling `SupressOutput()` and you can remove the element entirely by setting `TagName=null`. This is useful if you only want to conditionally render elements to the response.
- You can retrieve the contents of a Tag Helper by calling `GetChildContentAsync()` on the `TagHelperOutput` parameter. You can then Render this content to a `string` by calling `GetContent()`. This will render any Razor expressions and Tag Helpers to HTML, allowing you to manipulate the contents.
- View components are like partial views, but they allow you to use complex business and rendering logic. You can use them for sections of a page, such as the shopping

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

cart, a dynamic navigation menu, or suggested articles.

- Create a view component by deriving from the `ViewComponent` base class, and implement `InvokeAsync()`. You can pass parameters to this function from the Razor view template using HTML attributes, in a similar way to Tag Helpers.
- View components can use DI, have access to the `HttpContext`, and can render partial views. The partial views should be stored in the Pages/Shared/Components/<Name>/ folder, where Name is the name of the view component. If not specified, view components will look for a default view named Default.cshtml.
- You can create a custom `DataAnnotations` attribute by deriving from `ValidationAttribute` and overriding the `IsValid` method. You can use this to decorate your binding model properties and perform arbitrary validation.
- You can't use constructor DI with custom validation attributes. If the validation attribute needs access to services from the DI container, you must use the service locator pattern to load them from the validation context, using the `GetService<T>` method.
- FluentValidation is an alternative validation system that can replace the default `DataAnnotations` validation system. It is not based on attributes, which makes it easier to write custom validations for your validation rules and makes those rules easier to test.
- To create a validator for a model, create a class derived from `AbstractValidator<>` and call `RuleFor<>()` in the constructor to add validation rules. You can chain multiple requirements on `RuleFor<>()` in the same way that you could add multiple `DataAnnotation` attributes to a model.
- If you need to create a custom validation rule, you can use the `Must()` method to specify a predicate. If you wish to re-use the validation rule across multiple models, encapsulate the rule as an extension method, to reduce duplication.
- To add FluentValidation to your application, install the FluentValidation.AspNetCore NuGet package, call `AddFluentValidation()` after your call to `AddRazorPages()` or `AddControllers()`, and register your validators with the DI container. This will add FluentValidation validations in addition to the built-in `DataAnnotations` system.
- To remove the `DataAnnotations` validation system, and use FluentValidation only, set the `RunDefaultMvcValidationAfterFluentValidationExecutes` option to `false` in your call to `AddFluentValidation()`. Favor this approach where possible, to avoid receiving validation methods from two different systems.
- You can allow FluentValidation to automatically discover and register all the validators in your application by calling `RegisterValidatorsFromAssemblyContaining<T>()`, where `T` is a type in the assembly to scan. This means you don't have to register each validator in your application with the DI container individually.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

21 Calling remote APIs with IHttpClientFactory

This chapter covers

- **Problems caused by using `HttpClient` incorrectly to call HTTP APIs**
- **Using `IHttpClientFactory` to manage `HttpClient` lifetimes**
- **Encapsulating configuration and handling transient errors with `IHttpClientFactory`**

So far in this book we've focused on creating web pages and exposing APIs for others to consume. Whether that's customers browsing a Razor Pages application, or client-side SPAs and mobile apps consuming your APIs, we've been writing the APIs for others to consume.

However, it's very common for your application to interact with third-party-services by consuming *their* APIs. For example, an eCommerce site needs to take payments, send email and SMS messages, and retrieve exchange rates from a third-party service. The most common approach for interacting with services is using HTTP. So far in this book we've looked at how you can *expose* HTTP services, using API controllers, but we haven't looked at how you can *consume* HTTP services.

In section 21.1, you'll learn the best way to interact with HTTP services using `HttpClient`. If you have any experience with C#, it's very likely you've used this class to send HTTP requests, but there are two "gotchas" to think about, otherwise your app could run into difficulties.

`IHttpClientFactory` was introduced in .NET Core 2.1; it makes creating and managing `HttpClient` instances easier and avoids the common pitfalls. In section 21.2 you'll learn how `IHttpClientFactory` achieves this by managing the `HttpClient` handler pipeline. You'll learn how to create *named clients* to centralize the configuration for calling remote APIs, and how to use *typed clients* to encapsulate the remote service's behavior.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Network glitches are a fact of life when you're working with HTTP APIs, so it's important for you to handle them gracefully. In section 21.3 you'll learn how to use the open source resilience and fault-tolerance library Polly to handle common transient errors using simple retries, with the possibility for more complex policies.

Finally, in section 21.4 you'll see how you can create your own custom `HttpMessageHandler` handler managed by `IHttpClientFactory`. You can use custom handlers to implement cross cutting concerns such as logging, metrics, or authentication, where a function needs to execute every time you call an HTTP API. In section 21.4 you'll see how to create a handler that automatically adds an API key to all outgoing requests to an API.

To misquote John Donne, "no app is an island", and the most common way of interacting with other apps and services is over HTTP. In .NET Core, that means using `HttpClient`.

21.1 Calling HTTP APIs: the problem with HttpClient

In this section you'll learn how to use `HttpClient` to call HTTP APIs. I focus on two common pitfalls in using `HttpClient`, socket exhaustion and DNS rotation problems, and show why they occur. In section 21.2 you'll see how to avoid these issues by using `IHttpClientFactory`.

It's very common for an application to need to interact with other services to fulfill its duty. Take a typical ecommerce store for example. In even the most basic version of the application, you will likely need to send emails and take payments using credit cards or other services. You *could* try and build that functionality yourself, but it probably wouldn't be worth the effort.

Instead, it makes far more sense to delegate those responsibilities to third-party services which specialize in that functionality. Whichever service you use, they will almost certainly expose an HTTP API for interacting with the service. For many services, that will be the *only* way.

HTTP vs gRPC vs GraphQL

There are many ways to interact with third-party services, but HTTP RESTful services are still the king, decades after HTTP was first proposed. Every platform and programming language you can think of includes support for making HTTP requests and handling responses. That ubiquity makes it the go-to option for most services.

Despite their ubiquity, RESTful services are not perfect. They are relatively verbose, which means more data ends up being sent and received than some other protocols. It can also be difficult to evolve RESTful APIs after you have deployed them. These limitations have spurred interest in two alternative protocols in particular: gRPC and GraphQL.

gRPC is intended to be an efficient mechanism for server-to-server communication. It builds on top of HTTP/2, but typically provides much higher performance than traditional RESTful APIs. gRPC support was added in .NET Core 3.0 and is receiving many performance and feature updates. For a comprehensive view of .NET support, see the documentation at https://docs.microsoft.com/aspnet/core/grpc.

While gRPC is primarily intended for server-to-server communication, GraphQL is best used to provide evolvable APIs to mobile and SPA apps. It has become very popular among front-end developers, as it can reduce the friction involved in deploying and using new APIs. For details, I recommend *GraphQL in Action* by Samer Buna (Manning, 2020).

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Despite the benefits and improvements gRPC and GraphQL can bring, RESTful HTTP services are here to stay for the foreseeable future, so it's worth making sure you understand how to use them with `HttpClient`.

In .NET we use the `HttpClient` class for calling HTTP APIs. You can use it to make HTTP calls to APIs, providing all the headers and body to send in a request, and reading the response headers and data you get back. Unfortunately, it's hard to use correctly, and even when you do, it has limitations.

The source of the difficultly with `HttpClient` stems partly from the fact it implements the `IDisposable` interface. In general, when you use a class that implements `IDisposable`, you should wrap the class with a `using` statement whenever you create a new instance. This ensures that unmanaged resources used by the type are cleaned-up when the class is removed.

```
using (var myInstance = new MyDisposableClass())
{
    // use myInstance
}
```

That might lead you to think that the correct way to create an `HttpClient` is shown in the following listing. This shows a simple example where an API controller calls an external API to fetch the latest currency exchange rates and returns them as the response.

> **WARNING** Do not use `HttpClient` like what's shown in listing 21.1. Using it this way could cause your application to become unstable, as you'll see shortly.

Listing 21.1 The incorrect way to use `HttpClient`

```
[ApiController]
public class ValuesController : ControllerBase
{
    [HttpGet("values")]
    public async Task<string> GetRates()
    {
        using (HttpClient client = new HttpClient())              #A
        {
            client.BaseAddress                                    #B
                = new Uri("https://api.exchangeratesapi.io");     #B

            var response = await client.GetAsync("latest");       #C

            response.EnsureSuccessStatusCode();                   #D
            return await response.Content.ReadAsStringAsync();    #E
        }
    }
}
```

#A Wrapping the HttpClient in a using statement means it is disposed at the end of the using block
#B Configure the base URL used to make requests using the HttpClient
#C Make a GET request to the exchange rates API
#D Throws an exception if the request was not successful

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

#E Read the result as a string and return it from the action method

However, `HttpClient` is special, and you *shouldn't* use it like this! The problem is primarily due to the way the underlying protocol implementation works. Whenever your computer needs to send a request to an HTTP server, you must create a *connection* between your computer and the server. To create a connection, your computer opens a port, which has a random number between 0 and 65535, and connects to the HTTP server's IP address and port, as shown in figure 21.1. Your computer can then send HTTP requests to the server.

DEFINITION The combination of IP address and port is called a *socket*.

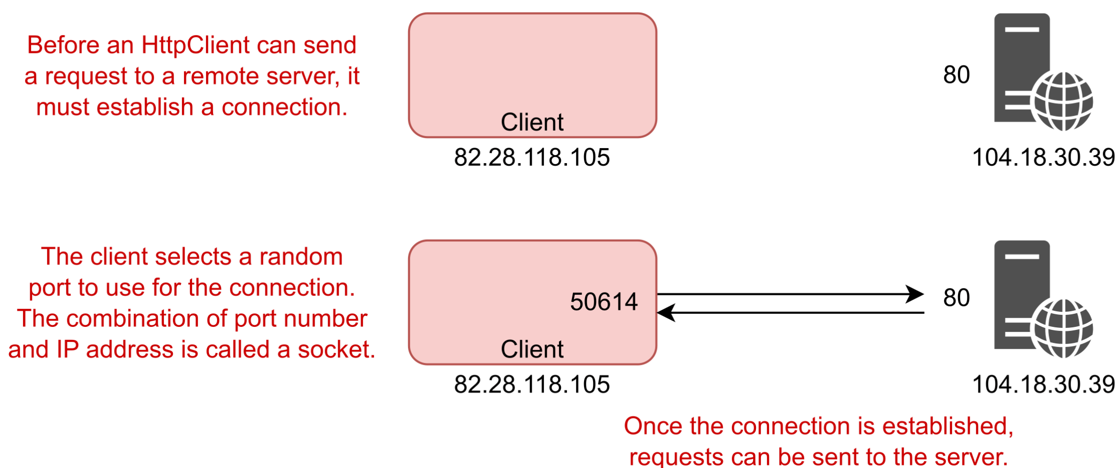


Figure 21.1 To create a connection, a client selects a random port and connects to the HTTP server's port and IP address. The combination of a port number and IP address is called a socket. The client can then send HTTP requests to the server.

The main problem with the `using` statement and `HttpClient` is that it can lead to a problem called *socket exhaustion*, as illustrated in figure 21.2. This happens when all the ports on your computer have been used up making other HTTP connections, so your computer can't make any more requests. At that point, your application will hang, waiting for a socket to become free. A very bad experience!

Given that I said there are 65536 different port numbers, you might think that's an unlikely situation. It's true, you will likely only run into this problem on a server that is making a lot of connections, but it's not as rare as you might think.

The problem is that when you dispose an `HttpClient`, *it doesn't close the socket immediately*. The design of the TCP/IP protocol used for HTTP requests means that after trying to close a connection, the connection moves to a state called TIME_WAIT. The connection then waits for a specific period (240 seconds on Windows) before closing the socket completely.

©Manning Publications Co. To comment go to liveBook

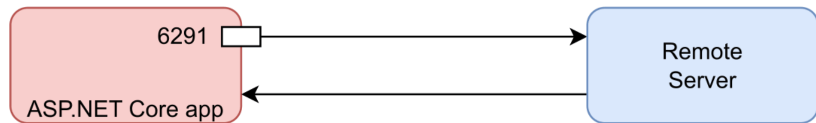
Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Until the TIME_WAIT period has elapsed, you can't reuse the socket in another `HttpClient` to make HTTP requests. If you're making a lot of requests, that can quickly lead to socket exhaustion, as shown in figure 21.2

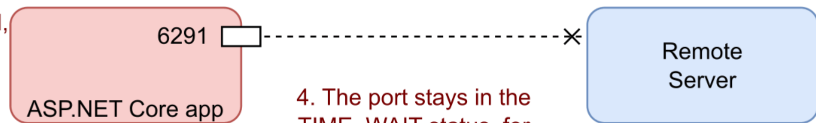
TIP **You can view the state of active ports/sockets in Windows and Linux by running the command `netstat` from the command line or a terminal window.**

1. The app creates a new HttpClient instance and initiates a request to the remote server.

2. A random port is assigned, 6291, and a connection is established to the remote server.

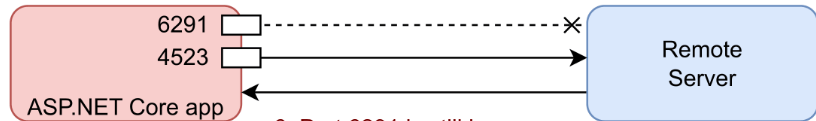


3 Once response is received, the application disposes the HttpClient, and starts to close the connection.



4. The port stays in the TIME_WAIT status, for 240 seconds.

5. When the application wants to send another request, it creates a new HttpClient.



6. Port 6291 is still in use, so another port must be used, port 4523 for example.

7. With sufficient numbers of requests, the machine running your app can run out of ports as they're all stuck, in TIME_WAIT, and you can no longer send or receive new requests.

Command Prompt (light) - netstat

```
TCP    192.168.8.31:54052    172.67.157.158:https    TIME_WAIT
TCP    192.168.8.31:54053    172.67.157.158:https    TIME_WAIT
TCP    192.168.8.31:54054    172.67.157.158:https    TIME_WAIT
TCP    192.168.8.31:54055    172.67.157.158:https    TIME_WAIT
TCP    192.168.8.31:54056    172.67.157.158:https    TIME_WAIT
TCP    192.168.8.31:54057    172.67.157.158:https    TIME_WAIT
TCP    192.168.8.31:54058    172.67.157.158:https    TIME_WAIT
TCP    192.168.8.31:54059    172.67.157.158:https    TIME_WAIT
TCP    192.168.8.31:54060    172.67.157.158:https    TIME_WAIT
TCP    192.168.8.31:54061    172.67.157.158:https    TIME_WAIT
TCP    192.168.8.31:54062    172.67.157.158:https    TIME_WAIT
TCP    192.168.8.31:54063    172.67.157.158:https    TIME_WAIT
TCP    192.168.8.31:54064    172.67.157.158:https    TIME_WAIT
TCP    192.168.8.31:54065    172.67.157.158:https    TIME_WAIT
TCP    192.168.8.31:54066    172.67.157.158:https    TIME_WAIT
TCP    192.168.8.31:54067    172.67.157.158:https    TIME_WAIT
```

Figure 21.2 Disposing of `HttpClient` can lead to socket exhaustion. Each new connection requires the

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

operating system to assign a new socket. Closing a socket doesn't make it available until the TIME_WAIT period of 240 seconds has elapsed. Eventually you can run out of sockets, at which point you can't make any outgoing HTTP requests.

Instead of disposing `HttpClient`, the general advice (before `IHttpClientFactory` was introduced in .NET Core 2.1) was to use a single instance of the `HttpClient`, as shown in the following listing.

Listing 21.2 Using a singleton `HttpClient` to avoid socket exhaustion

```
[ApiController]
public class ValuesController : ControllerBase
{
    private static readonly HttpClient _client = new HttpClient       #A
    {                                                                 #A
        BaseAddress = new Uri("https://api.exchangeratesapi.io")      #A
    };                                                                #A

    [HttpGet("values")]
    public async Task<string> GetRates()
    {
        var response = await _client.GetAsync("latest");          #B

        response.EnsureSuccessStatusCode();
        return await response.Content.ReadAsStringAsync();
    }
}
```

#A A single instance of the HttpClient is created and stored as a static field
#B Multiple requests use the same instance of HttpClient

This solves the problem of socket exhaustion. As you're not disposing the `HttpClient`, the socket is not disposed, so you can re-use the same port for multiple requests. No matter how many times you call `GetRates()` in the example above, you will only use a single socket. Problem solved!

Unfortunately, this introduces a different problem, primarily around DNS. DNS is how the friendly host names we use, such as manning.com, are converted into the IP addresses that computers need. When a new connection is required, the `HttpClient` first checks the DNS record for a host to find the IP address, and then makes the connection. For subsequent requests, the connection is already established, so it doesn't make another DNS call.

For singleton `HttpClient` instances this can be a problem, as the `HttpClient` won't detect DNS changes. DNS is often used in cloud environments for load balancing to do graceful rollouts of deployments.[102] If the DNS record of a service you're calling changes during the

[102] Azure Traffic Manager, for example, uses DNS to route requests. You can read more about how it works at https://azure.microsoft.com/en-gb/services/traffic-manager/.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

lifetime of your application, a singleton `HttpClient` will keep calling the old service, as shown in figure 21.3.

> **NOTE** `HttpClient` won't respect a DNS change while the original connection exists. If the original connection is closed, for example if the original server goes offline, then it will respect the DNS change as it must establish a new connection.

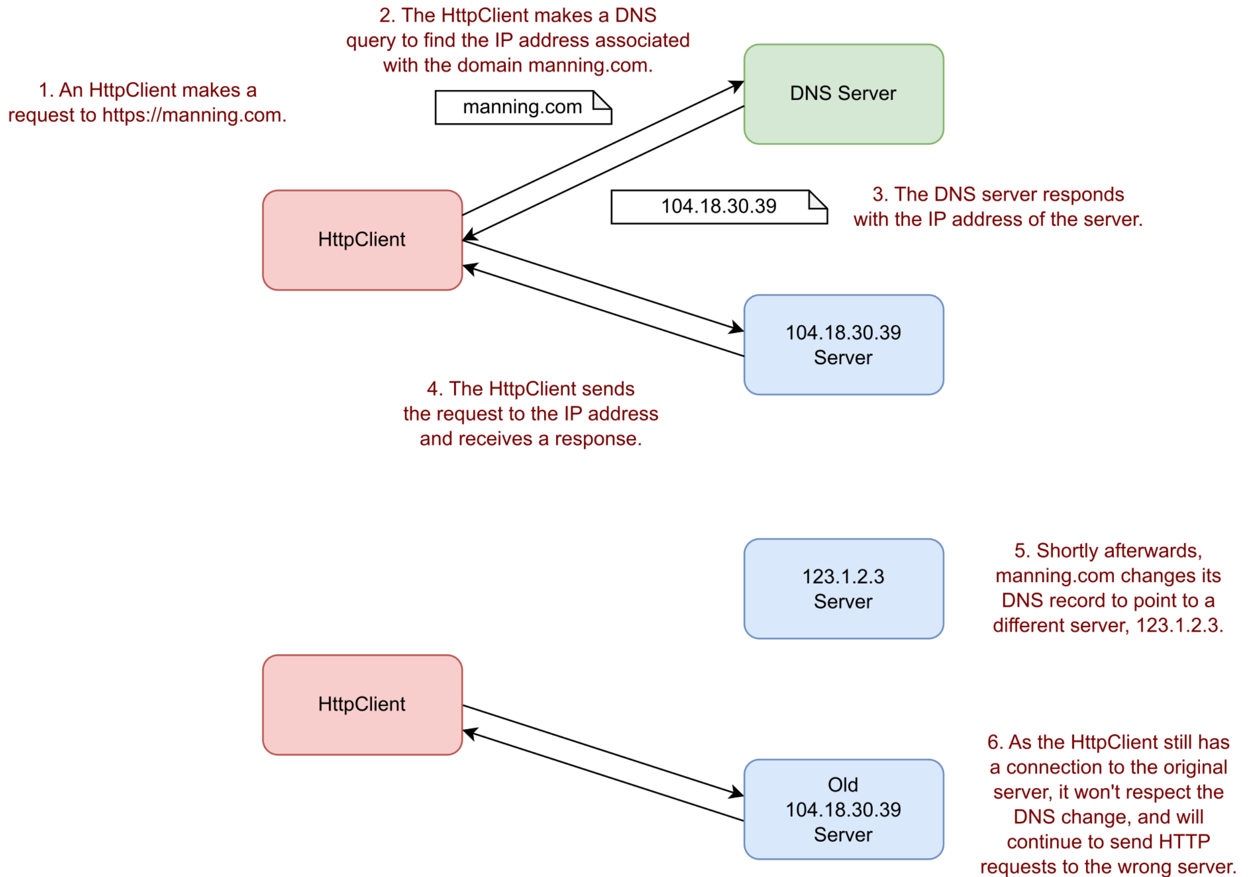


Figure 21.3. `HttpClient` **does a DNS lookup before establishing a connection, to determine the IP address associated with a hostname. If the DNS record for a hostname changes, a singleton** `HttpClient` **will not detect it, and will continue sending requests to the original server it connected to.**

It seems like you're damned if you, and you're damned if you don't! Luckily, `IHttpClientFactory` can take care of all this for you.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

21.2 Creating HttpClients with IHttpClientFactory

In this section you'll learn how you can use `IHttpClientFactory` to avoid the common pitfalls of `HttpClient`. I'll show several patterns you can use to create HttpClients:

- Using the `CreateClient()` as a drop-in replacement for `HttpClient`
- Using "named clients" centralize the configuration of an `HttpClient` used to call a specific third-party API.
- Using "typed clients" to encapsulate the interaction with a third-party API for easier consumption by your code.

`IHttpClientFactory` was introduced in .NET Core 2.1. It makes it easier to create `HttpClient` instances *correctly*, instead of relying on either of the faulty approaches I showed in section 21.1. It also makes it easier to configure multiple `HttpClients` and allows you to create a "middleware pipeline" for outgoing requests.

Before we look at how `IHttpClientFactory` achieves all that, we will look a little closer at how `HttpClient` works under the hood.

21.2.1 Using IHttpClientFactory to manage HttpClientHandler lifetime

In this section I describe the handler pipeline used by `HttpClient`. You'll see how `IHttpClientFactory` manages the lifetime of the handler pipeline and how this enables the factory to avoid both socket exhaustion and DNS issues.

The `HttpClient` class you typically use to make HTTP requests is responsible for orchestrating requests, but it isn't responsible for making the raw connection itself. Instead, the `HttpClient` calls into a pipeline of `HttpMessageHandler`, at the end of which is an `HttpClientHandler`, which makes the actual connection, and sends the HTTP request, as shown in figure 21.4.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

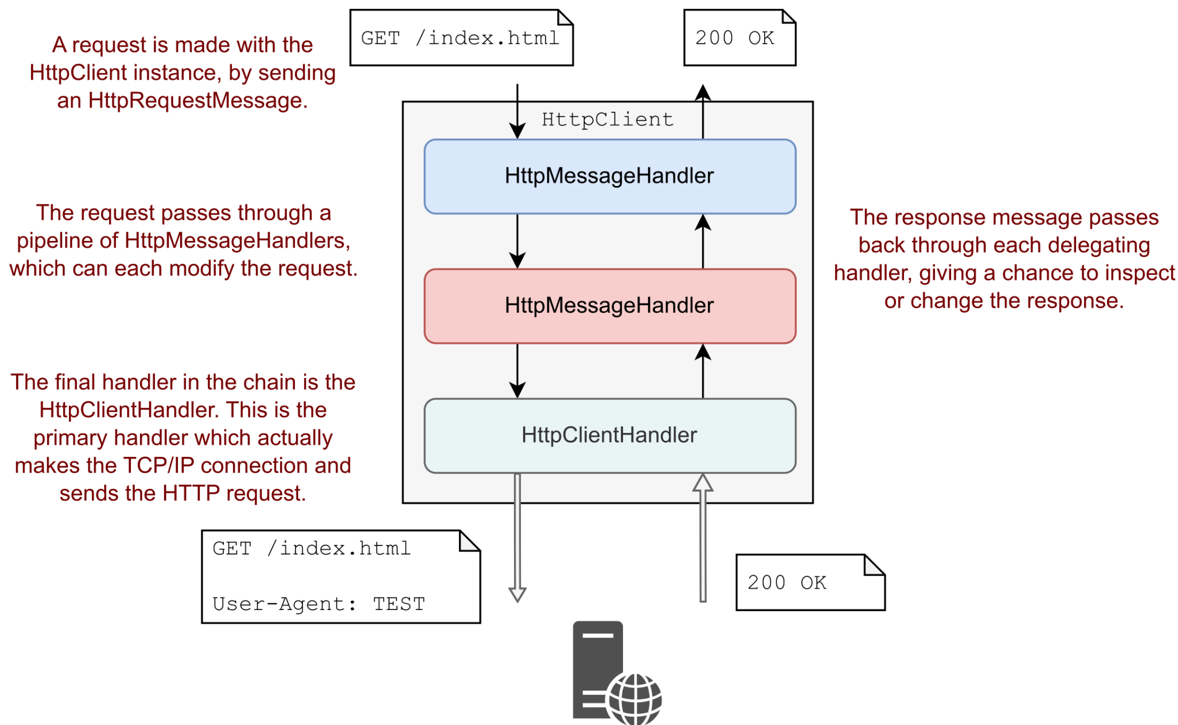


Figure 21.4. Each `HttpClient` contains a pipeline of `HttpMessageHandlers`. The final handler is an `HttpClientHandler`, which makes the connection to the remote server and sends the HTTP request. This configuration is similar to the ASP.NET Core middleware pipeline and allows you to make cross-cutting adjustments to outgoing requests.

This configuration is very reminiscent of the middleware pipeline used by ASP.NET Core applications, but this is an *outbound* pipeline. When an `HttpClient` makes a request, each handler gets a chance to modify the request, before the final `HttpClientHandler` makes the real HTTP request. Each handler in turn then gets a chance to view the response after it's received.

> **TIP** You'll see an example of using this handler pipeline for cross-cutting concerns in section 21.2.4 when we add a transient error handler.

The issues of socket exhaustion and DNS I described in section 21.1 are both related to the disposal of the `HttpClientHandler` at the end of the handler pipeline. By default, when you dispose an `HttpClient`, you dispose the handler pipeline too. `IHttpClientFactory` separates the lifetime of the `HttpClient` from the underlying `HttpClientHandler`.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Separating the lifetime of these two components enables the `IHttpClientFactory` to solve the problems of socket exhaustion and DNS rotation. It achieves this in two ways:

- *Creating a pool of available handlers*. Socket exhaustion occurs when you dispose an `HttpClientHandler`, due to the TIME_WAIT problem described previously. `IHttpClientFactory` solves this by creating a "pool" of handlers.

  `IHttpClientFactory` maintains an "active" handler, that is used to create all `HttpClient`s for two minutes. When the `HttpClient` is disposed, the underlying handler *isn't* disposed. As the handler isn't disposed, the connection isn't closed, and socket exhaustion isn't a problem.

- *Periodically disposing handlers*. Sharing handler pipelines solves the socket exhaustion problem, but it doesn't solve the DNS issue. To work around this, the `IHttpClientFactory` periodically (every two minutes) creates a new "active" `HttpClientHandler` that is used for each `HttpClient` created subsequently. As these `HttpClients` are using a new handler, they make a new TCP/IP connection, and so DNS changes are respected.

  `IHttpClientFactory` disposes "expired" handlers periodically in the background—once they are no longer used by an `HttpClient`. This ensures there are only ever a limited number of connections in use by your application's `HttpClient`s.[103]

Rotating handlers with `IHttpClientFactory` solves both the issues we've discussed. Another bonus is that it's easy to replace existing usages of `HttpClient` with `IHttpClientFactory`.

`IHttpClientFactory` is included by default in ASP.NET Core, you just need to add it to your application's services in the `ConfigureServices()` method of Startup.cs:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHttpClient()
}
```

This registers the `IHttpClientFactory` as a singleton in your application, so you can inject it into any other service. For example, the following listing shows how you can replace the `HttpClient` approach from listing 21.2 with a version that uses `IHttpClientFactory`.

Listing 21.3 Using `IHttpClientFactory` to create an `HttpClient`

```
[ApiController]
public class ValuesController : ControllerBase
{
    private readonly IHttpClientFactory _factory;          #A
    public ValuesController(IHttpClientFactory factory)    #A
    {
```

[103] I looked in depth about how IHttpClientFactory achieves this rotation. This is a detailed post, but may be of interest to those who like to know how things are implemented behind the scenes: https://andrewlock.net/exporing-the-code-behind-ihttpclientfactory/.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
        _factory = factory;
    }

    [HttpGet("values")]
    public async Task<string> GetRates()
    {
        HttpClient client = _factory.CreateClient();            #B

        client.BaseAddress =                                   #C
            new Uri("https://api.exchangeratesapi.io");        #C
        client.DefaultRequestHeaders.Add(                      #C
            HeaderNames.UserAgent, "ExchangeRateViewer");      #C

        var response = await client.GetAsync("latest");        #D

        response.EnsureSuccessStatusCode();                    #D
        return await response.Content.ReadAsStringAsync();     #D
    }
}
```

#A Inject the IHttpClientFactory using DI
#B Create an HttpClient instance with an HttpClientHandler managed by the factory
#C Configure the HttpClient for calling the API as before
#D Use the HttpClient in exactly the same way as you would otherwise

The immediate benefit of using `IHttpClientFactory` in this way is efficient socket and DNS handling. Minimal changes should be required to take advantage of this pattern, as the bulk of your code stays the same. This makes it a good option if you're refactoring an existing app.

21.2.2 Configuring named clients at registration time

In this section you'll learn how to use the "named client" pattern with `IHttpClientFactory`. This pattern encapsulates the logic for calling a third-party API in a single location, making it easier to use the `HttpClient` in your consuming code.

Using `IHttpClientFactory` solves the technical issues I described in section 21.1, but the code in listing 21.3 is still pretty messy in my eyes. That's primarily because you must configure the `HttpClient` to point to your service before you use it. If you use need to create an `HttpClient` to call the API in more than one place in your application, you must *configure* it more than one place too.

`IHttpClientFactory` provides a convenient solution to this problem by allowing you to centrally configure *named clients*. These clients have a `string` name, and a configuration function which runs whenever an instance of the named client is requested. You can define multiple configuration functions which run in sequence to configure your new `HttpClient`.

For example, the following listing shows how to register a named client called `"rates"`. This client is configured with the correct `BaseAddress` and sets default headers that are to be sent with each outbound request.

Listing 21.4 Configuring a named client using `IHttpClientFactory` in Startup.cs

```
public void ConfigureServices(IServiceCollection services)
```


Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
{
    services.AddHttpClient("rates", (HttpClient client) =>           #A
    {
        client.BaseAddress =                                         #B
            new Uri("https://api.exchangeratesapi.io");              #B
        client.DefaultRequestHeaders.Add(                            #B
            HeaderNames.UserAgent, "ExchangeRateViewer");            #B
    })
    .ConfigureHttpClient((HttpClient client) => {})                  #C
    .ConfigureHttpClient(
        (IServiceProvider provider, HttpClient client) => {});       #D
}
```

#A Provide a name for the client, and a configuration function
#B The configuration function runs every time the named HttpClient is requested
#C You can add additional configuration functions for the named client, which run in sequence
#D Additional overloads exist that allow access to the DI container when creating a named client

Once you have configured this named client, you can create it from an `IHttpClientFactory` instance using the name of the client, `"rates"`. The following listing shows how you could update listing 21.3 to use the named client configured in listing 21.4.

Listing 21.5 Using `IHttpClientFactory` to create a named `HttpClient`

```
[ApiController]
public class ValuesController : ControllerBase
{
    private readonly IHttpClientFactory _factory;                  #A
    public ValuesController(IHttpClientFactory factory)            #A
    {
        _factory = factory;
    }

    [HttpGet("values")]
    public async Task<string> GetRates()
    {
        HttpClient client = _factory.CreateClient("rates");          #B

        var response = await client.GetAsync("latest");             #C

        response.EnsureSuccessStatusCode();                          #C
        return await response.Content.ReadAsStringAsync();           #C
    }
}
```

#A Inject the IHttpClientFactory using DI
#B Request the named client called "rates" and configure it as defined in ConfigureServices()
#C Use the HttpClient in the same way as before

> **NOTE** You can still create “unconfigured” clients using `CreateClient()` without a name. Be aware that if you pass an unconfigured name, for example `CreateClient("MyRates")`, then the client returned will be unconfigured. Take care—client names are case sensitive, so `"rates"` is a different client to `"Rates"`!

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Named clients allow you to centralize your `HttpClient` configuration in one place, removing the responsibility of *configuring* the client from your consuming code. But you're still working with "raw" HTTP calls at this point, for example providing the relative URL to call (`"/latest"`) and parsing the response. `IHttpClientFactory` includes a feature that makes it easier to clean up this code.

21.2.3 Using Typed clients to encapsulate HTTP calls

In this section I teach about "typed clients." These take the "named client" approach one step further—as well as encapsulating the configuration for calling a third-party API, they also encapsulate the HTTP details, such as which URLs to call, what HTTP verbs to use, and what data is returned. Encapsulating these details in a single location makes the API easier to consumer in your code.

A common pattern when you need to interact with an API is to encapsulate the mechanics of that interaction into a separate service. You could easily do this with the `IHttpClientFactory` features you've already seen, by extracting the body of the `GetRates()` function from listing 21.5 into a separate service. But `IHttpClientFactory` has deeper support for this pattern too.

`IHttpClientFactory` supports *typed clients*. A typed client is a class that accepts a configured `HttpClient` in its constructor. It uses the `HttpClient` to interact with the remote API and exposes a clean interface for consumers to call. All of the logic for interacting with the remote API is encapsulated in the typed client, such as which URL paths to call, which HTTP verbs to use, and the types of response the API returns. This encapsulation makes it easier to call the third-party API from multiple places in your app by using the typed client.

For example, the following listing shows an example typed client for the exchange rates API shown in previous listings. It accepts an `HttpClient` in its constructor, and exposes a `GetLatestRates()` method that encapsulates the logic for interacting with the third-party API.

Listing 21.6 Creating a typed client for the exchange rates API

```
public class ExchangeRatesClient
{
    private readonly HttpClient _client;                     #A
    public ExchangeRatesClient(HttpClient client)            #A
    {
        _client = client;
    }

    public async Task<string> GetLatestRates()               #B
    {
        var response = await _client.GetAsync("latest");     #C
        response.EnsureSuccessStatusCode();                  #C

        return await response.Content.ReadAsString();        #C
    }
}
```

#A Inject an HttpClient using DI instead of an IHttpClientFactory

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

#B The GetLatestRates() logic encapsulates the logic for interacting with the API
#C Use the HttpClient the same way as before

We can then inject this `ExchangeRatesClient` into consuming services, and they don't need to know anything about how to make HTTP requests to the remote service, they just need to interact with the typed client. We can update listing 21.3 to use the typed client as shown in the following listing, at which point the `GetRates()` action method becomes trivial.

Listing 21.7 Consuming a typed client to encapsulate calls to a remote HTTP server

```
[ApiController]
public class ValuesController : ControllerBase
{
    private readonly ExchangeRatesClient _ratesClient;            #A
    public ValuesController(ExchangeRatesClient ratesClient)      #A
    {
        _ratesClient = ratesClient;
    }

    [HttpGet("values")]
    public async Task<string> GetRates()
    {
        return await _ratesClient.GetLatestRates();          #B
    }
}
```

#A Inject the typed client in the constructor
#B Call the typed client's API. The typed client handles making the correct HTTP requests

You may be a little confused at this point: I haven't mentioned how `IHttpClientFactory` is involved yet!

The `ExchangeRatesClient` takes an `HttpClient` in its constructor. `IHttpClientFactory` is responsible for creating the `HttpClient`, configuring it to call the remote service, and injecting it into a new instance of the typed client.

You can register the `ExchangeRatesClient` as a typed client and configure the `HttpClient` that is injected in `ConfigureServices`, as shown in the following listing. This is very similar to configuring a named client, so you can register additional configuration for the `HttpClient` that will be injected into the typed client

Listing 21.8 Registering a typed client with `HttpClientFactory` in Startup.cs

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHttpClient<ExchangeRatesClient>                   #A
        (HttpClient client) =>                                    #B
    {
        client.BaseAddress =                                      #B
            new Uri("https://api.exchangeratesapi.io");           #B
        client.DefaultRequestHeaders.Add(                         #B
            HeaderNames.UserAgent, "ExchangeRateViewer");         #B
    })
    .ConfigureHttpClient((HttpClient client) => {});              #C
```


Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
}
```

#A Register a typed client using the generic AddHttpClient method
#B You can provide an additional configuration function for the HttpClient that will be injected
#C As for named clients, you can provide multiple configuration methods

> **TIP** You can think of a typed client as a wrapper around a named client. I'm a big fan of this approach as it encapsulates all the logic for interacting with a remote service in one place. It also avoids the "magic strings" that you use with named clients, removing the possibility of typos.

Another option when registering typed clients is to register an interface, in addition to the implementation. This is often a good practice, as it makes it much easier to test consuming code. For example, if the typed client in listing 21.6 implemented the interface `IExchangeRatesClient`, you could register the interface and typed client implementation using

```
services.AddHttpClient<IExchangeRatesClient, ExchangeRatesClient>()
```

You could then inject this into consuming code using the interface type, for example:

```
public ValuesController(IExchangeRatesClient ratesClient)
```

Another commonly used pattern is to not provide any configuration for the typed client in `ConfigureServices()`. Instead, you could place that logic in the constructor of your `ExchangeRatesClient` using the injected `HttpClient`:

```
public class ExchangeRatesClient
{
    private readonly HttpClient _client;
    public ExchangeRatesClient(HttpClient client)
    {
        _client = client;
        _client.BaseAddress = new Uri("https://api.exchangeratesapi.io");
    }
}
```

This is functionally equivalent to the approach shown in listing 21.8, it's a matter of taste where you'd rather put the configuration for your `HttpClient`. If you take this approach, you don't need to provide a configuration lambda in `ConfigureServices`:

```
services.AddHttpClient<ExchangeRatesClient>();
```

Named clients and typed clients are convenient for managing and encapsulating `HttpClient` configuration, but `IHttpClientFactory` brings another advantage we haven't looked at yet: it's easier to extend the `HttpClient` handler pipeline.

21.3 Handling transient HTTP errors with Polly

In this section you'll learn how to handle a very common scenario: "transient" errors when you make calls to a remote service, caused by an error in the remote server, or temporary

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

network issues. You'll see how to use `IHttpClientFactory` to handle cross-cutting concerns like this by adding handlers to the `HttpClient` handler pipeline.

In section 21.2.1 I described `HttpClient` as consisting of a "pipeline" of handlers. The big advantage of this pipeline, much like the middleware pipeline of your application, is it allows you to add cross-cutting concerns to all requests. For example, `IHttpClientFactory` automatically adds a handler to each `HttpClient` that logs the status code and duration of each outgoing request.

As well as logging, another very common requirement is to handle transient errors when calling an external API. Transient errors can happen when the network drops out, or if a remote API goes offline temporarily. For transient errors, simply trying the request again can often succeed, but having to *manually* write the code to do so is cumbersome.

ASP.NET Core includes a library called Microsoft.Extensions.Http.Polly that makes handling transient errors easier. It uses the popular open source library Polly (www.thepollyproject.org) to automatically retry requests that fail due to transient network errors.

Polly is a mature library for handling transient errors that includes a variety of different error handling strategies, such as simple retries, exponential back off, circuit breaking, bulkhead isolation, and many more. Each strategy is explained in detail at https://github.com/App-vNext/Polly, so be sure to read the benefits and trade-offs when selecting a strategy.

To provide a taste of what's available, we'll add a simple retry policy to the `ExchangeRatesClient` shown in section 21.2. If a request fails due to a network problem such as a timeout or a server error, we'll configure Polly to automatically retry the request as part of the handler pipeline, as shown in figure 21.5.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

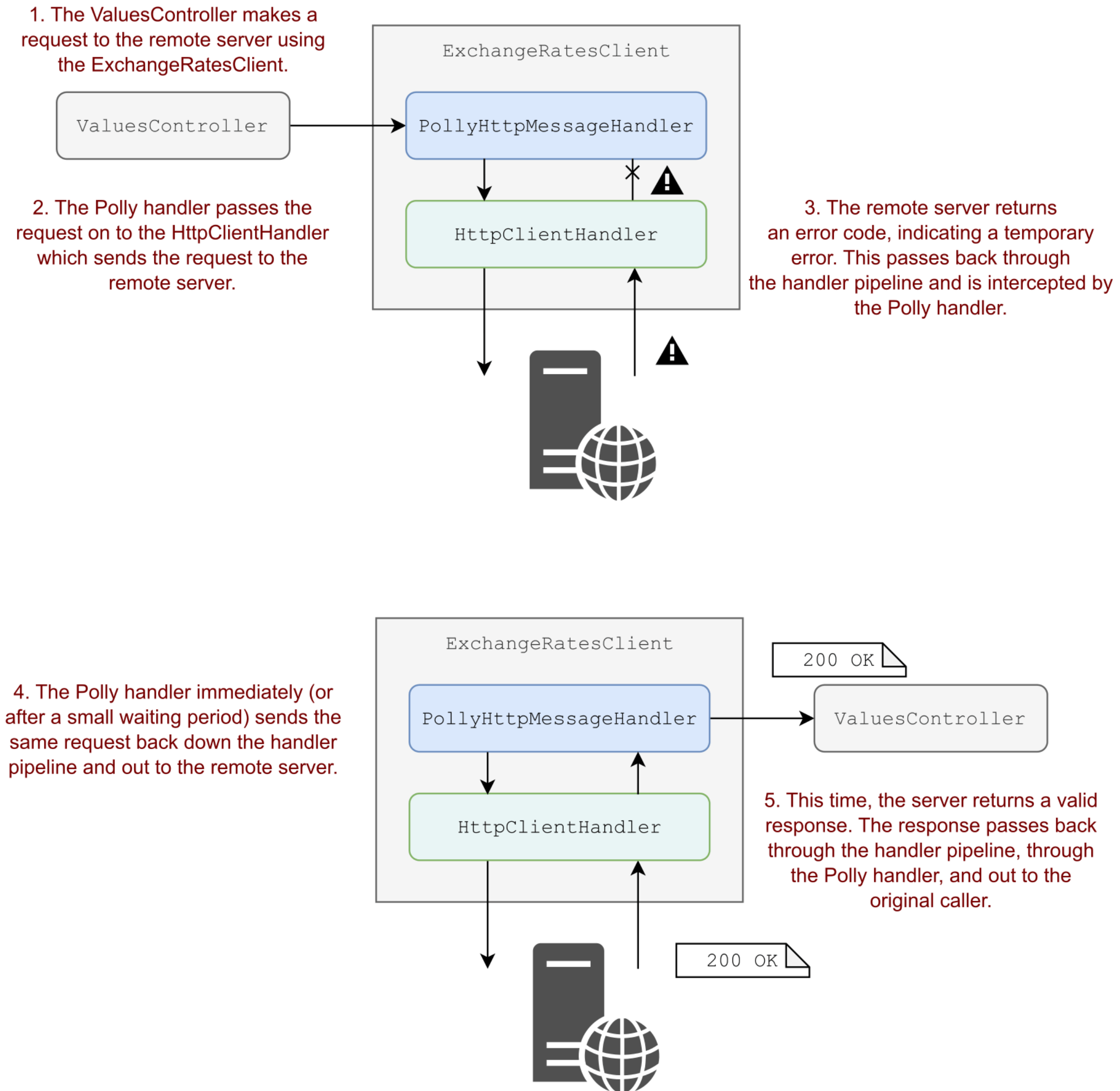


Figure 21.5 Using the `PolicyHttpMessageHandler` to handle transient errors. If an error occurs when calling the remote API, the Polly handler will automatically retry the request. If the request then succeeds, the result is passed back to the caller. The caller didn't have to handle the error themselves, making it simpler to use the `HttpClient` while remaining resilient to transient errors.

To add transient error handling to a named or `HttpClient`, you must:

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

1. Install the Microsoft.Extensions.Http.Polly NuGet package in your project by running `dotnet add package Microsoft.Extensions.Http.Polly`, by using the NuGet explorer in Visual Studio, or by adding a `<PackageReference>` element to your project file as shown below

```
<PackageReference Include="Microsoft.Extensions.Http.Polly"
    Version="3.1.6" />
```

2. Configure a named or typed client as shown in listings 21.5 and 21.7.
3. Configure a transient error handling policy for your client as shown in listing 21.8.

Listing 21.8 Configuring a transient error handling policy for a typed client in Startup.cs

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHttpClient<ExchangeRatesClient>()                 #A
        .AddTransientHttpErrorPolicy(policy =>                    #B
            policy.WaitAndRetryAsync(new[] {                      #C
                TimeSpan.FromMilliseconds(200),                   #D
                TimeSpan.FromMilliseconds(500),                   #D
                TimeSpan.FromSeconds(1)                           #D
            })
        );
}
```

#A You can add transient error handler to named or typed handlers
#B Use the extension methods provided by the NuGet package to add transient error handler
#C Configure the retry policy used by the handler. There are many types of policy to choose from
#D Configures a policy that waits and retries 3 times if an error occurs

In the listing above, we configure the error handler to catch transient errors and retry three times, waiting an increasing amount of time between requests. If the request fails on the third try, the handler will ignore the error, and pass it back to the client, just as if there was no error handler at all. By default, the handler will retry any request that either

- Throws an `HttpRequestException`, indicating an error at the protocol level, such as a closed connection, or
- Returns an HTTP 5xx status code, indicating a server error at the API, or
- Returns an HTTP 408 status code, indicating a timeout.

> **TIP** If you want to handle more cases automatically, or to restrict the responses that will be automatically retried, you can customize the selection logic as described in the documentation: https://github.com/App-vNext/Polly/wiki/Polly-and-HttpClientFactory.

Using standard handlers like the transient error handler allows you to apply the same logic across all requests made by a given `HttpClient`. The exact strategy you choose will depend on the characteristics of both the service and the request, but a good retry strategy is a must whenever you interact with potentially unreliable HTTP APIs.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

The Polly error handler is an example of an optional `HttpMessageHandler` that you can plug in to your `HttpClient`, but you can also create your own custom handler. In the next section you'll see how to create a handler that adds a header to all outgoing requests.

21.4 Creating a custom HttpMessageHandler

In this section you'll learn how to create an `HttpMessageHandler` that adds a custom HTTP header to all outgoing requests. You could use this handler to attach an API key to all outgoing requests to a given third-party API, for example. You'll see how to create the handler and how to register it with a typed client.

Most third-party APIs will require some form of authentication when you're calling them. For example, many services require you attach an API key to an outgoing request, so that the request can be tied to your account. Instead of having to remember to manually add this header for every request to the API, you could configure a custom `HttpMessageHandler` to automatically attach the header for you.

> **NOTE** More complex APIs may use Json Web Tokens (JWT) obtained from an identity provider. If that's the case, consider using the open source IdentityModel library (https://identitymodel.readthedocs.io) which provides integration points for ASP.NET Core Identity and `HttpClientFactory`.

You can configure a named or typed client using `IHttpClientFactory` to use your API-key handler as part of the `HttpClient`'s handler pipeline, as shown in figure 21.6. When you use the `HttpClient` to send a message, the `HttpRequestMesssage` is passed through each handler in turn. The API-key handler adds the extra header and passes the request to the next handler in the pipeline. Eventually, the `HttpClientHandler` makes the network request to send the HTTP request. After the response is received, each handler gets a chance to inspect (and potentially modify) the response.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

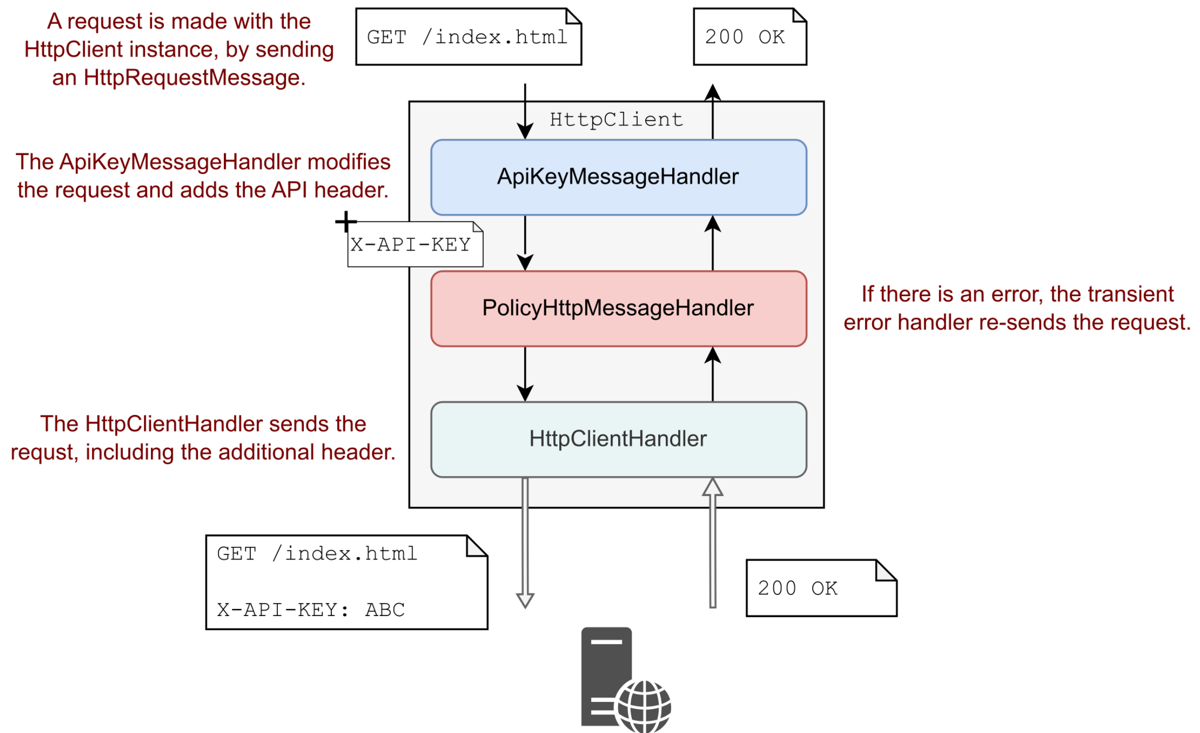


Figure 21.6 You can use a custom `HttpMessageHandler` to modify requests before they're sent to third-party APIs. Every request passes through the handler, before the final handler, the `HttpClientHandler` sends the request to the HTTP API. After the response is received, each handler gets a chance to inspect and modify the response.

To create a custom `HttpMessageHandler` and add it to a typed or named client's pipeline, you must:

1. Create a custom handler by deriving from the `DelegatingHandler` base class.
2. Override the `SendAsync()` method to provide your custom behavior. Call `base.SendAsync()` to execute the remainder of the handler pipeline.
3. Register your handler with the DI container. If your handler does not require state, you can register it as a singleton service, otherwise you should register it as a transient service.
4. Add the handler to one or more of your named or typed clients by calling `AddHttpMessageHandler<T>()`, on an `IHttpClientBuilder`, where `T` is your handler type. The order you register handlers dictates the order they will be added to the `HttpClient` handler pipeline. You can add the same handler type more than once in a pipeline if you wish, and to multiple typed or named clients.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

The following listing shows an example of a custom `HttpMessageHandler` that adds a header to every outgoing request. We use the custom `"X-API-KEY"` header in this example, but the header you need will vary depending on the third-party API you're calling. This example uses strongly typed configuration to inject the secret API key, as you saw in chapter 10.

Listing 21.9 Creating a custom `HttpMessageHandler`

```
public class ApiKeyMessageHandler : DelegatingHandler      #A
{
    private readonly ExchangeRateApiSettings _settings;    #B
    public ApiKeyMessageHandler(        #B
        IOptions<ExchangeRateApiSettings> settings)        #B
    {          #B
        _settings = settings.Value;      #B
    }          #B

    protected override async Task<HttpResponseMessage> SendAsync(    #C
        HttpRequestMessage request,             #C
        CancellationToken cancellationToken)     #C
    {
        request.Headers.Add("X-API-KEY", _settings.ApiKey);       #D

        HttpResponseMessage response =            #E
            await base.SendAsync(request, cancellationToken);      #E

        return response;           #F
    }
}
```

#A Custom HttpMessageHandlers should derive from DelegatingHandler
#B Inject the strongly typed configuration values using dependency injection
#C Override the SendAsync method to implement the custom behavior
#D Add the extra header to all outgoing requests
#E Call the remainder of the pipeline and receive the response
#F You could inspect or modify the response before returning it

To use the handler, you must register it with the DI container, and add it to a named or typed client. In the following listing, we add it to the `ExchangeRatesClient`, along with the transient error handler we registered in listing 21.8. This creates a pipeline similar to that shown in figure 21.6.

Listing 21.10 Registering a custom handler in Startup.ConfigureServices

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddTransient<ApiKeyMessageHandler>();     #A

    services.AddHttpClient<ExchangeRatesClient>()
        .AddHttpMessageHandler<ApiKeyMessageHandler>()    #B
        .AddTransientHttpErrorPolicy(policy =>      #C
            policy.WaitAndRetryAsync(new[] {
                TimeSpan.FromMilliseconds(200),
                TimeSpan.FromMilliseconds(500),
                TimeSpan.FromSeconds(1)
            })
```


Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
        );
}
```

#A Register the custom handler with the DI container
#B Configure the typed client to use the custom handler
#C Add the transient error handler. The order they are registered, dictates their order in the pipeline

Whenever you make a request using the typed client `ExchangeRatesClient`, you can be sure the API key will be added, and that transient errors will be handled automatically for you.

That brings us to the end of this chapter on `IHttpClientFactory`. Given the difficulties in using `HttpClient` correctly I showed in section 21.1, you should always favor `IHttpClientFactory` where possible. As a bonus, `IHttpClientFactory` allows you to easily centralize your API configuration using named clients and to encapsulate your API interactions using typed clients.

21.5 Summary

- Use the `HttpClient` class for calling HTTP APIs. You can use it to make HTTP calls to APIs, providing all the headers and body to send in a request, and reading the response headers and data you get back.
- `HttpClient` uses a pipeline of handlers, consisting of multiple `HttpMessageHandlers`, connected in a similar way to the middleware pipeline used in ASP.NET Core. The final handler is the `HttpClientHandler` which is responsible for making the network connection and sending the request.
- `HttpClient` implements `IDisposable`, but you shouldn't typically dispose it. When the `HttpClientHandler` which makes the TCP/IP connection is disposed, it keeps a connection open for the TIME_WAIT period. Disposing many `HttpClient`s in a short period of time can lead to socket exhaustion, preventing a machine from handling any more requests.
- Prior to .NET Core 2.1, the advice was to use a single `HttpClient` for the lifetime of your application. Unfortunately, a singleton `HttpClient` will not respect DNS changes, which are commonly used for traffic management in cloud environments.
- `IHttpClientFactory` solves both these problems by managing the lifetime of the `HttpMessageHandler` pipeline. You can create a new `HttpClient` by calling `CreateClient()`, and `IHttpClientFactory` takes care of disposing the handler pipeline when it is no longer in use.
- You can centralize the configuration of an `HttpClient` in `ConfigureServices()` using *named* clients by calling `AddHttpClient("test", c => {})`. You can then retrieve a configured instance of the client in your services by calling `IHttpClientFactory.CreateClient("test")`.
- You can create a *typed* client by injecting an `HttpClient` into a service, `T`, and configuring the client using `AddHttpClient<T>(c => {})`. Typed clients are great for abstracting the HTTP mechanics away from consumers of your client.
- You can use the Microsoft.Extensions.Http.Polly library to add transient HTTP error

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

handling to your `HttpClient`s. Call `AddTransientHttpErrorPolicy()` when configuring your `IHttpClientFactory` in `ConfigureServices`, and provide a Polly policy to control when errors should be automatically handled and retried.

- It's common to use a simple "retry" policy to try making a request multiple times before giving up and returning an error. When designing a policy be sure to consider the impact of your policy; in some circumstances it may be better to fail quickly instead of retrying a request which is never going to succeed. Polly includes additional policies such as circuit-breakers to create more advanced approaches.
- By default, the transient error handling middleware will handle connection errors, server errors that return a 5xx error code, and 408 (timeout) errors. You can customize this if you want to handle additional error types but ensure that you only retry requests which are safe to do so.
- You can create a custom `HttpMessageHandler` to modify each request made through a named or typed client. Custom handlers are good for implementing cross-cutting concerns such as logging, metrics, and authentication.
- To create a custom `HttpMessageHandler`, derive from `DelegatingHandler` and override the `SendAsync()` method. Call `base.SendAsync()` to send the request to the next handler in the pipeline and finally to the `HttpClientHandler` which makes the HTTP request.
- Register your custom handler in the DI container as either a transient or a singleton. Add it to a named or typed client using `AddHttpMessageHandler<T>()`. The order you register the handler in the `IHttpClientBuilder` is the order the handler will appear in the `HttpClient` handler pipeline.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

22 Building background tasks and services

This chapter covers

- **Creating tasks that run in the background for your application**
- **Using the generic `IHost` to create Windows Services and Linux daemons**
- **Using Quartz.NET to run tasks on a schedule, in a clustered environment**

We've covered a lot of ground in the book so far. You've learned how to create page-based applications using Razor Pages and how to create APIs for mobile clients and services. You've seen how to add authentication and authorization to your application, how to use EF Core for storing state in the database, and how to create custom components to meet your requirements.

As well as these "UI" focused apps, you may find you need to build "background" or "batch-task" services. These services aren't meant to interact with users directly. Rather they stay running in the background, processing items from a queue or periodically executing a long-running process.

For example, you might want to have a background service that sends email confirmations for eCommerce orders, or a batch job that calculates sales and losses for retail stores after the shops close. ASP.NET Core includes support for these "background tasks" by providing abstractions for running a task in the background when your application starts.

In section 22.1 you'll learn about the background task support provided in ASP.NET Core by the `IHostedService` interface. You'll learn how to use the `BackgroundService` helper class to create tasks that run on a timer, and how to manage your DI lifetimes correctly in a long-running task.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

In section 22.2 we take the background service concept one step further to create "headless" worker services, using the generic `IHost`. Worker services don't use Razor Pages or API controllers; instead they consist only of `IHostedService`s running tasks in the background. You'll also see how to configure and install a worker service app as a Windows Service, or as a Linux daemon.

In section 22.3 I introduce the open source library Quartz.NET, which provides extensive scheduling capabilities for creating background services. You'll learn how to install Quartz.NET in your applications, how to create complex schedules for your tasks, and how to add redundancy to your worker services by using clustering.

Before we get to more complex scenarios, we'll start by looking at the built-in support for running background tasks in your apps.

22.1 Running background tasks with IHostedService

In this section you'll learn how to create background tasks that run for the lifetime of the application using `IHostedService`. You'll create a task that caches the values from a remote service every 5 minutes, so that the rest of the application can retrieve the values from the cache. You'll then learn how to use services with a scoped lifetime in your singleton background services by managing container scopes yourself.

In most applications, it's common to want to create tasks that happen in the background, rather than in response to a request. This could be a task to process a queue of emails; handling events published to some sort of a message bus; or running a batch process to calculate daily profits. By moving this work to a background task, your user interface can stay responsive. Instead of trying to send an email immediately for example, you could add the request to a queue and return a response to the user immediately. The background task can consume that queue in the background at its leisure.

In ASP.NET Core, you can use the `IHostedService` interface to run tasks in the background. Classes which implement this interface are started when your application starts, shortly after your application starts handling requests, and are stopped shortly before your application is stopped. This provides the hooks you need to perform most tasks

> **NOTE** Even the ASP.NET Core server, Kestrel, runs as an `IHostedService`. In one sense, almost everything in an ASP.NET Core app is a "background" task!

In this section you'll see how to use the `IHostedService` to create a background task that runs continuously throughout the lifetime of your app. This could be used for many different things, but in the next section you'll see how to use it to populate a simple cache.

22.1.1 Running background tasks on a timer

In this section you'll learn how to create a background task that runs periodically on a timer, throughout the lifetime of your app. Running background tasks can be useful for many reasons, such as scheduling work to be performed later or for performing work in-advance.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

For example, in chapter 21, we used `IHttpClientFactory` and a typed client to call a third-party service to retrieve the current exchange rate between various currencies, and returned them in an API controller, as shown in the following listing. A simple optimization for this code might be to cache the exchange rate values for a period.

Listing 22.1 Using a typed client to return exchange rates from a third-party service

```
[ApiController]
public class ValuesController : ControllerBase
{
    private readonly ExchangeRatesClient _typedClient;          #A
    public ValuesController(ExchangeRatesClient typedClient)    #A
    {
        _typedClient = typedClient;
    }

    [HttpGet("values")]
    public async Task<string> GetRates()
    {
        return await _typedClient.GetLatestRatesAsync()    #B
    }
}
```

#A A typed client created using IHttpClientFactory is injected in the constructor
#B The typed client is used to retrieve exchange rates from the remote API and returns them

There are multiple ways you could implement that, but in this section, we'll use a simple cache that pre-emptively fetches the exchange rates in the background, as shown in figure 22.1. The API controller simply reads from the cache; it never has to make HTTP calls itself, so it remains fast.

> **NOTE** **An alternative approach might add caching to your strongly typed client,** `ExchangeRateClient`**. The downside is that when you need to update the rates, you will have to do the request immediately, making the overall response slower. Using a background service keeps your API controller consistently fast.**

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

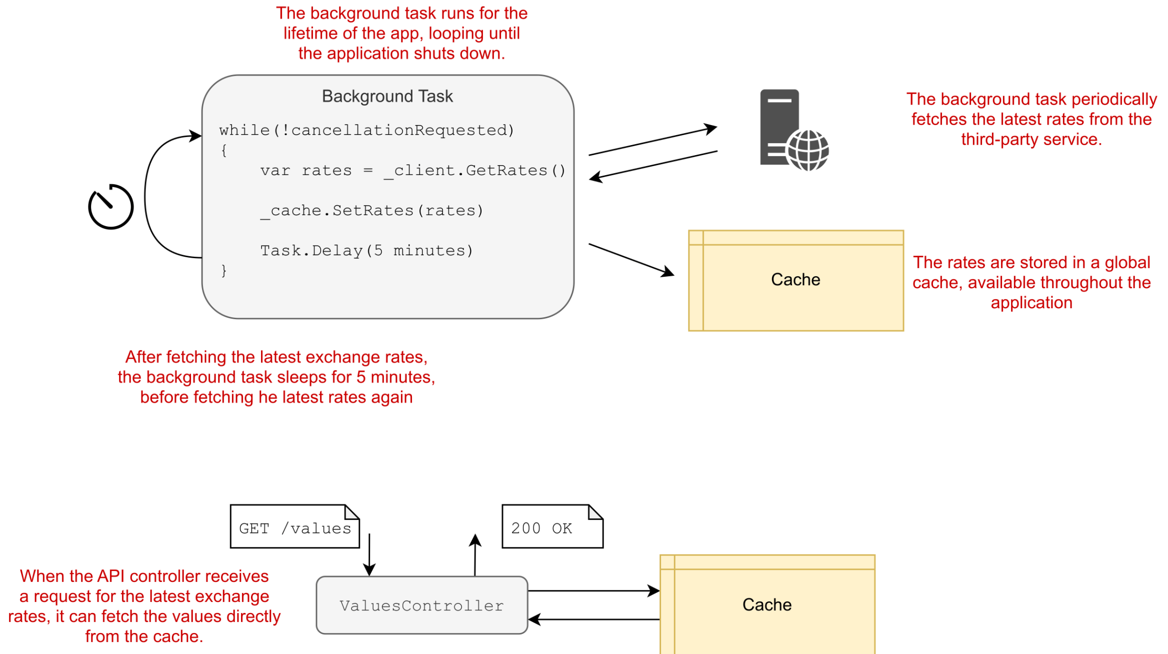


Figure 22.1 You can use a background task to cache the results from a third-party API on a schedule. The API controller can then read directly from the cache, instead of calling the third-party API itself. This reduces the latency of requests to your API controller, while ensuring the data remains fresh.

You can implement a background task using the interface `IHostedService`. This consists of two methods:

```
public interface IHostedService
{
    Task StartAsync(CancellationToken cancellationToken);
    Task StopAsync(CancellationToken cancellationToken);
}
```

but there are subtleties to implementing the interface correctly. In particular, the `StartAsync()` method, although asynchronous, runs “inline” as part of your application startup. Background tasks that are expected to run for the lifetime of your application must return a `Task` immediately, and schedule background work on a different thread.

> **WARNING** Calling await in `IHostedService.StartAsync()` method will block your application starting until the method completes. This can be useful in some cases but is often not the desired behavior for background tasks.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

To make it easier to create background services using best practice patterns, ASP.NET Core provides the abstract base class `BackgroundService` which implements `IHostedService` and is designed to be used for long running tasks. To create a background task you must override a single method of this class, `ExecuteAsync()`. You're free to use `async-await` inside this method and you can keep running the method for the lifetime of your app.

For example, the following listing shows a background service that fetches the latest interest rates using a typed client and saves them in a cache, as you saw in figure 22.1. The `ExecuteAsync()` method keeps looping and updating the cache until the `CancellationToken` passed as an argument indicates that the application is shutting down.

Listing 22.2 Implementing a `BackgroundService` that calls a remote HTTP API

```
public class ExchangeRatesHostedService : BackgroundService              #A
{
    private readonly IServiceProvider _provider;                         #B
    private readonly ExchangeRatesCache _cache;                          #C
    public ExchangeRatesHostedService(
        IServiceProvider provider, ExchangeRatesCache cache)
    {
        _provider = provider;
        _cache = cache;
    }

    protected override async Task ExecuteAsync(                          #D
        CancellationToken stoppingToken)                                 #E
    {
        while (!stoppingToken.IsCancellationRequested)                   #F
        {
            var client = _provider                                       #G
                .GetRequiredService<ExchangeRatesClient>();              #G

            string rates= await client.GetLatestRatesAsync();            #H
            _cache.SetRates(latest);                                     #I

            await Task.Delay(TimeSpan.FromMinutes(5), stoppingToken);    #J
        }
    }
}
```

#A Derive from BackgroundService to create a task that runs for the lifetime of your app
#B Inject an IServiceProvider so you can create instances of the typed client
#C A simple cache for exchange rates
#D You must override ExecuteAsync to set the service's behavior
#E The CancellationToken passed as an argument is triggered when the application shuts down
#F Keep looping until the application shuts down
#G Create a new instance of the typed client, so that the HttpClient is short-lived
#H Fetch the latest rates from the remote API
#I Store the rates in the cache
#J Wait for 5 minutes (or for the application to shut down) before updating the cache,

The `ExchangeRateCache` in listing 22.2 is a simple singleton that stores the latest rates. It must be thread-safe, as it will be accessed concurrently by your API controllers. You can see a simple implementation in the source code for this chapter.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

To register your background service with the DI container, use the `AddHostedService()` extension method in the `ConfigureServices()` method of Startup.cs, as shown in the following listing.

Listing 22.3 Registering an `IHostedService` with the DI container

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHttpClient<ExchangeRatesClient>()                #A
    services.AddSingleton<ExchangeRatesCache>();                 #B
    services.AddHostedService<ExchangeRatesHostedService>();     #C
}
```

#A Register the typed client as before
#B Add the cache object as a singleton, as you must share the same instance throughout your app
#C Register the ExchangeRatesHostedService as an IHostedService

By using a background service to fetch the exchange rates, your API controller becomes very simple. Instead of fetching the latest rates itself, it returns the value from the cache, which is kept up to date by the background service:

```
[ApiController]
public class ValuesController : ControllerBase
{
    private readonly ExchangeRatesCache _cache;
    public ValuesController(ExchangeRatesCache cache)
    {
        _cache = cache;
    }

    [HttpGet("values"]
    public string GetValues()
    {
        return _cache.GetLatestRates();
    }
}
```

One slightly messy aspect of listing 22.2 is that I've used the service-locator pattern to retrieve the typed client. This isn't ideal, but you shouldn't inject typed clients into background services directly. Typed clients are designed to be short lived, to ensure you take advantage of the `HttpClient` handler rotation as described in chapter 21. In contrast, background services are singletons that live for the lifetime of your application.

> **TIP** If you wish, you can avoid the service-locator pattern used in listing 22.2 by using the factory pattern described in this post: [https://www.stevejgordon.co.uk/ihttpclientfactory-patterns-using-typed-clients-from-singleton-services.](https://www.stevejgordon.co.uk/ihttpclientfactory-patterns-using-typed-clients-from-singleton-services)

The need to have short-lived services leads to another common question—how can you use "scoped" services in a background service?

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

22.1.2 Using scoped services in background tasks

Background services that implement `IHostedService` are created once when your application starts. That means they are, by necessity, singletons, as there will only ever be a single instance of the class.

That leads to a problem if you need to use services registered with a *scoped* lifetime. Any services you inject into the constructor of your singleton `IHostedService` must themselves be registered as singletons. Does that mean there's no way to use scoped dependencies in a background service?

> **REMINDER** As I discussed in chapter 10, a service should only use dependencies with a lifetime longer than or equal to the lifetime of the service, to avoid captured dependencies.

For example, lets imagine a slight variation of the caching example from section 22.1.1. Instead of storing the exchange rates in a singleton cache object, you want to save the exchange rates to a database, so you can look up the historic rates.

Most database providers, including EF Core's `DbContext`, register their services with scoped lifetimes. That means you need to access the *scoped* `DbContext`, from inside the *singleton* `ExchangeRatesHostedService`, which precludes injecting the `DbContext` with constructor injection. The solution is to create a new container scope every time you update the exchange rates.

In typical ASP.NET Core applications, the framework creates a new container scope every time a new request is received, just before the middleware pipeline executes. All the services that are used in that request are fetched from the scoped container. In a background service, there *are* no requests, so no container scopes are created. The solution is to create your own!

You can create a new container scope anywhere you have access to an `IServiceProvider` by calling `IServiceProvider.CreateScope()`. This creates a scoped container, which you can use to retrieve scoped services.

> **WARNING** Always make sure to dispose the `IServiceScope` returned by `CreateScope()` when you're finished with it, typically with a `using` statement. This disposes any services that were created by the scoped container, and prevents memory leaks.

The following listing shows a version of the `ExchangeRatesHostedService` that stores the latest exchange rates as an EF Core entity in the database. It creates a new scope for each iteration of the `while` loop and retrieves the scoped `AppDbContext` from the scoped container.

Listing 22.4 Consuming scoped services from an IHostedService

```
public class ExchangeRatesHostedService : BackgroundService             #A
{
    private readonly IServiceProvider _provider;                        #B
    public ExchangeRatesHostedService(IServiceProvider provider)        #B
    {
        _provider = provider;
```

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
    }

    protected override async Task ExecuteAsync(
        CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            using(IServiceScope scope = _provider.CreateScope())         #C
            {
                var scopedProvider = scope.ServiceProvider;              #D

                   var client = scope.ServiceProvider                       #E
                    .GetRequiredService<ExchangeRatesClient>();          #E

                var context = scope.ServiceProvider                      #E
                    .GetRequiredService<AppDbContext>();                 #E

                var rates= await client.GetLatestRatesAsync();           #F

                context.Add(rates);                                      #F
                await context.SaveChanges(rates);                        #F
             }                                                           #G

            await Task.Delay(TimeSpan.FromMinutes(5), stoppingToken);    #H
        }
    }
}
```

#A BackgroundService is registered as a singleton
#B The injected IServiceProvider can be used to retrieve singleton services, or to create scopes
#C Create a new scope using the root IServiceProvider
#D The scope exposes an IServiceProvider that can be used to retrieve scoped components
#E Retrieve the scoped services from the container
#F Fetch the latest rates, and save using EF Core
#G Dispose the scope with the using statement.
#H Wait for the next iteration. A new scope is created on the next iteration.

Creating scopes like this is a general solution whenever you find you need to access scoped services and you're not running in the context of a request. A prime example is when you're implementing `IConfigureOptions`, as you saw in chapter 19. You can take the exact same approach—creating a new scope—as shown here http://mng.bz/6m17.

> **TIP** **Using service location in this way always feels a bit convoluted. I typically try and extract the body of the task to a separate class and use service location to retrieve that class only. You can see an example of this approach in the documentation: https://docs.microsoft.com/aspnet/core/fundamentals/host/hosted-services#consuming-a-scoped-service-in-a-background-task.**

`IHostedService` is available in ASP.NET Core, so you can run background tasks in your Razor Pages or API controller applications. However, sometimes all you *want* is the background task and you don't need any UI. For those cases, you can use the raw `IHost` abstraction, without having to bother with HTTP handling at all.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

22.2 Creating headless worker services using IHost

In this section you'll learn about worker services, which are ASP.NET Core applications that do not handle HTTP traffic. You'll learn how to create a new worker service from a template and compare the generated code to a traditional ASP.NET Core application. You'll also learn how to install the worker service as a Windows Service or as a systemd daemon on Linux.

In section 22.1 we cached exchange rates based on the assumption that they're being consumed directly by the UI part of your application, by Razor Pages or API controllers for example. However, in the section 22.1.2 example we saved the rates to the database instead of storing them in-process. That raises the possibility of *other* applications with access to the database using the rates too. Taking that one step further, could we create an application which is *only* responsible for caching these rates, and has no UI at all?

Since .NET Core 3.0, ASP.NET Core has been built on top of a "generic" (as opposed to "web") `IHost` implementation. It is the `IHost` implementation that provides features such as configuration, logging, and dependency injection. ASP.NET Core adds the middleware pipeline for handling HTTP requests, as well as paradigms such as Razor Pages or MVC on top of that, as shown in figure 22.2.

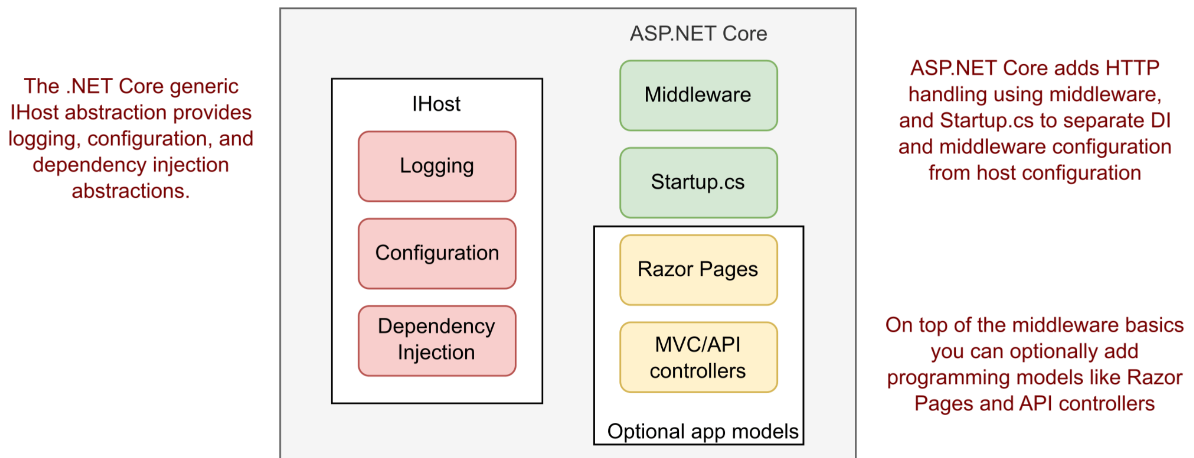


Figure 22.2 ASP.NET Core builds on the generic `IHost` implementation. The `IHost` provides features such as configuration, dependency injection, and configuration. ASP.NET Core adds HTTP handling on top of that by way of the middleware pipeline, Razor Pages and API controllers. If you don't need HTTP handling, you can use the `IHost` without the additional ASP.NET Core libraries to create a smaller application.

If your application doesn't *need* to handle HTTP requests, then there's no real reason to use ASP.NET Core. You can use the `IHost` implementation alone to create an application that will have a lower memory footprint, faster startup, and less surface area to worry about from a

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

security side than a full ASP.NET Core application. .NET Core applications that use this approach are commonly called *worker services* or *workers*.

> **DEFINITION** A *worker* is a .NET Core application that uses the generic `IHost` but doesn't include the ASP.NET Core libraries for handling HTTP requests. They are sometimes called "headless" services, as they don't expose a UI for you to interact with.

Workers are commonly used for running background tasks (`IHostedService` implementations) which don't require a UI. These tasks could be for running batch jobs, for running tasks repeatedly on a schedule, or for handling events using some sort of message bus. In the next section we'll create a worker for retrieving the latest exchange rates from a remote API, instead of adding the background task to an ASP.NET Core application.

22.2.1 Creating a worker service from a template

In this section you'll see how to create a basic worker service from a template. Visual Studio includes a template for creating worker services by choosing File > New > Project > Worker Service. You can create a similar template using the .NET CLI by running `dotnet new worker`. The resulting template consists of two C# files:

- Worker.cs—This is a simple `BackgroundService` implementation that writes to the log every second. You can replace this class with your own `BackgroundService` implementation, such as the example from listing 22.4.
- Program.cs—As with a typical ASP.NET Core application, this contains the entry point for your application, and is where the `IHost` is built and run. In contrast to a typical ASP.NET Core app, it's *also* where you will configure the dependency injection container for your application.

The most notable difference between the worker service template and an ASP.NET Core template is that there is no Startup.cs file. In ASP.NET Core applications, Startup.cs is where you usually configure your DI container and your middleware pipeline. The worker service doesn't have a middleware pipeline (as it doesn't handle HTTP requests), but it *does* use DI, so where is that configured?

In worker service templates, you configure DI in Program.cs using the `ConfigureServices()` method as shown in the following listing. This method is functionally identical to the `ConfigureServices()` method in Startup.cs, so you can use exactly the same syntax. The following listing shows how to configure EF Core, the exchange rates typed client from chapter 21, and the background service that saves exchange rates to the database, as you saw in section 22.1.2.

Listing 22.5 Program.cs for a worker service that saves exchange rates using EF Core

```
public class Program
{
    public static void Main(string[] args)
```


Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
    {
        CreateHostBuilder(args).Build().Run();                       #A
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>      #B
        Host.CreateDefaultBuilder(args)                              #B
            .ConfigureServices((hostContext, services) =>
            {
                services.AddHttpClient<ExchangeRatesClient>();          #C
                services                                             #C
                  .AddHostedService<ExchangeRatesHostedService>();      #C

                services.AddDbContext<AppDbContext>(options =>          #C
                  options.UseSqlite(                                 #C
                      hostContext.Configuration                      #D
                          .GetConnectionString("SqlLiteConnection"))    #D
                );
            });
}
```

#A A worker creates an IHostBuilder, builds an IHost, and runs it, the same as an ASP.NET Core app
#B The same HostBuilder code is used, but there is no call to ConfigureWebHostDefaults
#C Add services in ConfigureServices, the same as you typically would in Startup.cs
#D IConfiguration can be accessed from the HostBuilderContext parameter

> **TIP** **You can use the** `IHostBuilder.ConfigureServices()` **methods in ASP.NET Core apps too, but the general convention is to use Startup.cs instead. The** `IHostBuilder` **methods are useful in some circumstances when you need to control exactly when your background tasks start, as I describe in this post** https://andrewlock.net/controlling-ihostedservice-execution-order-in-aspnetcore-3/.

The changes in Program.cs, and the lack of a Startup.cs file, are the most obvious differences between a worker service and an ASP.NET Core app, but there are some important differences in the .csproj project file too. The following listing shows the project file for a worker service that uses `IHttpClientFactory` and EF Core, and highlights some of the differences compared to a similar ASP.NET Core application.

Listing 22.6 Project file for a worker service

```
<Project Sdk="Microsoft.NET.Sdk.Worker">                              #A

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>                  #B
    <UserSecretsId>5088-4277-B226-DC0A790AB790</UserSecretsId>        #C
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Hosting"            #D
        Version="3.1.6" />                                            #D
    <PackageReference Include="Microsoft.Extensions.Http"               #E
        Version="3.1.6" />                                            #E
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design"    #F
         Version="3.1.6" PrivateAssets="All" />                       #F
    <PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite"    #F
         Version="3.1.6" />                                           #F
```


Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
  </ItemGroup>
</Project>
```

#A Worker services use a different project SDK type to ASP.NET Core apps
#B The target framework is the same as for ASP.NET Core apps
#C Worker services use configuration, so can use UserSecrets, the same as ASP.NET Core apps
#D All worker services must explicitly add this package. ASP.NET Core apps add it implicitly
#E If you're using IHttpClientFactory, you'll need to add this package in worker services
#F EF Core packages must be explicitly added, the same as for ASP.NET Core apps

Some parts of the project file are the same for both worker services and ASP.NET Core apps:

- Both types of app must specify a `<TargetFramework>`, such as `netcoreapp3.1` for .NET Core 3.1, or `net5.0` for .NET 5.
- Both types of apps use the configuration system, so you can use `<UserSecretsId>` to manage secrets in development, as discussed in chapter 11.
- Both types of app must explicitly add references to the EF Core NuGet packages to use EF Core in the app.

There are also several differences in the project template:

- The `<Project>` element's `Sdk` for a worker service should be `Microsoft.NET.Sdk.Worker`, while for an ASP.NET Core app it is `Microsoft.NET.Sdk.Web`. The Web SDK includes implicit references to additional packages that are not generally required in worker services.
- The worker service *must* include an explicit `PackageReference` for the Microsoft.Extensions.Hosting NuGet package. This package includes the generic `IHost` implementation used by worker services.
- You may need to include additional packages to reference the same functionality, when compared to an ASP.NET Core app. An example of this is the Microsoft.Extensions.Http package (which provides `IHttpClientFactory`). This package is referenced implicitly in ASP.NET Core apps but must be explicitly referenced in worker services.

Running a worker service is the same as running an ASP.NET Core application: use `dotnet run` from the command line or hit F5 from Visual Studio. A worker service is essentially just a console application (as are ASP.NET Core applications), so they both run the same way.

You can run worker services in most of the same places you would run an ASP.NET Core application, though as a worker service doesn't handle HTTP traffic, some options make more sense than others. In the next section, we'll look at two supported ways of running your application: as a Windows Service or as a Linux systemd daemon.

22.2.2 Running worker services in production

In this section you'll learn how to run worker services in production. You'll learn how to install a worker service as a Windows service so that the operating system monitors and starts your worker service automatically. You'll also see how to prepare your application for installation as a systemd daemon on Linux.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Worker services, like ASP.NET Core applications, are fundamentally just .NET Core console applications. The difference is that they are typically intended to be long-running applications. The common approach for running these types of applications on Windows is to use a Windows Service or to use a systemd daemon on Linux.

> **NOTE** It's also very common to run applications in the Cloud using Docker containers or dedicated platform services like Azure App Service. The process for deploying a worker service to these managed services is typically identical to deploying an ASP.NET Core application.

Adding support for Windows services or systemd is easy, thanks to two optional NuGet packages:

- *Microsoft.Extensions.Hosting.Systemd*. Adds support for running the application as a systemd application. To enable systemd integration, call `UseSystemd()` on your `IHostBuilder` in Program.cs.
- *Microsoft.Extensions.Hosting.WindowsServices*. Adds support for running the application as a Windows Service. To enable the integration, call `UseWindowsService()` on your `IHostBuilder` in Program.cs.
- These packages each add a single extension method to `IHostBuilder` that enables the appropriate integration when running as a systemd daemon, or as a Windows Service. For example, the following listing shows how to enable Windows Service support.

Listing 22.7 Adding Windows Service support to a worker service

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>     #A
        Host.CreateDefaultBuilder(args)                                #A
            .ConfigureServices((hostContext, services) =>              #A
            {                                                          #A
                services.AddHostedService<Worker>();                   #A
            })                                                         #A
            .UseWindowsService();                                      #B
}
```

#A Configure your worker service as you would normally
#B Add support for running as a Windows Service

During development, or if you run your application as a console app, the `AddWindowsService()` does nothing; your application runs exactly the same as it would without the method call. However, your application can now be installed as a Windows Service, as your app now has the required integration hooks to work with the Windows Service system.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

The following describes the basic steps to install a worker service app as a Windows Service:

1. Add the Microsoft.Extensions.Hosting.WindowsServices NuGet package to your application using Visual Studio, by running `dotnet add package Microsoft.Extensions.Hosting.WindowsServices` in the project folder, or by adding a `<PackageReference>` to your .csproj file:

```
<PackageReference Include="Microsoft.Extensions.Hosting.WindowsServices"
Version="3.1.6" />
```

2. Add a call to `UseWindowsService()` on your `IHostBuilder`, as shown in listing 22.7.
3. Publish your application, as described in chapter 16. From the command line, you could run `dotnet publish -c Release` from the project folder.
4. Open a command prompt as Administrator, and install the application using the Windows `sc` utility. You need to provide the path to your published project's .exe file and a name to use for the service, for example `My Test Service`:

```
sc create "My Test Service" BinPath="C:\path\to\MyService.exe"
```

5. You can manage the service from the Services control panel in Windows, as shown in figure 22.3. Alternatively, to start the service from the command line run `sc start "My Test Service"`, or to delete the service run `sc delete "My Test Service"`.

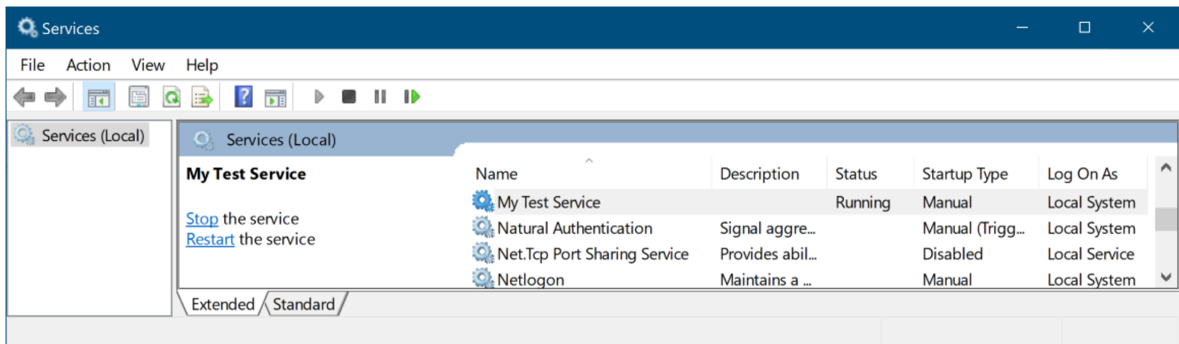


Figure 22.3 The Services control panel in Windows. After installing a worker service as a Windows Service using the `sc` utility, you can manage your worker service from here. This allows you to control when the Windows service starts and stops, the user account the application runs under, and how to handle errors.

> **WARNING** These steps are the bare minimum to install a Windows Service. When running in production you must consider many security aspects not covered here. For more details, see https://docs.microsoft.com/aspnet/core/host-and-deploy/windows-service.

After following the process above, your worker service will be running as a Windows service.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

An interesting point of note is that installing as a Windows Service or systemd daemon isn't limited to worker services only—you can install an ASP.NET Core application in the same way. Simply follow the instructions above, add the call to `UseWindowsService()`, and install your ASP.NET Core app. This is thanks to the fact that the ASP.NET Core functionality is built directly on top of the generic `Host` functionality.

You can follow a similar process to install a worker service as a systemd daemon by installing the Microsoft.Extensions.Hosting.Systemd package, and calling `UseSystemd()` on your `IHostBuilder`. For more details on configuring systemd, see https://docs.microsoft.com/aspnet/core/host-and-deploy/linux-nginx#monitor-the-app.

So far in this chapter we've used `IHostedService` and the `BackgroundService` to run tasks that repeat on an interval, and you've seen how to install worker services as long running applications, by installing as a Windows Service.

In the final section of this chapter, we look at how you can create more advanced schedules for your background tasks, as well as how to add resiliency to your application by running multiple instances of your workers. To achieve that, we'll use a mature third-party library, Quartz.NET.

22.3 Coordinating background tasks using Quartz.NET

In this section you'll learn how to use the open source scheduler library Quartz.NET. You'll learn how to install and configure the library, and how to add a background job to run on a schedule. You'll also learn how to enable clustering for your applications, so that you can run multiple instances of your worker service, and share jobs between them.

All the background tasks you've seen so far in this chapter repeat a task on an interval indefinitely, from the moment the application starts. However, sometimes you want more control of this timing. Maybe you always want to run the application at 15 minutes past each hour. Or maybe you only want to run a task on the second Tuesday of the month at 3am. Additionally, maybe you want to run multiple instances of your application for redundancy, but ensure that only one of the services runs a task at any one time.

It would certainly be possible to build all this extra functionality into an application yourself, but there are excellent libraries available which already provide all this functionality for you. Two of the most well known in the .NET space are Hangfire (www.hangfire.io) and Quartz.NET(www.quartz-scheduler.net).

Hangfire is an open source library which also has a "Pro" subscription option. One of its most popular features is a dashboard user interface, that shows the state of all your running jobs, each task's history, and any errors that occurred.

Quartz.NET is completely open source and essentially offers a "beefed-up" version of the `BackgroundService` functionality. It has extensive scheduling functionality, as well as support for running in a "clustered" environment, where multiple instances of your application coordinate to distribute the jobs amongst themselves.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

NOTE Quartz.NET is based on a similar Java library called Quartz Scheduler. When looking for information on Quartz.NET be sure you're looking at the correct Quartz!

Quartz.NET is based around four main concepts:

- *Jobs*. These are the background tasks that implement your logic.
- *Triggers*. These control *when* a job will run based on a schedule, such as "every 5 minutes", or "every second Tuesday". A job can have multiple triggers.
- *Job Factory*. The job factory is responsible for creating instances of your jobs. Quartz.NET integrates with ASP.NET Core's DI container, so you can use DI in your job classes.
- *Scheduler*. The Quartz.NET scheduler keeps track of the triggers in your application, creates job using the Job Factory, and runs your jobs. The scheduler typically runs as an `IHostedService` for the lifetime of your app.

Background services versus Cron Jobs

It's common to use cron jobs to run tasks on a schedule on linux and Windows has similar functionality called Scheduled Tasks. These are used to periodically run an application or script file, which is typically a short-lived task.

In contrast, .NET Core apps using background services are designed to be long-lived, even if they are only used to run tasks on a schedule. This allows your application to do things like adjust its schedule as required or perform optimizations. In addition, being long-lived means your app doesn't have to just run tasks on a schedule. It can respond to ad-hoc events, such as events in a message queue for example.

Of course, if you don't need those capabilities, and would rather not have a long-running application, you can use .NET Core in combination with cron jobs. You could create a simple .NET console app which runs your task and then shuts down, and could schedule it to execute periodically as a cron job. The choice is yours!

In this section I'll show how to install Quartz.NET and configure a background service to run on a schedule. I'll then show how to enable clustering, so that you can run multiple instances of your application and distribute the jobs between them.

22.3.1 Installing Quartz.NET in an ASP.NET Core application

In this section I show how to install the Quartz.NET scheduler into an ASP.NET Core application. Quartz.NET can be installed in any .NET Core application, so you can add it to an ASP.NET Core application or a worker service, depending on your requirements. Quartz.NET will run in the background in the same way as the `IHostedService` implementations do. In fact, Quartz.NET uses the `IHostedService` abstractions to schedule and run jobs.

DEFINITION A *job* in Quartz.NET is a task to be executed which implements the `IJob` interface. It is where you define the logic of your tasks to execute.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

In this section you'll see how to install Quartz.NET into a worker service. You'll install the necessary dependencies and configure the Quartz.NET scheduler to run as a background service in a worker service app. In section 22.3.2 we'll convert the exchange-rate downloader task from section 22.1 to a Quartz.NET `IJob` and configure triggers to run on a schedule.

> **NOTE** The instructions in this section can be used to install Quartz.NET into either a worker service or a full ASP.NET Core application. The only difference is whether you use the ConfigueServices() method in Program.cs or Startup.cs, respectively.

To install Quartz.NET:

1. Install the Quartz.AspNetCore NuGet package in your project by running `dotnet add package Quartz.AspNetCore`, by using the NuGet explorer in Visual Studio, or by adding a `<PackageReference>` element to your project file as shown below

```
<PackageReference Include="Quartz.AspNetCore" Version="3.1.0" />
```

2. Add the Quartz.NET `IHostedService` scheduler by calling `AddQuartzServer()` on the `IServiceCollection` in `ConfigureServices`, as shown below. Set `WaitForJobsToComplete=true` so that your app will wait for any jobs in progress to finish when shutting down

```
services.AddQuartzServer(q => q.WaitForJobsToComplete = true);
```

3. Configure the required Quartz.NET services in `ConfigureServices`. The example in the following listing configures the Quartz.NET job factory to retrieve job implementations from a scoped DI container, and adds a required service.

Listing 22.8 Configuring Quartz.NET in a ConfigureServices

```
public void ConfigureServices(IService collection)                #A
{
    services.AddQuartz(q =>                                        #B
    {
        q.UseMicrosoftDependencyInjectionScopedJobFactory();       #C
        q.UseSimpleTypeLoader();                                   #D
    });

    services.AddQuartzServer(q => q.WaitForJobsToComplete = true);    #E
}
```

#A Add Quartz.NET in Startup.cs for ASP.NET Core apps or in Program.cs for worker services
#B Register Quartz.NET services with the DI container
#C Configure Quartz.NET to load jobs from a scoped DI container
#D Required configuration for Quartz.NET internals
#E Add the Quartz.NET IHostedService that runs the Quartz.NET scheduler

This configuration registers all Quartz.NET's required components, so you can now run your application, using `dotnet run` or by pressing F5 in Visual Studio. When your app starts, the

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Quartz.NET `IHostedService` starts its scheduler, as shown in figure 22.4. We haven't configured any jobs to run yet, so the scheduler doesn't have anything to schedule yet.

Quartz.NET uses an in-memory store for tracking jobs and schedules by default.

Quartz.NET runs in non-clustered mode by default, so each running instance of your app is independent.

No jobs or triggers have been configured for this application.

```
Command Prompt (light)
info: Quartz.Core.SchedulerSignalerImpl[0]
      Initialized Scheduler Signaller of type: Quartz.Core.SchedulerSignalerImpl
info: Quartz.Core.QuartzScheduler[0]
      Quartz Scheduler v.3.1.0.0 created.
info: Quartz.Core.QuartzScheduler[0]
      JobFactory set to: Quartz.MicrosoftDependencyInjectionScopedJobFactory
info: Quartz.Simpl.RAMJobStore[0]
      RAMJobStore initialized.
info: Quartz.Core.QuartzScheduler[0]
      Scheduler meta-data: Quartz Scheduler (v3.1.0.0) 'QuartzScheduler' with instanceId 'NON_CLUSTERED'
  Scheduler class: 'Quartz.Core.QuartzScheduler' - running locally.
  NOT STARTED.
  Currently in standby mode.
  Number of jobs executed: 0
  Using thread pool 'Quartz.Simpl.DefaultThreadPool' - with 10 threads.
  Using job-store 'Quartz.Simpl.RAMJobStore' - which does not support persistence. and is not clustered.

info: Quartz.Impl.StdSchedulerFactory[0]
      Quartz scheduler 'QuartzScheduler' initialized
info: Quartz.Impl.StdSchedulerFactory[0]
      Quartz scheduler version: 3.1.0.0
info: Quartz.ContainerConfigurationProcessor[0]
      Adding 0 jobs, 0 triggers.
info: Quartz.Core.QuartzScheduler[0]
      Scheduler QuartzScheduler_$_NON_CLUSTERED started.
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
```

Figure 22.4 The Quartz.NET scheduler starts on app startup and logs its configuration. The default configuration stores the list of jobs and their schedules in memory and runs in a non-clustered state. In this example you can see that no jobs or triggers have been registered, so the scheduler has nothing to schedule yet.

> **TIP** **Running your application *before* you've added any jobs is a good practice to check that you have installed and configure Quartz.NET correctly before you get to more advanced configuration.**

A job scheduler without any jobs to schedule isn't a lot of use, so in the next section we'll create a job and add a trigger for it to run on a timer.

22.3.2 Configuring a job to run on a schedule with Quartz.NET

In section 22.1 we created an `IHostedService` that downloads exchange rates from a remote service and saves the results to a database using EF Core. In this section you'll see how you can create a similar Quartz.NET `IJob` and configure it to run on a schedule.

The following listing shows an implementation of `IJob` which downloads the latest exchange rates from a remote API using a typed client, `ExchangeRatesClient`. The results are then saved using an EF Core `DbContext`, `AppDbContext`.

Listing 22.9 A Quartz.NET IJob for downloading and saving exchange rates

```
public class UpdateExchangeRatesJob : IJob                             #A
{
    private readonly ILogger<UpdateExchangeRatesJob> _logger;          #B
    private readonly ExchangeRatesClient _typedClient;                 #B
```

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
    private readonly AppDbContext _dbContext;                          #B
    public UpdateExchangeRatesJob(                                     #B
        ILogger<UpdateExchangeRatesJob> logger,                        #B
        ExchangeRatesClient typedClient,                               #B
        AppDbContext dbContext)                                        #B
    {                                                                  #B
        _logger = logger;                                              #B
        _typedClient = typedClient;                                    #B
        _dbContext = dbContext;                                        #B
    }                                                                  #B

    public async Task Execute(IJobExecutionContext context)            #C
    {
        _logger.LogInformation("Fetching latest rates");

        var latestRates = await _typedClient.GetLatestRatesAsync();    #D

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

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

#A Quartz.NET jobs must implement the IJob interface
#B You can use standard dependency injection to inject any dependencies
#C IJob requires you implement a single asynchronous method, Execute
#D Download the rates from the remote API
#E Save the rates to the database

Functionally, the `IJob` in the previous listing is doing a similar task to the `BackgroundService` implementation I provided 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 your `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 9am-5pm, Monday-Friday).

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

In the following listing, we register the `UpdateExchangeRatesJob` with the DI container using the `AddJob<T>()` method and provide a unique name to identify the job. We also configure a trigger which fires immediately, and then every 5 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();
        q.UseSimpleTypeLoader();

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

        q.AddTrigger(opts => opts                                    #C
            .ForJob(jobKey)                                          #C
            .WithIdentity(jobKey.Name + " trigger")                  #D
            .StartNow()                                              #E
            .WithSimpleSchedule(x => x                               #F
                .WithInterval(TimeSpan.FromMinutes(5))               #F
                .RepeatForever())                                    #F
        );
    });

    services.AddQuartzServer(q => q.WaitForJobsToComplete = true);
}
```

#A Create a unique key for the job, used to associate it with a trigger
#B Add the IJob to the DI container, and associate with the job key
#C Register a trigger for the IJob via the job key
#D Provide a unique name for the trigger for use in logging and in clustered scenarios
#E Fire the trigger as soon as the Quartz.NET scheduler runs on app startup
#F Fire the trigger every five minutes, until the app shuts down.

Simple triggers like the schedule defined above are common, but you can also achieve more complex configurations using other schedules. For example, the following configuration would configure a trigger to fire every week, on a Friday, at 5:30pm:

```
q.AddTrigger(opts => opts
    .ForJob(jobKey)
    .WithIdentity("Update exchange rates trigger")
    .WithSchedule(CronScheduleBuilder
        .WeeklyOnDayAndHourAndMinute(DayOfWeek.Friday, 17, 30))
```

You can configure a wide array of time and calendar-based triggers with Quartz.NET. You can also control how Quartz.NET handles "missed triggers"—that is, triggers that should have fired, but your app wasn't running at the time. For a detailed description of the trigger configuration options and more examples, see the Quartz.NET documentation at www.quartz-scheduler.net/documentation/.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

> **TIP** A common problem people run into with long running jobs is that Quartz.NET will keep starting new instances of the job when a trigger fires, even though it's already running! To avoid that, tell Quartz.NET to not start another instance by decorating your `IJob` implementation with the `[DisallowConcurrentExecution]` attribute.

The ability to configure advanced schedules, simple use of dependency injection in background tasks, and the separation of jobs from triggers, is reason enough for me to recommend Quartz.NET as soon as you have anything more than the most basic background service needs. However, the real tipping point is when you need to scale your application for redundancy or performance reasons—that's when Quartz.NET's clustering capabilities make it shine.

22.3.3 Using clustering to add redundancy and to your background tasks

In this section you'll learn how to configure Quartz.NET to persist its configuration to a database. This is a necessary step that enables clustering, so that multiple instances of your application can coordinate to run your Quartz.NET jobs.

As your applications become more popular, you may find you need to run more instances of your app to handle the traffic they receive. If you keep your ASP.NET Core applications stateless, then the process of scaling is relatively simple: the more applications you have, the more traffic you can handle, everything else being equal.

However, scaling applications that use `IHostedService` to run background tasks might *not* be as simple. For example, imagine your application includes the `BackgroundService` that we created in section 22.1.2, which saves exchange rates to the database every 5 minutes. When you're running a single instance of your app, the task runs every 5 minutes as expected.

But what happens if you scale your application, and run 10 instances of it? Every one of those applications will be running the `BackgroundService`, and they'll *all* be updating every 5 minutes from the time each instance started!

One option would be to move the `BackgroundService` to a separate worker service app. You could then continue to scale your ASP.NET Core application to handle the traffic as required but deploy a single instance of the worker service. As only a single instance of the `BackgroundService` would be running, the exchange rates would be updated on the correct schedule again.

> **TIP** Differing scaling requirements, as in this example, are one of the best reasons for splitting up bigger apps into smaller "microservices". Breaking up an app like this has a maintenance overhead, however, so think about the tradeoffs if you take this route. For more on this tradeoff, I recommend *Microservices in .NET Core* by Christian Horsdal Gammelgaard (Manning, 2020)

However, if you take this route, you add a hard limitation that you can only *ever* have a single instance of your worker service. If your need to run more instances of your worker service to handle additional load, you'll be stuck.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

An alternative option to enforcing a single service is to use *clustering*. Clustering allows you to run multiple instances of your application, and tasks are distributed between all the instances of your application. Quartz.NET achieves clustering by using a database as a backing store. When a trigger indicates a job needs to execute, the Quartz.NET schedulers in each app attempt to obtain a lock to execute the job, as shown in figure 22.5. Only a single app can be successful, ensuring that a single app handles the trigger for the `IJob`.

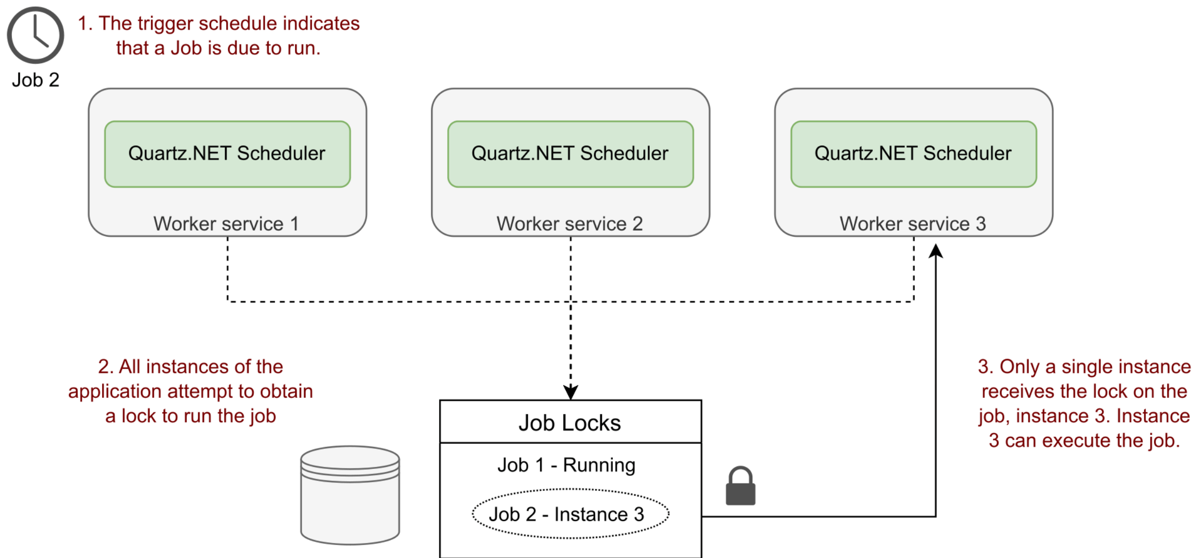


Figure 22.5 Using clustering with Quartz.NET allows horizontal scaling. Quartz.NET uses a database as a backing store, ensuring that only a single instance of the application handles a trigger at a time. This makes it possible to run multiple instances of your application to meet scalability requirements.

Quartz.NET relies on a persistent database for its clustering functionality. Quartz.NET stores descriptions of the jobs and triggers in the database, including when the trigger last fired for example. It’s the “locking” features of the database that ensure only a single application can execute a task at a time.

> **TIP** You can enable persistence without enabling clustering. This allows the Quartz.NET scheduler to “catch up” with missed triggers.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

The following listing shows how to enable persistence for Quartz.NET, and, additionally, how to enable clustering. The example below stores data in an MS SQLServer (or LocalDB) server, but Quartz.NET supports many other databases. The example below uses the recommended values for enabling clustering and persistence as outlined in the documentation.[104]

Listing 22.11 Enabling persistence and clustering for Quartz.NET

```
public void ConfigureServices(IService collection)                  #A
{
    var connectionString = Configuration                            #B
        .GetConnectionString("DefaultConnection");                  #B

    services.AddQuartz(q =>
    {
        q.SchedulerId = "AUTO";                                     #C

        q.UseMicrosoftDependencyInjectionScopedJobFactory();
        q.UseSimpleTypeLoader();

        q.UsePersistentStore(s =>                                   #D
        {
           s.UseSqlServer(connectionString);                        #E
           s.UseClustering();                                       #F
           s.UseProperties = true;                                  #G
           s.UseJsonSerializer();                                   #G
        });

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

        q.AddTrigger(opts => opts
            .ForJob(jobKey)
            .WithIdentity(jobKey.Name + " trigger")
            .StartNow()
            .WithSimpleSchedule(x => x
                .WithInterval(TimeSpan.FromMinutes(5))
                .RepeatForever())
        );
    });

    services.AddQuartzServer(q => q.WaitForJobsToComplete = true);
}
```

#A Configuration is identical for both ASP.NET Core apps and worker services
#B Obtain the connection string for your database from configuration
#C Each instance of your app must have a unique SchedulerId. AUTO takes care of this for you
#D Enable database persistence for the Quartz.NET scheduler data
#E Store the scheduler data in a SQLServer (or LocalDb) database
#F Enables clustering between multiple instances of your app

---

[104] The Quartz.NET documentation discusses many configuration setting controls for persistence (https://www.quartz-scheduler.net/documentation/quartz-3.x/tutorial/job-stores.html).

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

#G Adds the recommended configuration for job persistence

With this configuration, Quartz.NET stores a list of jobs and triggers in the database and uses database locking to ensure only a single instance of your app "handles" a trigger and runs the associated job.

> **NOTE** SQLite doesn't support the database locking primitives required for clustering. You can use SQLite as a persistence store, but you won't be able to use clustering.

Quartz.NET stores data in your database, but it doesn't attempt to create the tables it uses itself. Instead, you must manually add the required tables. Quartz.NET provides SQL scripts for all of the supported database server types on GitHub, including MS SQL Server, SQLite, PostgreSQL, MySQL, and many more: https://github.com/quartznet/quartznet/tree/master/database/tables

> **TIP** If you're using EF Core migrations to manage your database, I suggest using them even for "ad hoc" scripts like these. In the code sample associated with this chapter, you can see a migration that creates the required tables using the Quartz.NET scripts.

Clustering is one of those advanced features that is only necessary as you start to scale your application, but it's an important tool to have in your belt. It gives you the ability to safely scale your services as you add more jobs. There are some important things to bear in mind however, so I suggest reading through the warnings in the Quartz.NET documentation https://www.quartz-scheduler.net/documentation/quartz-3.x/tutorial/advanced-enterprise-features.html.

That brings us to the end of this chapter on background services. In the final chapter of this book, I'll describe an important aspect of web development which, sometimes despite the best of intentions, is often left until last: testing. You'll learn how to write simple unit tests for your classes, how to design for testability, and how to build integration tests that test your whole app.

22.4 Summary

- You can use the `IHostedService` interface to run tasks in the background of your ASP.NET Core apps. Call `AddHostedSerice<T>()` to add an implementation `T` to the DI container. `IHostedService` is useful for implementing long-running tasks.
- Typically, you should derive from `BackgroundService` to create an `IHostedService`, as this implements best-practices required for long-running tasks. You must override a single method, `ExecuteAsync`, which is called when your app starts. You should run your tasks within this method until the provided `CancellationToken` indicates the app is shutting down.
- You can manually create DI scopes using `IServiceProvider.CreateScope()`. This is useful for accessing *scoped* lifetime services from within a *singleton* lifetime

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

component, for example from an IHostedService implementation.

- A *worker service* is a .NET Core application that uses the generic IHost but doesn't include the ASP.NET Core libraries for handling HTTP requests. They generally have a smaller memory and disk footprint than the ASP.NET Core equivalent.
- Worker services use the same logging, configuration, and dependency injection systems as ASP.NET Core apps. However, they don't use the Startup.cs file, so you must configure your DI services in IHostBuilder.ConfigureServices().
- To run a worker service or ASP.NET Core app as a Windows Service, add the Microsoft.Extensions.Hosting.WindowsServices NuGet package, and call AddWindowsService() on IHostBuilder. You can install and manage your app with the Windows sc utility.
- To install a Linux systemd daemon, add the Microsoft.Extensions.Hosting.Systemd NuGet package, and call AddSystemd() on IHostBuilder. Both the Systemd and Windows Service integration packages do nothing when running the application as a console app, which is great for testing your app.
- Quartz.NET runs jobs based on triggers using advanced schedules. It builds on the IHostedService implementation to add extra features and scalability. You can install Quartz by adding the Quartz.AspNetCore NuGet package, and calling AddQuartz() and AddQuartzServer() in ConfigureServices().
- You can create a Quartz.NET job by implementing the IJob interface. This requires implementing a single method, Execute. You can enable DI for the job by calling UseMicrosoftDependencyInjectionScopedJobFactory in AddQuartz(). This allows you to directly inject scoped (or transient) services into your job, without having to create your own scopes.
- You must register your job, T, with DI by calling AddJob<T>() and providing a JobKey name for the job. You can add an associated trigger by calling AddTrigger() and providing the JobKey. Triggers have a wide variety of schedules available for controlling when a job should be executed.
- By default, triggers will continue to spawn new instances of a job as often as necessary. For long running jobs scheduled with a short interval that will result in many instances of your app running concurrently. If you only want a trigger to execute a job when an instance is not already running, decorate your job with the [DisallowConcurrentExecution] attribute.
- Quartz.NET supports database persistence for storing when triggers have executed. To enable persistence, call UsePersistentStore() in your AddQuartz() configuration method, and configure a database, using UseSqlServer() for example. With persistence, Quartz.NET can persist details about jobs and triggers between application restarts.
- Enabling persistence also allows you to use clustering. Clustering enables multiple apps using Quartz.NET to coordinate, so that jobs are spread across multiple schedulers. To enable clustering, first enable database persistence, and then call UseClustering().

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

23 Testing your application

This chapter covers

- **Creating unit test projects with xUnit**
- **Writing unit tests for custom middleware and API controllers**
- **Using the Test Host package to write integration tests**
- **Testing your real application's behavior with** `WebApplicationFactory`
- **Testing code dependent on EF Core with the in-memory database provider**

When I first started programming, I didn't understand the benefits of automated testing. It involved writing so much more code—wouldn't it be more productive to be working on new features instead? It was only when my projects started getting bigger that I appreciated the advantages. Instead of having to manually run my app and test each scenario, I could press Play on a suite of tests and have my code tested for me automatically.

Testing is universally accepted as good practice, but how it fits into your development process can often turn into a religious debate. How many tests do you need? Is anything less than 100% coverage of your code base adequate? Should you write tests before, during, or after the main code?

This chapter won't address any of those questions. Instead, I focus on the *mechanics* of testing an ASP.NET Core application. I show you how to use isolated *unit tests* to verify the behavior of your services in isolation, how to test your API controllers and custom middleware, and how to create *integration tests* that exercise multiple components of your application at once. Finally, I touch on the EF Core in-memory provider, a feature that lets you test components that depend on a `DbContext` without having to connect to a database.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

TIP For a broader discussion around testing, or if you're brand new to unit testing, see *The Art of Unit Testing, Third Edition* by Roy Osherove (Manning, 2021). Alternatively, for an in-depth look at testing with xUnit in .NET Core, see *.NET Core in Action* by Dustin Metzgar (Manning, 2018).

In section 23.1, I introduce the .NET SDK testing framework, and how you can use it to create unit testing apps. I describe the components involved, including the testing SDK and the testing frameworks themselves, like xUnit and MSTest. Finally, I cover some of the terminology I use throughout the chapter.

In section 23.2, you'll create your first test project. You'll be testing a simple class at this stage, but it'll allow you to come to grips with the various testing concepts involved. You'll create several tests using the xUnit test framework, make assertions about the behavior of your services, and execute the test project both from Visual Studio and the command line.

In sections 23.3 and 23.4, we'll look at how to test common features of your ASP.NET Core apps: API controllers and custom middleware. I show you how to write isolated unit tests for both, much like you would any other service, as well as the tripping points to look out for.

To ensure components work correctly, it's important to test them in isolation. But you also need to test they work correctly in a middleware pipeline. ASP.NET Core provides a handy Test Host package that lets you easily write these *integration tests* for your components. You can even go one step further with the `WebApplicationFactory` helper class, and test that your *app* is working correctly. In section 23.5, you'll see how to use `WebApplicationFactory` to simulate requests to your application, and to verify it generates the correct response.

In the final section of this chapter, I demonstrate how to use the SQLite database provider for EF Core with an in-memory database. You can use this provider to test services that depend on an EF Core `DbContext`, without having to use a real database. That avoids the pain of having a known database infrastructure, of resetting the database between tests, and of different people having slightly different database configurations.

Let's start by looking at the overall testing landscape for ASP.NET Core, the options available to you, and the components involved.

23.1 An introduction to testing in ASP.NET Core

In this section you'll learn about the basics of testing in ASP.NET Core. You'll learn about the different types of tests you can write, such as unit tests and integration tests, and why you should write both types. Finally, you'll see how testing fits into ASP.NET Core.

If you have experience building apps with the full .NET Framework or mobile apps with Xamarin, then you might have some experience with unit testing frameworks. If you were building apps in Visual Studio, exactly how to create a test project would vary between testing frameworks (xUnit, NUnit, MSTest), and running the tests in Visual Studio often required installing a plugin. Similarly, running tests from the command line varied between frameworks.

With the .NET Core SDK, testing in ASP.NET Core and .NET Core is now a first-class citizen, on a par with building, restoring packages, and running your application. Just as you can run

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

`dotnet build` to build a project, or `dotnet run` to execute it, you can use `dotnet test` to execute the tests in a test project, regardless of the testing framework used.

The `dotnet test` command uses the underlying .NET Core SDK to execute the tests for a given project. This is exactly the same as when you run your tests using the Visual Studio test runner, so whichever approach you prefer, the results are the same.

Test projects are console apps that contain a number of *tests*. A test is typically a method that evaluates whether a given class in your app behaves as expected. The test project will typically have dependencies on at least three components:

- The .NET Test SDK
- A unit testing framework, for example xUnit, NUnit, Fixie, or MSTest
- A test-runner adapter for your chosen testing framework, so that you can execute your tests by calling `dotnet test`

These dependencies are normal NuGet packages that you can add to a project, but they allow you to hook in to the `dotnet test` command and the Visual Studio test runner. You'll see an example csproj from a test app in the next section.

Typically, a test consists of a method that runs a small piece of your app in isolation and checks that it has the desired behavior. If you were testing a `Calculator` class, you might have a test that checks that passing the values `1` and `2` to the `Add()` method returns the expected result, `3`.

You can write lots of small, isolated tests like this for your app's classes to verify that each component is working correctly, independent of any other components. Small isolated tests like these are called *unit tests*.

Using the ASP.NET Core framework, you can build apps that you can easily unit test; you can test some aspects of your controllers in isolation from your action filters and model binding. This is because the framework:

- Avoids static types.
- Uses interfaces instead of concrete implementations.
- Has a highly modular architecture; you can test your controllers in isolation from your action filters and model binding, for example.

But just because all your components work correctly independently, doesn't mean they'll work when you put them together. For that, you need *integration tests*, which test the interaction between multiple components.

The definition of an integration test is another somewhat contentious issue, but I think of integration tests as any time you're testing multiple components together, or you're testing large vertical slices of your app. Testing a user manager class that can save values to a database, or testing that a request made to a health-check endpoint returns the expected response, for example. Integration tests don't necessarily include the *entire* app, but they definitely use more components than unit tests.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

NOTE I don't cover UI tests which, for example, interact with a browser to provide true end-to-end automated testing. Selenium (www.seleniumhq.org) and Cypress (www.cypress.io) are two of the best known tools for UI testing.

ASP.NET Core has a couple of tricks up its sleeve when it comes to integration testing. You can use the Test Host package to run an in-process ASP.NET Core server, which you can send requests to and inspect the responses. This saves you from the orchestration headache of trying to spin up a web server on a different process, making sure ports are available and so on, but still allowing you to exercise your whole app.

At the other end of the scale, the EF Core SQLite in-memory database provider lets you isolate your tests from the database. Interacting and configuring a database is often one of the hardest aspects of automating tests, so this provider lets you sidestep the issue entirely. You'll see how to use it in section 23.6.

The easiest way to get to grips with testing is to give it a try, so in the next section, you'll create your first test project and use it to write unit tests for a simple custom service.

23.2 Unit testing with xUnit

In this section, you'll learn how to create unit test projects, how to reference classes in other projects, and how to run tests with Visual Studio or the .NET CLI. You'll create a test project and use it to test the behavior of a basic currency converter service. You'll write some simple unit tests that check that the service returns the expected results, and that it throws exceptions when you expect it to.

As I described in section 23.1, to create a test project you need to use a testing framework. You have many options, such as NUnit or MSTest, but the most commonly used test framework with .NET Core is xUnit (https://xunit.github.io/). The ASP.NET Core framework project itself uses xUnit as its testing framework, so it's become somewhat of a convention. If you're familiar with a different testing framework, then feel free to use that instead.

23.2.1 Creating your first test project

Visual Studio includes a template to create a .NET Core xUnit test project, as shown in figure 23.1. Choose File > New Project and choose xUnit Test Project (.NET Core) from the New Project dialog. Alternatively, you could choose MSTest Test Project (.NET Core) or NUnit Test Project (.NET Core) if you're more comfortable with those frameworks.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

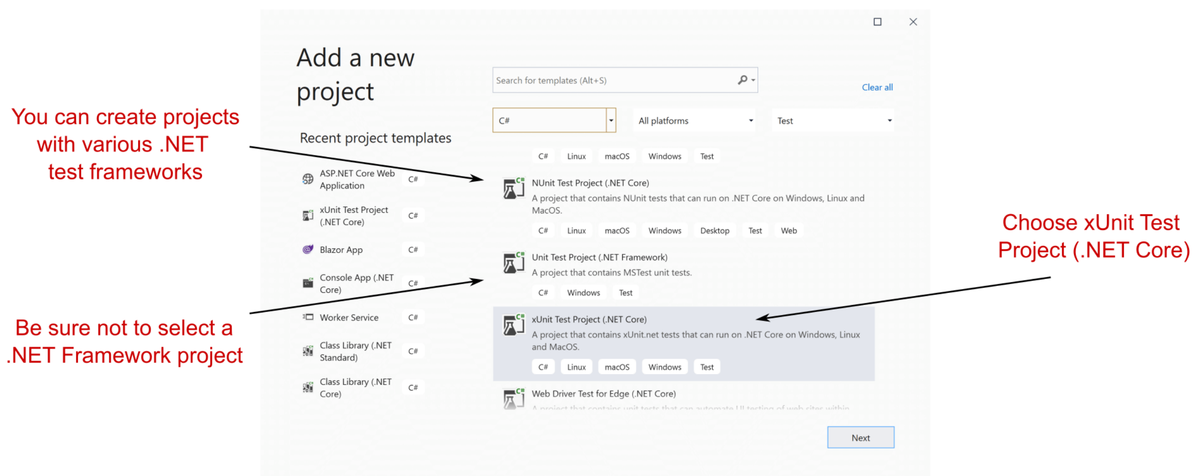


Figure 23.1 The New Project Dialog in Visual Studio. Choose xUnit Test Project to create an xUnit project or choose Unit Test Project to create an MSTest project.

Alternatively, if you're not using Visual Studio, you can create a similar template using the .NET CLI with

```
dotnet new xunit
```

Whether you use Visual Studio or the .NET CLI, the template creates a console project and adds the required testing NuGet packages to your csproj file, as shown in the following listing. If you chose to create an MSTest (or other framework) test project, then the xUnit and xUnit runner packages would be replaced with packages appropriate to your testing framework of choice.

Listing 23.1 The csproj file for an xUnit test project

```
<Project Sdk="Microsoft.NET.Sdk">     #A
  <PropertyGroup>                    #A
    <TargetFramework>netcoreapp3.1</TargetFramework>    #A
    <IsPackable>false</IsPackable>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference
        Include="Microsoft.NET.Test.Sdk" Version="16.5.0" />      #B
    <PackageReference Include="xunit" Version="2.4.0" />      #C
    <PackageReference
        Include="xunit.runner.visualstudio" Version="2.4.0" />      #D
    <PackageReference Include="coverlet.collector" Version="1.2.0" />     #E
  </ItemGroup>
</Project>
```

#A The test project is a standard .NET Core project targeting .NET Core 3.1.
#B The .NET Test SDK, required by all test projects

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

#C The xUnit test framework
#D The xUnit test adapter for the .NET Test SDK
#E An optional package that collects metrics about how much of your code base is covered by tests

In addition to the NuGet packages, the template includes a single example unit test. This doesn't *do* anything, but it's a valid xUnit test all the same, as shown in the following listing. In xUnit, a test is a method on a public class, decorated with a `[Fact]` attribute.

Listing 23.2 An example xUnit unit test, created by the default template

```
public class UnitTest1          #A
{
    [Fact]                      #B
    public void Test1()         #C
    {
    }
}
```

#A xUnit tests must be in public classes.
#B The [Fact] attribute indicates the method is a test method.
#C The Fact must be public and have no parameters.

Even though this test doesn't test anything, it highlights some characteristics of xUnit `[Fact]` tests:

- Tests are denoted by the `[Fact]` attribute.
- The method should be public, with no method arguments.
- The method is `void`. It could also be an `async` method and return `Task`.
- The method resides inside a public, non-static class.

> **NOTE** The `[Fact]` attribute, and these restrictions, are specific to the xUnit testing framework. Other frameworks will use other ways to denote test classes and have different restrictions on the classes and methods themselves.

It's also worth noting that, although I said that test projects are console apps, there's no `Program` class or `static void main` method. Instead, the app looks more like a class library. This is because the test SDK automatically injects a `Program` class at build time. It's not something you have to worry about in general, but you may have issues if you try to add your own Program.cs file to your test project.[105]

Before we go any further and create some useful tests, we'll run the test project as it is, using both Visual Studio and the .NET SDK tooling, to see the expected output.

---

[105] This isn't a common thing to do, but I've seen it used occasionally. I describe this issue in detail, and how to fix it, at http://mng.bz/1NyQ.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

23.2.2 Running tests with dotnet test

When you create a test app that uses the .NET Test SDK, you can run your tests either with Visual Studio or using the .NET CLI. In Visual Studio, you run tests by choosing Tests > Run > All Tests from the main menu, or by clicking Run All in the Test Explorer window, as shown in figure 23.2.

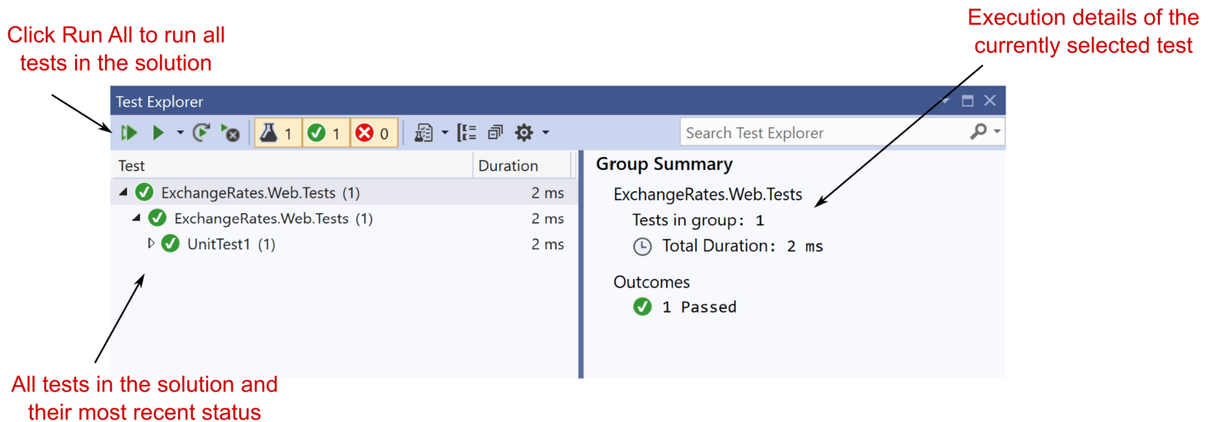


Figure 23.2 The Test Explorer window in Visual Studio lists all tests found in the solution, and their most recent pass/fail status. Click a test in the left-hand pane to see details about the most recent test run in the right-hand pane.

The Test Explorer window lists all the tests found in your solution and the results of each test. In xUnit, a test will pass if it doesn't throw an exception, so `Test1` passed successfully.

Alternatively, you can run your tests from the command line using the .NET CLI by running

```
dotnet test
```

from the unit test project's folder, as shown in figure 23.3.

> **NOTE** You can also run `dotnet test` from the *solution* folder. This will run all test projects referenced in the .sln solution file.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

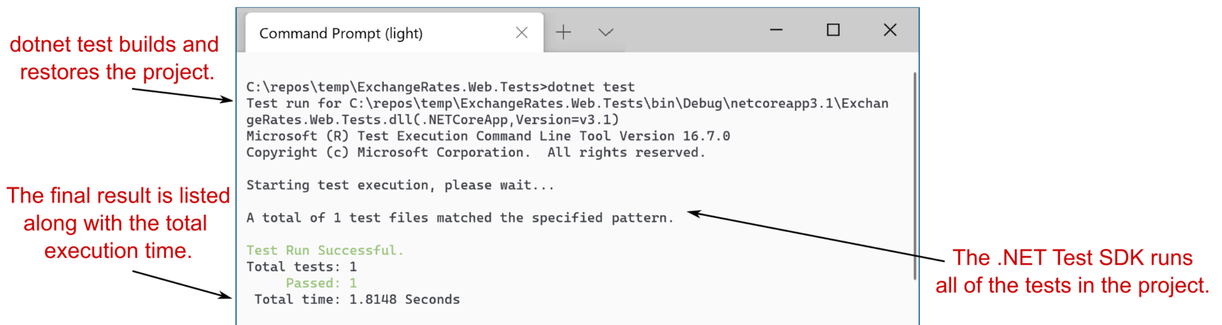


Figure 23.3 You can run tests from the command line using `dotnet test`. This restores and builds the test project before executing all the tests in the project.

Calling `dotnet test` runs a restore and build of your test project and then runs the tests, as you can see from the console output in figure 23.3. Under the hood, the .NET CLI calls into the same underlying infrastructure as Visual Studio does (the .NET Core SDK), so you can use whichever approach better suits your development style.

You've seen a successful test run, so it's time to replace that placeholder test with something useful. First things first, you need something to test.

23.2.3 Referencing your app from your test project

In test-driven development (TDD), you typically write your unit tests before you write the actual class you're testing, but I'm going to take a more traditional route here and create the class to test first. You'll write the tests for it afterwards.

Let's assume you've created an app called ExchangeRates.Web, which is used to convert between different currencies, and you want to add tests for it. You've added a test project to your solution as described in section 23.2.1, so your solution looks like figure 23.4.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

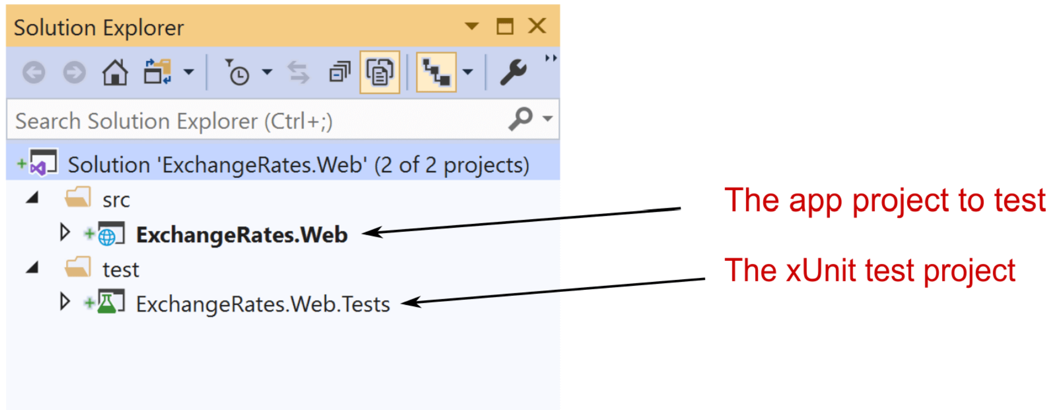


Figure 23.4 A basic solution, containing an ASP.NET Core app called ExchangeRates.Web and a test project called ExchangeRates.Web.Test.

In order for the ExchangeRates.Web.Test project to be able to test the classes in the ExchangeRates.Web project, you need to add a reference to the web project in your test project. In Visual Studio, you can do this by right-clicking the Dependencies node of your test project and choosing Add Reference, as shown in figure 23.5. You can then select the web project from the Add Reference Dialog. After adding it to your project, it shows up inside the Dependencies node, under Projects.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

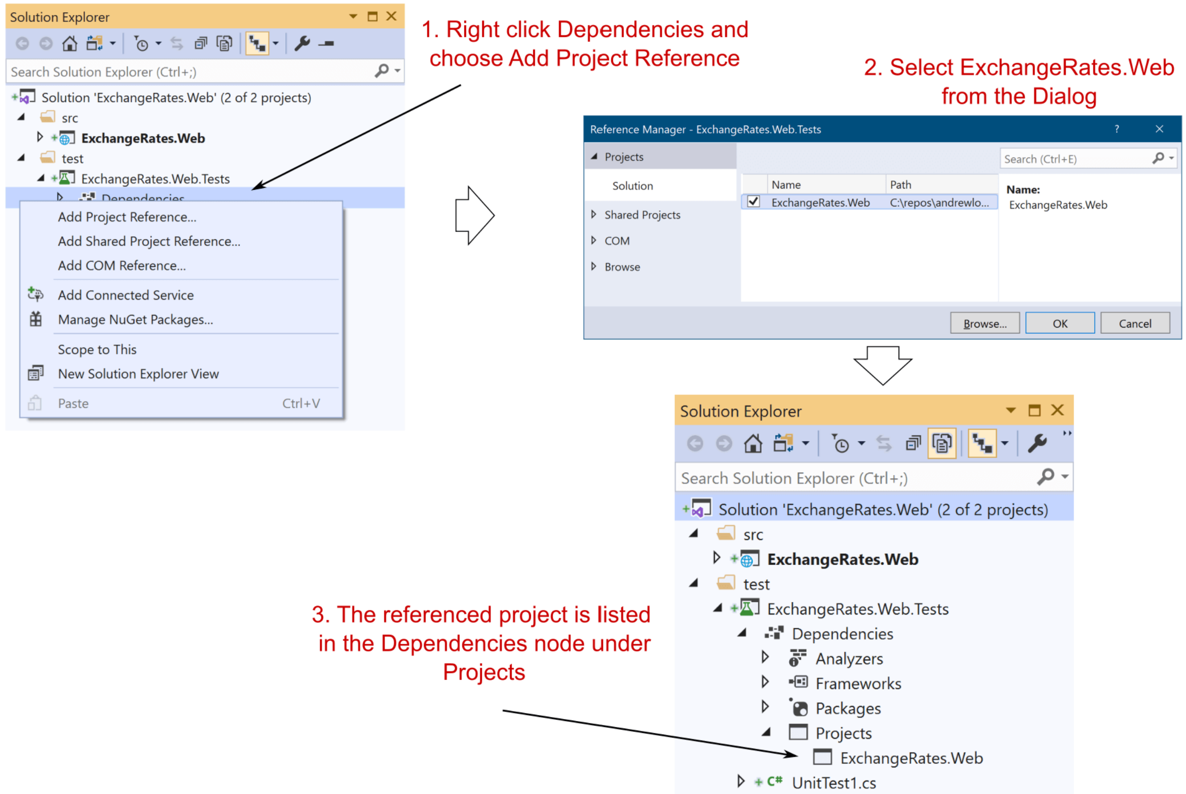


Figure 23.5 To test your app project, you need to add a reference to it from the test project. Right-click the Dependencies node and choose Add Project Reference. The app project is shown referenced inside the Dependencies Node, under Projects.

Alternatively, you can edit the csproj file directly and add a `<ProjectReference>` element inside an `<ItemGroup>` element with the relative path to the referenced project's csproj file.

```
<ItemGroup>
  <ProjectReference
    Include="..\..\src\ExchangeRates.Web\ExchangeRates.Web.csproj" />
</ItemGroup>
```

Note that the path is the *relative* path. A `".."` in the path means the parent folder, so the relative path shown correctly traverses the directory structure for the solution, including both the src and test folders shown in Solution Explorer in figure 23.5.

> **TIP** **Remember, you can edit the csproj file directly in Visual Studio by doubly-clicking the project in Solution Explorer.**

©Manning Publications Co. To comment go to liveBook

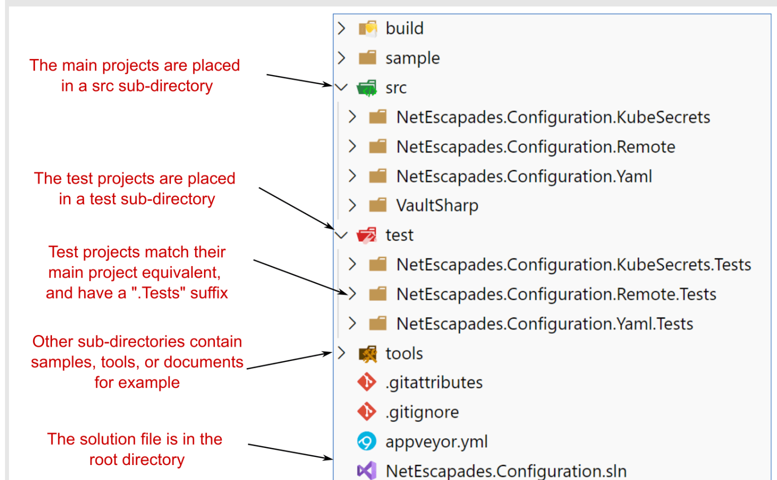
Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Common conventions for project layout

The layout and naming of projects within a solution is completely up to you, but ASP.NET Core projects have generally settled on a couple of conventions that differ slightly from the Visual Studio File New defaults. These conventions are used by the ASP.NET team on GitHub, as well as many other open source C# projects.

The following figure shows an example of these layout conventions. In summary, these are:

- The .sln solution file is in the root directory.
- The main projects are placed in a src sub-directory.
- The test projects are placed in a test or tests sub-directory.
- Each main project has a test project equivalent, named the same as the associated main project with a ".Test" or ".Tests" suffix.
- Other folders, such as samples, tools, or docs contain sample projects, tools for building the project, or documentation.



Conventions around project structures have emerged in the ASP.NET Core framework libraries and open source projects on GitHub. You don't have to follow them for your own project, but it's worth being aware of them.

Whether or not you choose to follow these conventions is entirely up to you, but it's good to be aware of them at least, so you can easily navigate other projects on GitHub.

Your test project is now referencing your web project, so you can write tests for classes in the web project. You're going to be testing a simple class used for converting between currencies, as shown in the following listing.

Listing 23.3 Example `CurrencyConverter` class to convert currencies to GBP

```
public class CurrencyConverter
{
    public decimal ConvertToGbp(                                #A
        decimal value, decimal exchangeRate, int decimalPlaces)    #A
    {
        if (exchangeRate <= 0)              #B
        {                                  #B
```


Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
            throw new ArgumentException(       #B
               "Exchange rate must be greater than zero",          #B
               nameof(exchangeRate));          #B
        }                        #B
        var valueInGbp = value / exchangeRate;      #C
        return decimal.Round(valueInGbp, decimalPlaces);           #D
    }
}
```

#A The ConvertToGbp method converts a value using the provided exchange rate and rounds it.
#B Guard clause, as only positive exchange rates are valid.
#C Converts the value
#D Rounds the result and returns it

This class only has a single method, `ConvertToGbp()`, which converts a `value` from one currency into GBP, given the provided `exchangeRate`. It then rounds the value to the required number of decimal places and returns it.

> **WARNING** **This class is only a basic implementation. In practice, you'd need to handle arithmetic overflow/underflow for large or negative values, as well as considering other edge cases. This is only for demonstration purposes!**

Imagine you want to convert 5.27 USD to GBP, and the exchange rate from GBP to USD is 1.31. If you want to round to 4 decimal places, you'd call

```
converter.ConvertToGbp(value: 5.27, exchangeRate: 1.31, decimalPlaces: 4);
```

You have your sample application, a class to test, and a test project, so it's about time you wrote some tests.

23.2.4 Adding Fact and Theory unit tests

When I write unit tests, I usually target one of three different paths through the method under test:

- *The happy path*—Where typical arguments with expected values are provided
- *The error path*—Where the arguments passed are invalid and tested for
- *Edge cases*—Where the provided arguments are right on the edge of expected values

I realize this is a broad classification, but it helps me think about the various scenarios I need to consider.[106] Let's start with the happy path, by writing a unit test that verifies that the `ConvertToGbp()` method is working as expected with typical input values.

---

[106] A whole other way to approach testing is property-based testing. This fascinating approach is common in functional programming communities, like F#. You can find a great introduction here https://fsharpforfunandprofit.com/posts/property-based-testing/. That post uses F#, but it is still highly accessible even if you're new to the language.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Listing 23.4 Unit test for `ConvertToGbp` using expected arguments

```
[Fact]                                                          #A
public void ConvertToGbp_ConvertsCorrectly()                    #B
{
    var converter = new CurrencyConverter();                    #C
    decimal value = 3;                                          #D
    decimal rate = 1.5m;                                        #D
    int dp = 4;                                                 #D
    decimal expected = 2;                                       #E

    var actual = converter.ConvertToGbp(value, rate, dp);       #F

    Assert.Equal(expected, actual);                             #G
}
```

#A The [Fact] attribute marks the method as a test method.
#B You can call the test anything you like.
#C The class to test, commonly called the "system under test" (SUT)
#D The parameters of the test that will be passed to ConvertToGbp
#E The result you expect.
#F Executes the method and captures the result
#G Verifies the expected and actual values match. If they don't, this will throw an exception.

This is your first proper unit test, which has been configured using the Arrange, Act, Assert (AAA) style:

- *Arrange*—Define all the parameters and create an instance of the system (class) under test (SUT)
- *Act*—Execute the method being tested and capture the result
- *Assert*—Verify that the result of the Act stage had the expected value

Most of the code in this test is standard C#, but if you're new to testing, the `Assert` call will be unfamiliar. This is a helper class provided by xUnit for making assertions about your code. If the parameters provided to `Assert.Equal()` aren't equal, the `Equal()` call will throw an exception and fail the test. If you change the `expected` variable in listing 23.4 to be `2.5` instead of `2`, for example, and run the test, you can see that Test Explorer shows the failure in figure 23.6.

> **TIP** Alternative assertion libraries such as FluentAssertions (https://fluentassertions.com/) and Shouldly (https://github.com/shouldly/shouldly) allow you to write your assertions in a more natural style, for example `actual.Should().Be(expected)`. These libraries are entirely optional, but I find they make tests more readable and error messages easier to understand.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

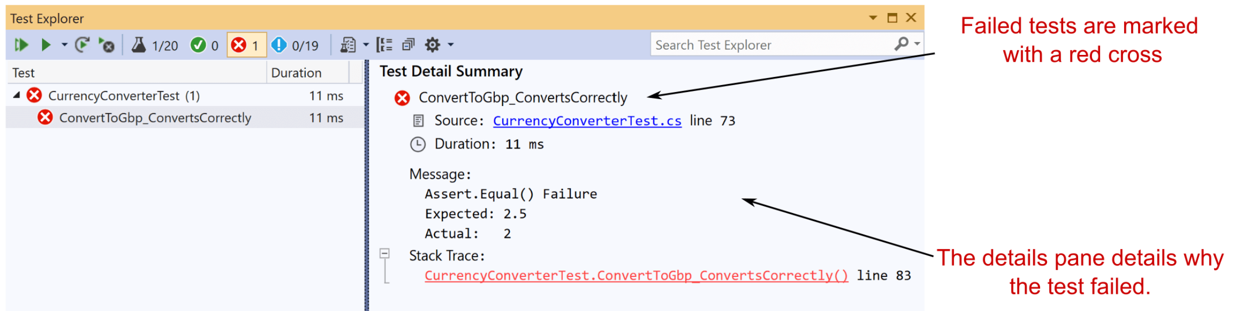


Figure 23.6 When a test fails, it's marked with a red cross in Test Explorer. Clicking the test in the left pane shows the reason for the failure in the right pane. In this case, the expected value was `2.5`, but the actual value was `2`.

> **NOTE** The names of your test class and method are used throughout the test framework to describe your test. You can customize how these are displayed in Visual Studio and in the CLI by configuring an xunit.runner.json file, as described here: https://xunit.github.io/docs/configuring-with-json.html.

In listing 23.4, you chose specific values for `value`, `exchangeRate`, and `decimalPlaces` to test the happy path. But this is only one set of values in an infinite number of possibilities, so you should probably at least test *a few* different combinations.

One way to achieve this would be to copy and paste the test multiple times, tweak the parameters, and change the test method name to make it unique. xUnit provides an alternative way to achieve the same thing without requiring so much duplication.

Instead of creating a `[Fact]` test method, you can create a `[Theory]` test method. A Theory provides a way of parameterizing your test methods, effectively taking your test method and running it multiple times with different arguments. Each set of arguments is considered a different test.

You could rewrite the `[Fact]` test in listing 23.4 to be a `[Theory]` test, as shown next. Instead of specifying the variables in the method body, pass them as parameters to the method, then decorate the method with three `[InlineData]` attributes. Each instance of the attribute provides the parameters for a single run of the test.

Listing 23.5 Theory test for `ConvertToGbp` testing multiple sets of values

```
[Theory]                                                    #A
[InlineData(0, 3, 0)]                                       #B
[InlineData(3, 1.5, 2)]                                     #B
[InlineData(3.75, 2.5, 1.5)]                                #B
public void ConvertToGbp_ConvertsCorrectly (                #C
    decimal value, decimal rate, decimal expected)          #C
{
    var converter = new CurrencyConverter();
    int dps = 4;
```


Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
    var actual = converter.ConvertToGbp(value, rate, dps);  #D

    Assert.Equal(expected, actual);                         #E
}
```

#A Marks the method as a parameterized test
#B Each [InlineData] attribute provides all the parameters for a single run of the test method.
#C The method takes parameters, which are provided by the [InlineData] attributes.
#D Executes the system under test
#E Verifies the result

If you run this `[Theory]` test using `dotnet test` or Visual Studio, then it will show up as three separate tests, one for each set of `[InlineData]`, as shown in figure 23.7.

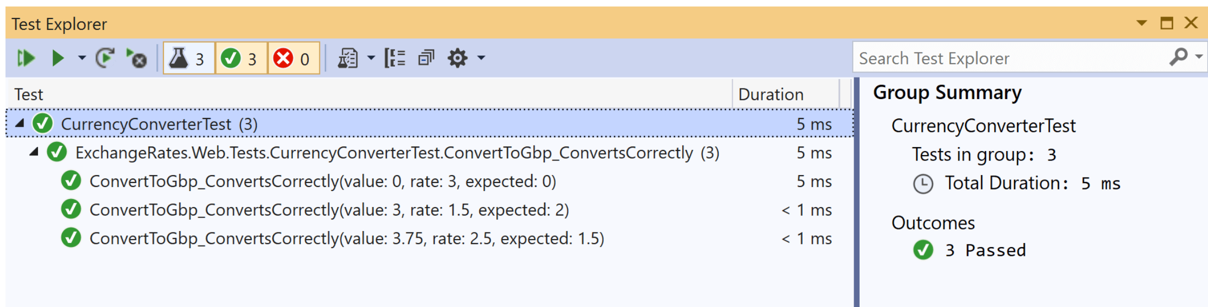


Figure 23.7 Each set of parameters in an `[InlineData]` attribute for a `[Theory]` test creates a separate test run. In this example, a single `[Theory]` has three `[InlineData]` attributes, so it creates three tests, named according to the method name and the provided parameters.

`[InlineData]` isn't the only way to provide the parameters for your theory tests, though it's one of the most commonly used. You can also use a static property on your test class with the `[MemberData]` attribute, or a class itself using the `[ClassData]` attribute.[107]

You now have some tests for the happy path of the `ConvertToGbp()` method, and I even sneaked an edge case into listing 23.5 by testing the case where `value = 0`. The final concept I'll cover is testing error cases, where invalid values are passed to the method under test.

23.2.5 Testing failure conditions

A key part of unit testing is checking that the system under test handles edge cases and errors correctly. For the `CurrencyConverter`, that would mean checking how the class handles negative values, small or zero exchange rates, large values and rates, and so on.

---

[107] I describe how you to use the `[ClassData]` and `[MemberData]` attributes in a blog post at http://mng.bz/8ayP.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Some of these edge cases might be rare but valid cases, whereas other cases might be technically invalid. Calling `ConvertToGbp` with a negative `value` is probably valid; the converted result should be negative too. A negative exchange rate doesn't make sense conceptually, so should be considered an invalid value.

Depending on the design of the method, it's common to throw exceptions when invalid values are passed to a method. In listing 23.3 you saw that we throw an `ArgumentException` if the `exchangeRate` parameter is less than or equal to zero.

xUnit includes a variety of helpers on the `Assert` class for testing whether a method throws an exception of an expected type. You can then make further assertions on the exception, for example to test whether the exception had an expected message.

> **WARNING** **Take care not to tie your test methods too closely to the internal implementation of a method. Doing so can make your tests brittle, where trivial changes to a class break the unit tests.**

The following listing shows a `[Fact]` test to check the behavior of the `ConvertToGbp()` method when you pass it a zero `exchangeRate`. The `Assert.Throws` method takes a lambda function that describes the action to execute, which should throw an exception when run.

Listing 23.6 Using `Assert.Throws<>` to test whether a method throws an exception

```
[Fact]
public void ThrowsExceptionIfRateIsZero()
{
    var converter = new CurrencyConverter();
    const decimal value = 1;
    const decimal rate = 0;                                    #A
    const int dp = 2;
    var ex = Assert.Throws<ArgumentException>(                 #B
        () => converter.ConvertToGbp(value, rate, dp));        #C

    // Further assertions on the exception thrown, ex
}
```

#A An invalid value
#B You expect an ArgumentException to be thrown.
#C The method to execute, which should throw an exception

The `Assert.Throws` method executes the lambda and catches the exception. If the exception thrown matches the expected type, the test will pass. If no exception is thrown or the exception thrown isn't of the expected type, then the `Assert.Throws` method will throw an exception and fail the test.

That brings us to the end of this introduction on unit testing with xUnit. The examples in this section described how to use the new .NET Test SDK, but we didn't cover anything specific to ASP.NET Core. In the rest of this chapter, we'll focus on testing ASP.NET Core projects specifically. We'll start by unit testing middleware.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

23.3 Unit testing custom middleware

In this section you'll learn how to test custom middleware in isolation. You'll see how to test whether your middleware handled a request or whether it called the next middleware in the pipeline. You'll also see how to read the response stream for your middleware.

In chapter 19, you saw how to create custom middleware and how you could encapsulate middleware as a class with an `Invoke` function. In this section, you'll create unit tests for a simple health-check middleware, similar to the one in chapter 19. This is a basic implementation, but it demonstrates the approach you can take for more complex middleware components.

The middleware you'll be testing is shown in listing 23.7. When invoked, this middleware checks that the path starts with `/ping` and, if it does, returns a plain text `"pong"` response. If the request doesn't match, it calls the next middleware in the pipeline (the provided `RequestDelegate`).

Listing 23.7 `StatusMiddleware` to be tested, which returns a `"pong"` response

```
public class StatusMiddleware
{
    private readonly RequestDelegate _next;                    #A
    public StatusMiddleware(RequestDelegate next)              #A
    {
        _next = next;
    }
    public async Task Invoke(HttpContext context)              #B
    {
        if(context.Request.Path.StartsWithSegments("/ping"))   #C
        {                                                      #C
            context.Response.ContentType = "text/plain";       #C
            await context.Response.WriteAsync("pong");         #C
            return;                                            #C
        }                                                      #C
        await _next(context);        #D
    }
}
```

#A The RequestDelegate representing the rest of the middleware pipeline
#B Called when the middleware is executed
#C If the path starts with "/ping", a "pong" response is returned . . .
#D . . . otherwise, the next middleware in the pipeline is invoked.

In this section, you're only going to test two simple cases:

- When a request is made with a path of `"/ping"`
- When a request is made with a different path

> **WARNING** Where possible, I recommend you *don't* directly inspect paths in your middleware like this. A better approach is to use endpoint routing instead, as I discussed in chapter 19. The middleware in this section is for demonstration purposes only.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Middleware is slightly complicated to unit test because the `HttpContext` object is conceptually a *big* class. It contains all the details for the request and the response, which can mean there's a lot of surface area for your middleware to interact with. For that reason, I find unit tests tend to be tightly coupled to the middleware implementation, which is generally undesirable.

For the first test, you'll look at the case where the incoming request `Path` *doesn't* start with `/ping`. In this case, `StatusMiddleware` should leave the `HttpContext` unchanged, and should call the `RequestDelegate` provided in the constructor, which represents the next middleware in the pipeline.

You could test this behavior in several ways, but in listing 23.8 you test that the `RequestDelegate` (essentially a one-parameter function) is executed by setting a local variable to `true`. In the `Assert` at the end of the method, you verify the variable was set and therefore that the delegate was invoked. To invoke `StatusMiddleware`, create and pass in a `DefaultHttpContext`,[108] which is an implementation of `HttpContext`.

Listing 23.8 Unit testing `StatusMiddleware` when a nonmatching `path` is provided

```
[Fact]
public async Task ForNonMatchingRequest_CallsNextDelegate()
{
    var context = new DefaultHttpContext();      #A
    context.Request.Path = "/somethingelse";     #A
    var wasExecuted = false;                     #B
    RequestDelegate next = (HttpContext ctx) =>    #C
    {                                            #C
        wasExecuted = true;                      #C
        return Task.CompletedTask;               #C
    };                                           #C
    var middleware = new StatusMiddleware(next);     #D

    await middleware.Invoke(context);         #E

    Assert.True(wasExecuted);          #F
}
```

#A Creates a DefaultHttpContext and sets the path for the request
#B Tracks whether the RequestDelegate was executed
#C The RequestDelegate representing the next middleware, should be invoked in this example.
#D Creates an instance of the middleware, passing in the next RequestDelegate
#E Invokes the middleware with the HttpContext, should invoke the RequestDelegate
#F Verifies RequestDelegate was invoked

When the middleware is invoked, it checks the provided `Path` and finds that it doesn't match the required value of `/ping`. The middleware therefore calls the next `RequestDelegate` and returns.

---

[108] The `DefaultHttpContext` derives from `HttpContext` and is part of the base ASP.NET Core framework abstractions. If you're so inclined, you can explore the source code for it at https://github.com/dotnet/aspnetcore/blob/v3.1.7/src/Http/Http/src/DefaultHttpContext.cs.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

The other obvious case to test is when the request `Path` is `"/ping"`, and so the middleware should generate an appropriate response. You could test several different characteristics of the response:

- The response should have a `200 OK` status code
- The response should have a `Content-Type` of `text/plain`
- The response body should contain the `"pong"` string

Each of these characteristics represents a different requirement, so you'd typically codify each as a separate unit test. This makes it easier to tell exactly which requirement hasn't been met when a test fails. For simplicity, in listing 23.9 I show all these assertions in the same test.

The positive case unit test is made more complex by the need to read the response body to confirm it contains `"pong"`. `DefaultHttpContext` uses `Stream.Null` for the `Response.Body` object, which means anything written to `Body` is lost. To capture the response and read it out to verify the contents, you must replace the `Body` with a `MemoryStream`. After the middleware executes, you can use a `StreamReader` to read the contents of the `MemoryStream` into a `string` and verify it.

Listing 23.9 Unit testing `StatusMiddleware` when a matching `Path` is provided

```
[Fact]
public async Task ReturnsPongBodyContent()
{
    var bodyStream = new MemoryStream();          #A
    var context = new DefaultHttpContext();       #A
    context.Response.Body = bodyStream;           #A
    context.Request.Path = "/ping";               #B
    RequestDelegate next = (ctx) => Task.CompletedTask;    #C
    var middleware = new StatusMiddleware(next: next);     #C

    await middleware.Invoke(context);                          #D

    string response;                                           #E
    bodyStream.Seek(0, SeekOrigin.Begin);                      #E
    using (var stringReader = new StreamReader(bodyStream))    #E
    {                                                          #E
        response = await stringReader.ReadToEndAsync();        #E
    }                                                          #E

    Assert.Equal("pong", response);                               #F
    Assert.Equal("text/plain", context.Response.ContentType);     #G
    Assert.Equal(200, context.Response.StatusCode);               #H
}
```

#A Creates a DefaultHttpContext and initializes the body with a MemoryStream to capture the response
#B The path is set to the required value for the StatusMiddleware.
#C Creates an instance of the middleware and passes in a simple RequestDelegate
#D Invokes the middleware
#E Rewinds the MemoryStream and reads the response body into a string
#F Verifies the response has the correct value
#G Verifies the Content-Type response is correct
#H Verifies the Status Code response is correct

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

As you can see, unit testing middleware requires a lot of setup to get it working. On the positive side, it allows you to test your middleware in isolation, but in some cases, especially for simple middleware without any dependencies on databases or other services, integration testing can (somewhat surprisingly) be easier. In section 23.5, you'll create integration tests for this middleware to see the difference.

Custom middleware is common in ASP.NET Core projects, but far more common are Razor Pages and API controllers. In the next section, you'll see how you can unit test them in isolation from other components.

23.4 Unit testing API controllers

In this section you'll learn how to unit test API controllers. You'll learn about the benefits and difficulties of testing these components in isolation, and the situations when it can be useful.

Unit tests are all about isolating behavior; you want to test only the logic contained in the component itself, separate from the behavior of any dependencies. The Razor Pages and MVC/API *frameworks* use the filter pipeline, routing, and model binding systems, but these are all *external* to the controller or `PageModel`s. The `PageModel`s and controllers themselves are responsible for only a limited number of things. Typically,

- For invalid requests (that have failed validation, for example), return an appropriate `ActionResult` (API controllers) or redisplay a form (Razor Pages)
- For valid requests, call the required business logic services and return an appropriate `ActionResult` (API controllers), or show or redirect to a success page (Razor Pages).
- Optionally, apply resource-based authorization as required.

Controllers and Razor Pages generally shouldn't contain business logic themselves; instead, they should call out to other services. Think of them more as orchestrators, serving as the intermediary between the HTTP interfaces your app exposes and your business logic services.

If you follow this separation, you'll find it easier to write unit tests for your business logic, and you'll benefit from greater flexibility to change your controllers to meet your needs. With that in mind, there's often a drive to make your controllers and page handlers as thin as possible,[109] to the point where there's not much left to test!

All that said, controllers and actions are classes and methods, so you *can* write unit tests for them. The difficulty is deciding what you want to test! As an example, we'll consider the simple API controller in the following listing, which converts a value using a provided exchange rate, and returns a response.

[109] One of my first introductions to this idea was a series of posts by Jimmy Bogard. The following is a link to the last post in the series, but it contains links to all the earlier posts too. Jimmy Bogard is also behind the MediatR library (https://github.com/jbogard/MediatR), which makes creating thin controllers even easier. See https://lostechies.com/jimmybogard/2013/12/19/put-your-controllers-on-a-diet-posts-and-commands/.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Listing 23.10 The API controller under test

```
[Route("api/[controller]")]
public class CurrencyController : ControllerBase
{
    private readonly CurrencyConverter _converter            #A
       = new CurrencyConverter();                            #A

    [HttpPost]
    public ActionResult<decimal> Convert(InputModel model)    #B
    {
        if (!ModelState.IsValid)                             #C
        {                                                    #C
            return BadRequest(ModelState);                   #C
        }                                                    #C

        decimal result = _convert.ConvertToGbp(model)        #D

        return result;                                       #E
    }
}
```

#A The CurrencyConverter would normally be injected using DI. Created here for simplicity
#B The Convert method returns an ActionResult<T>
#C If the input is invalid, returns a 400 Bad Request result, including the ModelState.
#D If the model is valid, calculate the result
#E Return the result directly

Let’s first consider the happy-path, when the controller receives a valid request. The following listing shows that you can create an instance of the API controller, call an action method, and you’ll receive an `ActionResult<T>` response.

Listing 23.11 A simple API Controller unit test

```
public class CurrencyControllerTest
{
    [Fact]
    public void Convert_ReturnsValue()
    {
        var controller = new CurrencyController();                #A
        var model = new ConvertInputModel                         #A
        {                                                         #A
            Value = 1,                                            #A
            ExchangeRate = 3,                                     #A
            DecimalPlaces = 2,                                    #A
        };                                                        #A

        ActionResult<decimal> result = controller.Convert(model);  #B
        Assert.NotNull(result);                                   #C
    }
}
```

#A Creates an instance of the ConvertController to test and a model to send to the API
#B Invokes the ConvertToGbp method and captures the value returned
#C Asserts that the IActionResult is a ViewResult

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

An important point to note here is that you're *only* testing the return value of the action, the `ActionResult<T>`, *not* the response that's sent back to the user. The process of serializing the result to the response is handled by the MVC formatter infrastructure, as you saw in chapter 9, not by the controller.

When you unit test controllers, you're testing them *separately* from the MVC infrastructure, such as formatting, model binding, routing, and authentication. This is obviously by design, but as with testing middleware in section 23.3, it can make testing some aspects of your controller somewhat complex.

Consider model validation. As you saw in chapter 6, one of the key responsibilities of action methods and Razor Page handlers is to check the `ModelState.IsValid` property and act accordingly if a binding model is invalid. Testing that your controllers and Page Models correctly handle validation failures seems like a good candidate for a unit tests.

Unfortunately, things aren't simple here either. The Razor Page/MVC framework automatically sets the `ModelState` property as part of the model binding process. In practice, when your action method or page handler is invoked in your running app, you know that the `ModelState` will match the binding model values. But in a unit test, there's no model binding, so you must set the `ModelState` yourself manually.

Imagine you're interested in testing the sad path for the controller in listing 23.10, where the model is invalid, and the controller should return `BadRequestObjectResult`. In a unit test, you can't rely on the `ModelState` property being correct for the binding model. Instead, you must *manually* add a model-binding error to the controller's `ModelState` before calling the action, as shown here.

Listing 23.12 Testing handling of validation errors in MVC Controllers

```
[Fact]
public void Convert_ReturnsBadRequestWhenInvalid()
{
    var controller = new CurrencyController();   #A
    var model = new ConvertInputModel            #B
    {                                            #B
        Value = 1,                               #B
        ExchangeRate = -2,                       #B
        DecimalPlaces = 2,                       #B
    };                                           #B

    controller.ModelState.AddModelError(              #C
        nameof(model.ExchangeRate),                   #C
        "Exchange rate must be greater than zero"     #C
    );                                                #C

    ActionResult<decimal> result = controller.Convert(model);    #D

    Assert.IsType<BadRequestObjectResult>(result.Result);        #E
}
```

#A Creates an instance of the Controller to test
#B Creates an invalid binding model by using a negative ExchangeRate
#C Manually adds a model error to the Controller's ModelState. This sets ModelState.IsValid to false.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

#D Invokes the action method, passing in the binding models
#E Verifies the action method returned a BadRequestObjectResult

NOTE In listing 23.12, I passed in an invalid model, but I could just as easily have passed in a *valid* model, or even `null`; the controller doesn't use the binding model if the `ModelState` isn't valid, so the test would still pass. But if you're writing unit tests like this one, I recommend trying to keep your model consistent with your `ModelState`, otherwise your unit tests aren't testing a situation that occurs in practice!

Personally, I tend to shy away from unit testing API controllers directly in this way[110]. As you've seen with model binding, the controllers are somewhat dependent on earlier stages of the MVC framework which you often need to emulate. Similarly, if your controllers access the `HttpContext` (available on the `ControllerBase` base classes), you may need to perform additional setup.

NOTE I haven't discussed Razor Pages much in this section, as they suffer from many of the same problems, in that they are dependent on the supporting infrastructure of the framework. Nevertheless, if you do wish to test your Razor Page `PageModel`, you can read about it here: https://docs.microsoft.com/aspnet/core/test/razor-pages-tests.

Instead of using unit testing, I try to keep my controllers and Razor Pages as "thin" as possible. I push as much of the "behavior" in these classes into business logic services that can be easily unit tested, or into middleware and filters, which can be more easily tested independently.

NOTE This is a personal preference. Some people like to get as close to 100% test coverage for their code base as possible, but I find testing "orchestration" classes is often more hassle than it's worth.

Although I often forgo *unit* testing controllers and Razor Pages, I often write *integration* tests that test them in the context of a complete application. In the next section, we look at ways to write integration tests for your app, so you can test its various components in the context of the ASP.NET Core framework as a whole.

23.5 Integration testing: testing your whole app in-memory

In this section you'll learn how to create integration tests that test component interactions. You'll learn to create a `TestServer` that sends HTTP requests in-memory to test custom middleware components more easily. You'll then learn how to run integration tests for a real application, using your real app's configuration, services, and middleware pipeline. Finally,

[110] You can read more about why I generally don't unit test my controllers here: https://andrewlock.net/should-you-unit-test-controllers-in-aspnetcore/.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

you'll then learn how to use `WebApplicationFactory` to replace services in your app with test versions, to avoid depending on third-party APIs in your tests.

If you search the internet for the different types of testing, you'll find a host of different types to choose from. The differences between them are sometimes subtle and people don't universally agree upon the definitions. I chose not to dwell on it in this book—I consider unit tests to be isolated tests of a component and integration tests to be tests that exercise multiple components at once.

In this section, I'm going to show how you can write integration tests for the `StatusMiddleware` from section 23.3 and the API controller from section 23.4. Instead of isolating the components from the surrounding framework and invoking them directly, you'll specifically test them in a similar context to when you use them in practice.

Integration tests are an important part of confirming that your components function correctly, but they don't remove the need for unit tests. Unit tests are excellent for testing small pieces of logic contained in your components and are typically quick to execute. Integration tests are normally significantly slower, as they require much more configuration and may rely on external infrastructure, such as a database.

Consequently, it's normal to have far more unit tests for an app than integration tests. As you saw in section 23.2, unit tests typically verify the behavior of a component, using valid inputs, edge cases, and invalid inputs, to ensure that the component behaves correctly in all cases. Once you have an extensive suite of unit tests, you'll likely only need a few integration tests to be confident your application is working correctly.

You could write many different types of integration tests for an application. You could test that a service can write to a database correctly, that it can integrate with a third-party service (for sending emails, for example), or that it can handle HTTP requests made to it.

In this section, you're going to focus on the last point, verifying that your app can handle requests made to it, just as you would if you were accessing the app from a browser. For this, you're going to use a useful library provided by the ASP.NET Core team called Microsoft.AspNetCore.TestHost.

23.5.1 Creating a TestServer using the Test Host package

Imagine you want to write some integration tests for the `StatusMiddleware` from section 23.3. You've already written unit tests for it, but you want to have at least one integration test that tests the middleware in the context of the ASP.NET Core infrastructure.

You could go about this in many ways. Perhaps the most complete approach would be to create a separate project and configure `StatusMiddleware` as the only middleware in the pipeline. You'd then need to run this project, wait for it to start up, send requests to it, and inspect the responses.

This would possibly make for a good test, but it would also require a lot of configuration, and be fragile and error-prone. What if the test app can't start because it tries to use an already-taken port? What if the test app doesn't shut down correctly? How long should the integration test wait for the app to start?

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

The ASP.NET Core Test Host package lets you get close to this setup, without having the added complexity of spinning up a separate app. You add the Test Host to your test project by adding the Microsoft.AspNetCore.TestHost NuGet package, either using the Visual Studio NuGet GUI, Package Manager Console, or the .NET CLI. Alternatively, add the `<PackageReference>` element directly to your test project's .csproj file:

```
<PackageReference Include="Microsoft.AspNetCore.TestHost" Version="3.1.7"/>
```

In a typical ASP.NET Core app, you create a `HostBuilder` in your `Program` class, configure a web server (Kestrel) and define your application's configuration, services, and middleware pipeline (using a Startup file). Finally, you call `Build()` on the `HostBuilder` to create an instance of an `IHost` that can be run and will listen for requests on a given URL and port.

The Test Host package uses the same `HostBuilder` to define your test application, but instead of listening for requests at the network level, it creates an `IHost` that uses in-memory request objects instead, as shown in figure 23.8. It even exposes an `HttpClient` that you can use to send requests to the test app. You can interact with the `HttpClient` as though it were sending requests over the network, but in reality, the requests are kept entirely in memory.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

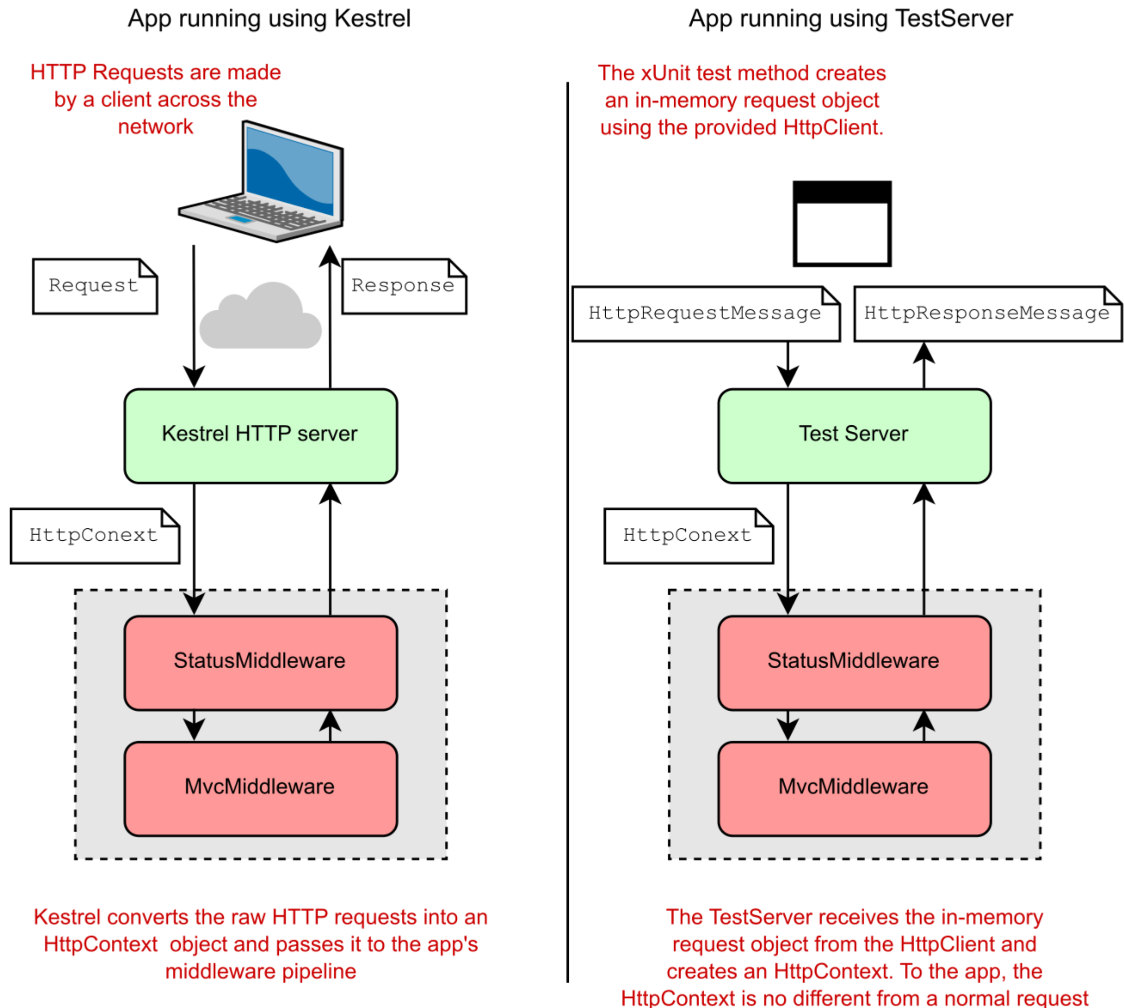


Figure 23.8 When your app runs normally, it uses the Kestrel server. This listens for HTTP requests and converts the requests into an `HttpContext`, which is passed to the middleware pipeline. The `TestServer` doesn't listen for requests on the network. Instead, you use an `HttpClient` to make in-memory requests. From the point of view of the middleware, there's no difference.

Listing 23.13 shows how to use the Test Host package to create a simple integration test for the `StatusMiddleware`. First, create a `HostBuilder` and call `ConfigureWebHost()` to define your application by adding middleware in the `Configure` method. This is equivalent to the `Startup.Configure()` method you would typically use to configure your application.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Call the `UseTestServer()` extension method in `ConfigureWebHost()`, which replaces the default Kestrel server with the `TestServer` from the Test Host package. The `TestServer` is the main component in the Test Host package, which makes all the magic possible. After configuring the `HostBuilder`, call `StartAsync()` to build and start the test application. You can then create an `HttpClient` using the extension method `GetTestClient()`. This returns an `HttpClient` configured to make in-memory requests to the `TestServer`.

Listing 23.13 Creating an integration test with `TestServer`

```
public class StatusMiddlewareTests
{
    [Fact]
    public async Task StatusMiddlewareReturnsPong()
    {
        var hostBuilder = new HostBuilder()                         #A
            .ConfigureWebHost(webHost =>                            #A
            {
                webHost.Configure(app =>                            #B
                    app.UseMiddleware<StatusMiddleware>());         #B
                webHost.UseTestServer();                            #C
            });

        IHost host = await hostBuilder.StartAsync();                #D
        HttpClient client = host.GetTestClient();                   #E

        var response = await client.GetAsync("/ping");              #F

        response.EnsureSuccessStatusCode();                          #G
        var content = await response.Content.ReadAsStringAsync();    #H
        Assert.Equal("pong", content);                               #H
    }
}
```

#A Configures a HostBuilder to define the in-memory test app
#B Add the StatusMiddleware as the only middleware in the pipeline
#C Configure the host to use the TestServer instead of Kestrel
#D Build and start the host
#E Creates an HttpClient, or you can interact directly with the server object
#F Makes an in-memory request, which is handled by the app as normal.
#G Verifies the response was a success (2xx) status code
#H Reads the body content and verifies that it contained "pong"

This test ensures that the test application defined by `HostBuilder` returns the expected value when it receives a request to the `/ping` path. The request is entirely in-memory, but from the point of view of `StatusMiddleware`, it's the same as if the request came from the network.

The `HostBuilder` configuration in this example is simple. Even though I've called this an integration test, you're specifically testing the `StatusMiddleware` on its own, rather than in the context of a "real" application. In many ways, I think this setup is preferable for testing custom middleware compared to the "proper" unit-tests I showed in section 23.3.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Regardless of what you call it, this test relies on very simple configuration for the test app. You may also want to test the middleware in the context of your *real* application, so that the result is representative of your app's *real* configuration.

If you want to run integration tests based on an existing app, then you don't want to have to configure the test `HostBuilder` manually like you did in listing 23.13. Instead, you can use another helper package, Microsoft.AspNetCore.Mvc.Testing.

23.5.2 Testing your application with WebApplicationFactory

Building up a `HostBuilder` and using the Test Host package, as you did in section 23.5.1, can be useful when you want to test isolated "infrastructure" components, such as middleware. It's also very common to want to test your "real" app, with the full middleware pipeline configured, as well as all the required services added to DI. This gives you the most confidence that your application is going to work in production.

The `TestServer` that provides the in-memory server can be used for testing your "real" app, but, in principle, there's a lot more configuration required. Your real app likely loads configuration files or web static files from disk, and may use Razor Pages and views. Prior to .NET Core 2.1, configuring all of these was cumbersome. Thankfully, the introduction of the Microsoft.AspNetCore.Mvc.Testing package and `WebApplicationFactory` largely solves these configuration issues for you.

You can use the `WebApplicationFactory` class (provided by the Microsoft.AspNetCore.Mvc.Testing NuGet package) to run an in-memory version of your real application. It uses the `TestServer` behind the scenes, but it uses your app's real configuration, DI service registration, and middleware pipeline. For example, the following listing shows an example that tests that when your application receives a `"/ping"` request, it responds with `"pong"`.

Listing 23.14 Creating an integration test with `TestServer`

```
public class IntegrationTests:              #A
    IClassFixture<WebApplicationFactory<Startup>>       #A
{
    private readonly WebApplicationFactory<Startup> _fixture;       #B
    public StatusMiddlewareWebApplicationFactoryTests(     #B
        WebApplicationFactory<Startup> fixture)       #B
    {
        _fixture = fixture;       #B
    }

    [Fact]
    public async Task PingRequest_ReturnsPong()
    {
        HttpClient client = _fixture.CreateClient();     #C

        var response = await client.GetAsync("/ping");     #D

        response.EnsureSuccessStatusCode();        #D
        var content = await response.Content.ReadAsStringAsync();    #D
        Assert.Equal("pong", content);           #D
```


Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
    }
}
```

#A Your test must implement the marker interface
#B Inject an instance of WebApplicationFactory<T>, where T is a class in your app
#C Create an HttpClient that sends request to the in-memory TestServer
#D Make requests and verify the response as before

One of the advantages of using `WebApplicationFactory` as shown in listing 23.14 is that it requires *less* manual configuration than using the `TestServer` directly, as shown in listing 23.13, despite performing *more* configuration behind the scenes. The `WebApplicationFactory` tests your app using the configuration defined in your Program.cs and Startup.cs files.

Listings 23.14 and 23.13 are *conceptually* quite different too. Listing 23.13 tests that the `StatusMiddleware` behaves as expected, in the context of a dummy ASP.NET Core app; listing 23.14 tests that *your app behaves as expected for a given input*. It doesn't say anything specific about *how* that happens. Your app doesn't have to use the `StatusMiddleware` for the test in listing 23.14 to pass, it just has to respond correctly to the given request. That means the test knows less about the internal implementation details of your app, and is only concerned with its *behavior*.

> **DEFINITION** **Tests that fail whenever you change your app slightly are called *brittle* or *fragile*. Try to avoid brittle tests by ensuring they aren't dependent on implementation details of your app.**

To create tests that use `WebApplicationFactory`:

- Install the Microsoft.AspNetCore.Mvc.Testing NuGet package in your project by running `dotnet add package Microsoft.AspNetCore.Mvc.Testing`, by using the NuGet explorer in Visual Studio, or by adding a `<PackageReference>` element to your project file as shown below

```
<PackageReference Include="Microsoft.AspNetCore.Mvc.Testing"
    Version="3.1.7" />
```

- Update the `<Project>` element in your test project's .csproj file to the following:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
```

  This is required by `WebApplicationFactory` so that it can find your configuration files and static files

- Implement `IClassFixture<WebApplicationFactory<T>>` in your xUnit test class, where `T` is a class in your real application's project. By convention, you typically use your application's `Startup` class for `T`.
    - `WebApplicationFactory` uses the `T` reference to find your application's `Program.CreateHostBuilder()` method to build an appropriate `TestServer` for tests.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

  - The `IClassFixture<TFixture>` is an xUnit marker interface, that tells xUnit to build an instance of `TFixture` before building the test class and to inject the instance into the test class's constructor. You can read more about fixtures at https://xunit.net/docs/shared-context.
- Accept an instance of `WebApplicationFactory<T>` in your test class's constructor. You can use this fixture to create an `HttpClient` for sending in-memory requests to the `TestServer`. Those requests emulate your application's production behaviour, as your application's real configuration, services, and middleware are all used.

The big advantage of `WebApplicationFactory` is that you can easily test your real app's behavior. That power comes with responsibility—your app will behave just as it would in real life, so it will write to a database and send to third-party APIs! Depending on what you're testing, you may want to replace some of your dependencies to avoid this, as well as to make testing easier.

23.5.3 Replacing dependencies in WebApplicationFactory

When you use `WebbApplicationFactory` to run integration tests on your app, your app will be running in-memory, but other than that, it's as through you're running your application using `dotnet run`. That means, any connection strings, secrets, or API keys that can be loaded locally will also be used to run your application!

> **TIP** **By default, `WebApplicationFactory` uses the `"Development"` hosting environment, the same as when you run locally.**

On the plus side, that means you have a genuine test that your application can start correctly. For example, if you've forgotten to register a required DI dependency that is detected on application startup, any tests that use `WebApplicationFactory` will fail.

On the downside, that means all your tests will be using the same database connection and services as when you run your application locally. It's common to want to replace those with alternative "test" versions of your services.

As a simple example, lets imagine the `CurrencyConverter` that you've been testing in this app uses `IHttpClientFactory` to call a third-party API to retrieve the latest exchange rates. You don't want to hit that API repeatedly in your integration tests, so you want to replace the `CurrencyConverter` with your own `StubCurrencyConverter`.

The first step is to ensure the service `CurrencyConverter` implements an interface, `ICurrencyConverter` for example, and that your app uses this interface throughout, not the implementation. For our simple example, the interface would probably look like the following:

```
public interface ICurrencyConverter
{
    decimal ConvertToGbp(decimal value, decimal rate, int dps);
}
```

You would register the service in `Startup.ConfigureServices()` using:

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
services.AddScoped<ICurrencyConverter, CurrencyConverter>();
```

Now that your application only indirectly depends on `CurrencyConverter`, you can provide an alternative implementation in your tests.

> **TIP** Using an interface decouples your application services from a specific implementation, allowing you to substitute alternative implementations. This is a key practice for making classes testable.

We'll create a simple alternative implementation of `ICurrencyConverter` for our tests, that always returns the same value, `3`. It's obviously not very useful as an actual converter, but that's not the point: you have complete control! Create the following class in your test project:

```
public class StubCurrencyConverter : ICurrencyConverter
{
    public decimal ConvertToGbp(decimal value, decimal rate, int dps)
    {
        return 3;
    }
}
```

You now have all the pieces you need to replace the implementation in your tests. To achieve that, we'll use a feature of `WebApplicationFactory` that lets you customize the DI container before starting the test server.

> **TIP** It's important to remember you *only* want to replace the implementation when running in the test project. I've seen some people try and configure their *real* apps to replace live services for fake services when a specific value is set for example. That is generally unnecessary, bloats your apps with "test" services, and generally adds confusion!

`WebApplicationFactory` exposes a method, `WithWebHostBuilder`, that allows you to customize your application before the in-memory `TestServer` starts. The following listing shows an integration test that uses this builder to replace the "default" `ICurrencyConverter` implementation with our test stub.

Listing 23.15 Replacing a dependency in an integration test using WithWebHostBuilder

```
public class IntegrationTests:                                     #A
    IClassFixture<WebApplicationFactory<Startup>>                  #A
{
    private readonly WebApplicationFactory<Startup> _fixture;      #A
    public StatusMiddlewareWebApplicationFactoryTests(             #A
        WebApplicationFactory<Startup> fixture)                    #A
    {
        _fixture = fixture;                                        #A
    }

    [Fact]
    public async Task ConvertReturnsExpectedValue()
    {
        var customFactory = _fixture.WithWebHostBuilder(             #B
```


Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
            (IWebHostBuilder hostBuilder) =>                      #B
        {
            hostBuilder.ConfigureTestServices(services =>         #C
            {
                services.RemoveAll<ICurrencyConverter>();         #D
                services.AddSingleton
                    <ICurrencyConverter, StubCurrencyConverter>(); #E
            });
        });

        HttpClient client = customFactory.CreateClient();         #F

        var response = await client.GetAsync("/api/currency");    #G

        response.EnsureSuccessStatusCode();                       #G
        var content = await response.Content.ReadAsStringAsync(); #G

        Assert.Equal("3", content);                               #H
    }
}
```

#A Implement the required interface, and inject it into the constructor
#B Create a custom factory with the additional configuration
#C ConfigureTestServices executes after all other DI services are configured in your real app
#D Removes all implementations of ICurrencyConverter from the DI container
#E Adds the test service as a replacement
#F Calling CreateClient bootstraps the application and starts the TestServer
#G Invoke the currency converter endpoint
 # H As the test converter always returns 3, so does the API endpoint

There are a couple of important points to note in this example

- `WithWebHostBuilder()` returns a *new* `WebApplicationFactory` instance. The new instance has your custom configuration, while the original injected `_fixture` instance remains unchanged.
- `ConfigureTestServices()` is called after your real app's `ConfigureServices()` method. That means you can replace services that have been previously been registered. You can also use this to override configuration values, as you'll see in section 23.6.

`WithWebHosBuilder()` is handy when you want to replace a service for a single test. But what if you wanted to replace the `ICurrencyConverter` in every test. All that boilerplate would quickly become cumbersome. Instead, you can create a custom `WebApplicationFactory`.

23.5.4 Reducing duplication by creating a custom WebApplicationFactory

If you find yourself writing `WithWebHosBuilder()` a lot in your integration tests, it might be worth creating a custom `WebApplicationFactory` instead. The following listing shows how to centralize the test service we used in listing 23.15 into a custom `WebApplicationFactory`.

Listing 23.16 Creating a custom `WebApplicationFactory` to reduce duplication

```
public class CustomWebApplicationFactory                          #A
```

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
    : WebApplicationFactory<Startup>                               #A
{
    protected override void ConfigureWebHost(                      #B
        IWebHostBuilder builder)                                   #B
    {
        builder.ConfigureTestServices(services =>                  #C
        {                                                          #C
            services.RemoveAll<ICurrencyConverter>();              #C
            services.AddSingleton                                  #C
                <ICurrencyConverter, StubCurrencyConverter>();     #C
        });                                                    #C
    }
}
```

#A Derive from WebApplicationFactory
#B There are many functions available to override. This is equivalent to calling WithWebHostBuilder
#C Add custom configuration for your application

In this example, we override `ConfigureWebHost` and configure the test services for the factory[111]. You can use your custom factory in any test by injecting it as an `IClassFixture`, as you have before. For example, the following listing shows how you would update listing 23.15 to use the custom factory defined in listing 23.16.

Listing 23.17 Using a custom WebApplicationFactory in an integration test

```
public class IntegrationTests:                                      #A
    IClassFixture<CustomWebApplicationFactory>                      #A
{
    private readonly CustomWebApplicationFactory _fixture;          #B
    public IntegrationTests(CustomWebApplicationFactory fixture)    #B
    {
        _fixture = fixture;
    }

    [Fact]
    public async Task ConvertReturnsExpectedValue()
    {
        HttpClient client = _fixture.CreateClient();               #C

        var response = await client.GetAsync("/api/currency");

        response.EnsureSuccessStatusCode();
        var content = await response.Content.ReadAsStringAsync();

        Assert.Equal("3", content);                               #D
    }
}
```

#A Implement the IClassFixture interface for the custom factory
#B Inject an instance of the factory in the constructor

---

[111] WebApplicationFactory has many other services you could implement for other scenarios. For details see https://docs.microsoft.com/aspnet/core/test/integration-tests.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

#C The client already contains the test service configuration
#D The result confirms the test service was used

You can also combine your custom `WebApplicationFactory` that substitutes services that you always want to replace, with the `WithWebHostBuilder()` method to override additional services on a per-test basis. That combination gives you the best of both worlds: reduced duplication with the custom factory, and control with the per-test configuration.

Running integration tests using your real app's configuration provides about the closest you'll get to a guarantee that your app is working correctly. The sticking point in that guarantee is nearly always external dependencies, such as third-party APIs and databases.

In the final section of this chapter, we'll look at how to use the SQLite provider for EF Core with an in-memory database. You can use this approach to write tests for services that use an EF Core database context, without needing access to a real database.

23.6 Isolating the database with an in-memory EF Core provider

In this section you'll learn how to write unit tests for code that relies on an EF Core `DbContext`. You'll learn how to create an in-memory database, and the difference between the in-memory provider and the SQLite in-memory provider. Finally, you'll see how to use the in-memory SQLite provider to create fast, isolated tests for code that relies on a `DbContext`.

As you saw in chapter 12, EF Core is an ORM that is used primarily with relational databases. In this section, I'm going to discuss one way to test services that depend on an EF Core `DbContext`, without having to configure or interact with a real database.

> **NOTE** To learn more about testing your EF Core code, see *Entity Framework Core in Action* by Jon P Smith (Manning, 2021), http://mng.bz/k0dz.

The following listing shows a highly stripped-down version of the `RecipeService` you created in chapter 12 for the recipe app. It shows a single method to fetch the details of a recipe using an injected EF Core `DbContext`.

Listing 23.18 `RecipeService` to test, which uses EF Core to store and load entities

```
public class RecipeService
{
    readonly AppDbContext _context;             #A
    public RecipeService(AppDbContext context)  #A
    {                                           #A
        _context = context;                     #A
    }                                           #A
    public RecipeViewModel GetRecipe(int id)
    {
        return _context.Recipes                 #B
            .Where(x => x.RecipeId == id)
            .Select(x => new RecipeViewModel
            {
                Id = x.RecipeId,
                Name = x.Name
```


Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
            })
            .SingleOrDefault();
    }
}
```

#A An EF Core DbContext is injected in the constructor.
#B Uses the DbSet<Recipes> property to load recipes and creates a RecipeViewModel

Writing unit tests for this class is a bit of a problem. Unit tests should be fast, repeatable, and isolated from other dependencies, but you have a dependency on your app's `DbContext`. You probably don't want to be writing to a real database in unit tests, as it would make the tests slow, potentially unrepeatable, and highly dependent on the configuration of the database: a fail on all three requirements!

> **NOTE** Depending on your development environment, you *may* want to use a real database for your integration tests, despite these drawbacks. Using a database like the one you'll use in production increases the likelihood you'll detect any problems in your tests. You can find an example of using Docker to achieve this here: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/multi-container-microservice-net-applications/test-aspnet-core-services-web-apps.

Luckily, Microsoft ships two *in-memory* database providers for this scenario. Recall from chapter 12 that when you configure your app's `DbContext` in `Startup.ConfigureServices()`, you configure a specific database provider, such as SQL Server, for example:

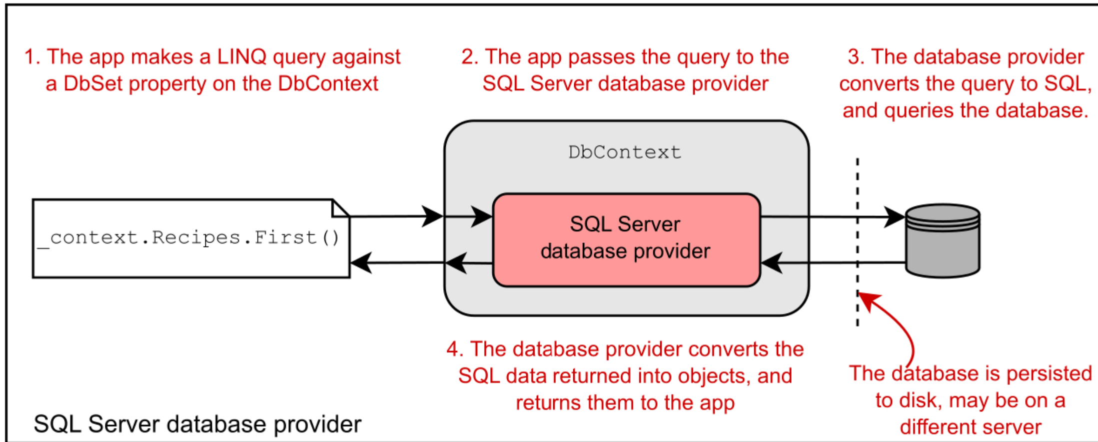
```
services.AddDbContext<AppDbContext>(options =>
   options.UseSqlServer(connectionSttring);
```

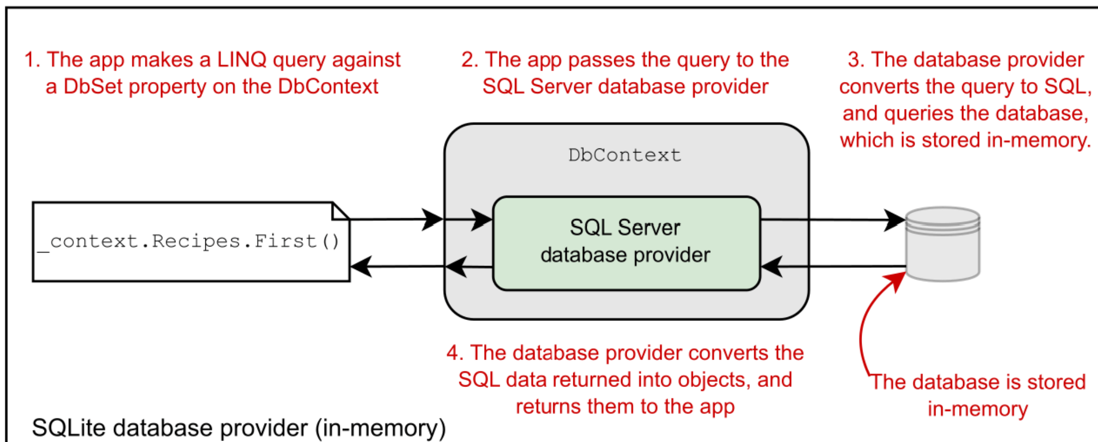
The in-memory database providers are alternative providers designed *only for testing*. Microsoft includes two in-memory providers in ASP.NET Core:

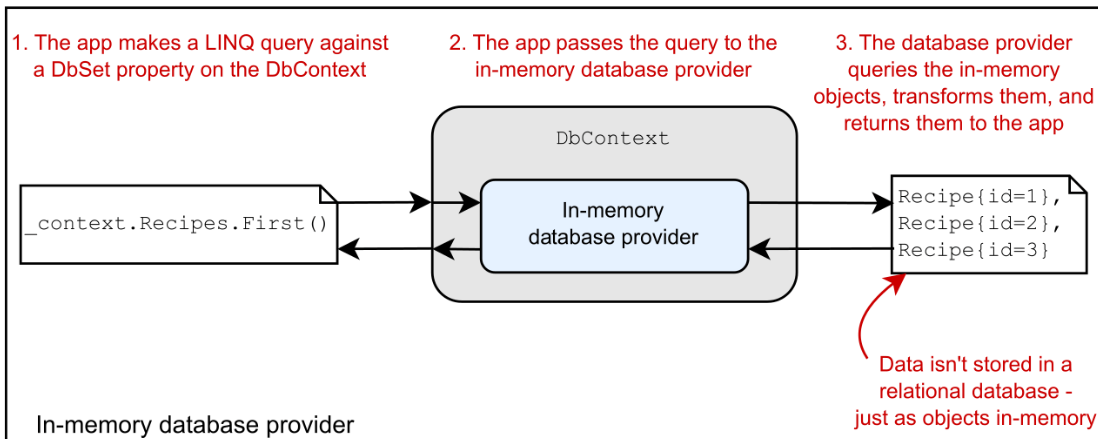
- *Microsoft.EntityFrameworkCore.InMemory*—This provider doesn't simulate a database. Instead, it stores objects directly in memory. It isn't a relational database as such, so it doesn't have all the features of a normal database. You can't execute SQL against it directly, and it won't enforce constraints, but it's fast.
- *Microsoft.EntityFrameworkCore.Sqlite*—SQLite is a relational database. It's limited in features compared to a database like SQL Server, but it's a true relational database, unlike the in-memory database provider. Normally, a SQLite database is written to a file, but the provider includes an in-memory mode, in which the database stays in memory. This makes it much faster and easier to create and use for testing.

Instead of storing data in a database on disk, both of these providers store data in memory, as shown in figure 23.9. This makes them fast, easy to create and tear down, and allows you to create a new database for every test, to ensure your tests stay isolated from one another.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>







©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Figure 23.9 The in-memory database provider and SQLite provider (in-memory mode) compared to the SQL Server database provider. The in-memory database provider doesn't simulate a database as such. Instead, it stores objects in memory and executes LINQ queries against them directly.

NOTE In this post, I describe how to use the SQLite provider as an in-memory database, as it's more fully featured than the in-memory provider. For details on using the in-memory provider see http://mng.bz/hdlq.

To use the SQLite provider in memory, add the Microsoft.EntityFrameworkCore.Sqlite package to your test project's .csproj file. This adds the `UseSqlite()` extension method, which you'll use to configure the database provider for your unit tests.

Listing 23.19 shows how you could use the in-memory SQLite provider to test the `GetRecipe()` method of `RecipeService`. Start by creating a `SqliteConnection` object and using the `"DataSource=:memory:"` connection string. This tells the provider to store the database in memory and then open the connection.

WARNING The in-memory database is destroyed when the connection is closed. If you don't open the connection yourself, EF Core will close the connection to the in-memory database when you dispose the `DbContext`. If you want to share an in-memory database between `DbContexts`, you must explicitly open the connection yourself.

Next, pass the `SqlLiteConnection` instance into the `DbContextOptionsBuilder<>` and call `UseSqlite()`. This configures the resulting `DbContextOptions<>` object with the necessary services for the SQLite provider and provides the connection to the in-memory database. By passing this options object into an instance of `AppDbContext`, all calls to the `DbContext` result in calls to the in-memory database provider.

Listing 23.19 Using the in-memory database provider to test an EF Core `DbContext`

```
[Fact]
public void GetRecipeDetails_CanLoadFromContext()
{
    var connection = new SqliteConnection("DataSource=:memory:");    #A
    connection.Open();                                                #B

    var options = new DbContextOptionsBuilder<AppDbContext>()        #C
        .UseSqlite(connection)                                        #C
        .Options;                                                     #C

    using (var context = new AppDbContext(options))                   #D
    {
        context.Database.EnsureCreated();                             #E
        context.Recipes.AddRange(                                     #F
            new Recipe { RecipeId = 1, Name = "Recipe1" },            #F
            new Recipe { RecipeId = 2, Name = "Recipe2" },            #F
            new Recipe { RecipeId = 3, Name = "Recipe3" });           #F
        context.SaveChanges();                                        #G
    }
    using (var context = new AppDbContext(options))             #H
    {
```


Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
        var service = new RecipeService(context);               #I
        var recipe = service.GetRecipe (id: 2);                 #J
        Assert.NotNull(recipe);                    #K
        Assert.Equal(2, recipe.Id);                #K
        Assert.Equal("Recipe2", recipe.Name);      #K
    }
}
```

#A Configures an in-memory SQLite connection using the special "in-memory" connection string
#B Opens the connection so EF Core won't close it automatically
#C Creates an instance of DbContextOptions<> and configures it to use the SQLite connection
#D Creates a DbContext and passes in the options
#E Ensures the in-memory database matches EF Core's model (similar to running migrations)
#F Adds some recipes to the DbContext
#G Saves the changes to the in-memory database
#H Creates a fresh DbContext to test that you can retrieve data from the DbContext
#I Creates the RecipeService to test and pass in the fresh DbContext
#J Executes the GetRecipe function. This executes the query against the in-memory database.
#K Verifies that you correctly retrieved the recipe from the in-memory database

This example follows the standard format for any time you need to test a class that depends on an EF Core `DbContext`:

1. Create a `SqliteConnection` with the `"DataSource=:memory:"` connection string and open the connection.
2. Create a `DbContextOptionsBuilder<>` and call `UseSqlite()`, passing in the open connection.
3. Retrieve the `DbContextOptions` object from the `Options` property.
4. Pass the options to an instance of your `DbContext` and ensure the database matches EF Core's model by calling `context.Database.EnsureCreated()`. This is similar to running migrations on your database but should *only* be used on test databases. Create and add any required test data to the in-memory database and call `SaveChanges()` to persist the data.
5. Create a new instance of your `DbContext` and inject it into your test class. All queries will be executed against the in-memory database.

By using two separate `DbContext`s, you can avoid bugs in your tests due to EF Core caching data without writing it to the database. With this approach, you can be sure that any data read in the second `DbContext` was persisted to the underlying in-memory database provider.

This was a very brief introduction to using the SQLite provider as an in-memory database provider, and EF Core testing in general, but if you follow the setup shown in listing 23.19, it should take you a long way. The source code for this chapter shows how you can combine this code with a custom `WebApplictaionFactory` to use an in-memory database for your integration tests. For more details on testing EF Core, including additional options and strategies, see *Entity Framework Core in Action* by Jon P Smith (Manning, 2021).

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

23.7 Summary

- Unit test apps are console apps that have a dependency on the .NET Test SDK, a test framework such as xUnit, MSTest, or NUnit, and a test runner adapter. You can run the tests in a test project by calling `dotnet test` from the command line in your test project or by using the Test Explorer in Visual Studio.
- Many testing frameworks are compatible with the .NET Test SDK, but xUnit has emerged as an almost *de facto* standard for ASP.NET Core projects. The ASP.NET Core team themselves use it to test the framework.
- To create an xUnit test project, choose xUnit Test Project (.NET Core) in Visual Studio or use the `dotnet new xunit` CLI command. This creates a test project containing the Microsoft.NET.Test.Sdk, xunit, and xunit.runner.visualstudio NuGet packages.
- xUnit includes two different attributes to identify test methods. `[Fact]` methods should be public and parameterless. `[Theory]` methods can contain parameters, so they can be used to run a similar test repeatedly with different parameters. You can provide the data for each `[Theory]` run using the `[InlineData]` attribute.
- Use assertions in your test methods to verify that the system under test (SUT) returned an expected value. Assertions exist for most common scenarios, including verifying that a method call raised an exception of a specific type. If your code raises an unhandled exception, the test will fail.
- Use the `DefaultHttpContext` class to unit test your custom middleware components. If you need access to the response body, you must replace the default `Stream.Null` with a `MemoryStream` instance and manually read the stream after invoking the middleware.
- API controllers and Razor Page models can be unit tested just like other classes, but they should generally contain little business logic, so it may not be worth the effort. For example, the API controller is tested independently of routing, model validation, and filters, so you can't easily test logic that depends on any of these aspects.
- Integration tests allow you to test multiple components of your app at once, typically within the context of the ASP.NET Core framework itself. The Microsoft.AspNetCore.TestHost package provides a `TestServer` object that you can use to create a simple web host for testing. This creates an in-memory server that you can make requests to and receive responses from. You can use the `TestServer` directly when you wish to create integration tests for custom components like middleware.
- For more extensive integration tests of a real application, you should use the `WebApplicationFactory` class in the Microsoft.AspNetCore.Mvc.Testing package. Implement `IClassFixture<WebApplicationFactory<Startup>>` on your test class and inject an instance of `WebApplicationFactory<Startup>` into the constructor. This creates an in-memory version of your whole app, using the same configuration, DI services, and middleware pipeline. You can send in-memory requests to your app to get the best idea of how your application will behave in production.
- To customize the `WebApplicationFactory`, call `WithWebHostBuilder()` and call `ConfigureTestServices()`. This method is invoked after your app's standard DI

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

configuration. This enables you to add or remove the default services for your app, for example to replace a class that contacts a third-party API with a stub implementation.

- If you find you need to customise the services for every test, you can create a custom `WebApplicationFactory` by deriving from it and overriding the `ConfigureWebHost` method. You can place all your configuration in the custom factory and implement `IClassFixture<CustomWebApplicationFactory>` in your test classes, instead of calling `WithWebHostBuilder()` in every test method.
- You can use the EF Core SQLite provider as an in-memory database to test code that depends on an EF Core database context. You configure the in-memory provider by creating a `SqliteConnection` with a `"DataSource=:memory:"` connection string. Create a `DbContextOptionsBuilder<>` object and call `Use-Sqlite()`, passing in the connection. Finally, pass `DbContextOptions<>` into an instance of your app's `DbContext` and call `context.Database.EnsureCreated()` to prepare the in-memory database for use with EF Core.
- The SQLite in-memory database is maintained as long as there's an open `SqliteConnection`. By opening the connection manually, the database can be used with multiple `DbContext`s. If you don't call `Open()` on the connection, EF Core will close the connection (and delete the in-memory database) when the `DbContext` is disposed.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Appendix A
Preparing your development environment

This appendix covers

- **Installing the .NET Core SDK**
- **Choosing an editor or IDE**

For .NET developers in a Windows-centric world, Visual Studio was pretty much a developer requirement in the past. But with .NET and ASP.NET Core going cross-platform, that's no longer the case.

All of ASP.NET Core (creating new projects, building, testing, and publishing) can be run from the command line for any supported operating system. All you need is the .NET SDK, which provides the .NET Command Line Interface (CLI). Alternatively, if you're on Windows, and not comfortable with the command line, you can still use File > New Project in Visual Studio to dive straight in. With ASP.NET Core, it's all about choice!

In a similar vein, you can now get a great editing experience outside of Visual Studio thanks to the OmniSharp project.[112] This is a set of libraries and editor plugins that provide code suggestions and autocomplete (IntelliSense) across a wide range of editors and operating systems. How you setup your environment will likely depend on which operating system you're using and what you're used to.

[112] Information about the OmniSharp project can be found at www.omnisharp.net. Source code can be found at https://github.com/omnisharp.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Remember that for .NET Core and .NET 5, the operating system you choose for development has no bearing on the final systems you can run on—whether you choose Windows, macOS, or Linux for development, you can deploy to any supported system.

In this appendix I show how to install the .NET SDK so you can build, run, and publish .NET apps. I'll also discuss some of the integrated development environment (IDE) and editor options available for you to build applications.

A.1 Installing the .NET SDK

The most important thing you need for .NET Core and .NET 5 development is the .NET SDK. In this section I describe how to install the .NET SDK and how to check which version you have installed.

To start programming with .NET, you need to install the .NET SDK (previously called the .NET Core SDK). This contains the base libraries, tooling, and the compiler you need to create .NET applications.

You can install the .NET SDK from https://dotnet.microsoft.com/download. This contains links to download the latest version of .NET for your operating system. If you're using Windows or macOS, this page contains installer download links; if you're using Linux, there are instructions for installing .NET using your distributions package manager, as a Snap package, or as a manual download.

> **WARNING** Make sure you download the .NET *SDK* not the .NET *Runtime*. The .NET runtime is used to execute .NET applications, but it can't be used to build them. The .NET SDK includes a copy of the runtime, so it can run your applications, but it can also build, test, and publish them.

After installing the .NET SDK, you can run commands with the .NET CLI using the `dotnet` command. Run `dotnet --info` to see information about the version of the .NET SDK currently in use, as well as the .NET SDKs and .NET runtimes you have installed, as shown in figure A.1.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Figure A.1. Use `dotnet --info` to check which version of the .NET SDK is currently used, and which versions are available. Use this command to check that you have installed the .NET SDK correctly. This screenshot shows I am currently using a preview version of .NET 5.

As you can see in figure A.1, I have multiple versions of the .NET SDK (previously, the .NET Core SDK) installed. This is perfectly fine, but not necessary. Newer versions of the .NET SDK can build applications that target older versions of .NET Core. For example, the .NET 5 SDK can build .NET 5, .NET Core 3.x, .NET Core 2.x and .NET Core 1.x applications. In contrast, the .NET Core 3.1 SDK *can't* build .NET 5 applications.

> **TIP** Some IDEs, such as Visual Studio, can automatically install .NET 5 as part of their installation process. There is no problem installing multiple versions of .NET Core and .NET 5 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 https://andrewlock.net/exploring-the-new-rollforward-and-allowprerelease-settings-in-global-json/.

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.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

A.2 Choosing an IDE or editor

In this section I 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 only)

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 integrated development environment, which provides one of the best all-around experiences for developing ASP.NET Core applications. Luckily, the Visual Studio Community edition is now free for open source, students, and small teams of developers.

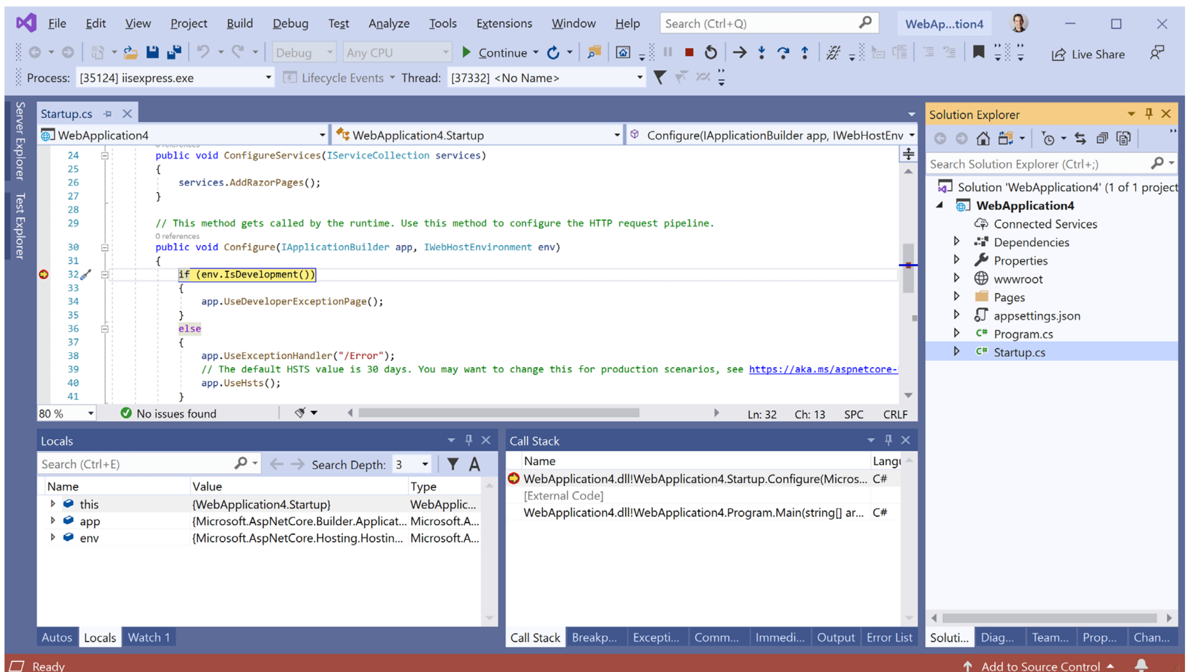


Figure A.2 Visual Studio provides one of the most complete ASP.NET Core development environments for Windows users.

Visual studio comes loaded with a whole host of templates for building new projects, best-in-class debugging, and publishing, without ever needing to touch a command prompt. It's

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

especially suited if you're publishing to Azure, as it has many direct hooks into Azure features to make development and deployment easier.

You can install Visual Studio by visiting https://visualstudio.microsoft.com/vs/ and clicking Download Visual Studio. Choose the Community Edition (unless you have a license for the Professional or Enterprise version) and follow the prompts to install Visual Studio.

The Visual Studio *installer* is an application in-and-of-itself and will ask you to select *workloads* to install. You can select as many as you like, but for ASP.NET Core development, ensure you select at a minimum:

- ASP.NET and web development
- .NET Core cross-platform development

After selecting these workloads, click Download, and fetch a beverage of your choice. Despite having been on a diet recently, Visual Studio still requires many GB to be downloaded and installed! Once it's finished, you'll be ready to start building ASP.NET Core applications.

A.2.2 JetBrains Rider (Windows, Linux, macOS)

Rider (figure A.3), from the company JetBrains, is a cross-platform IDE alternative to Visual Studio. Released in 2017, Rider is another full-featured IDE, based on the venerable ReSharper plugin. If you're used to using Visual Studio with the ReSharper plugin, and the multitude of refactorings this plugin provides, then I strongly suggest investigating Rider. Similarly, if you're familiar with JetBrains' IntelliJ products, you will feel at home in Rider.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

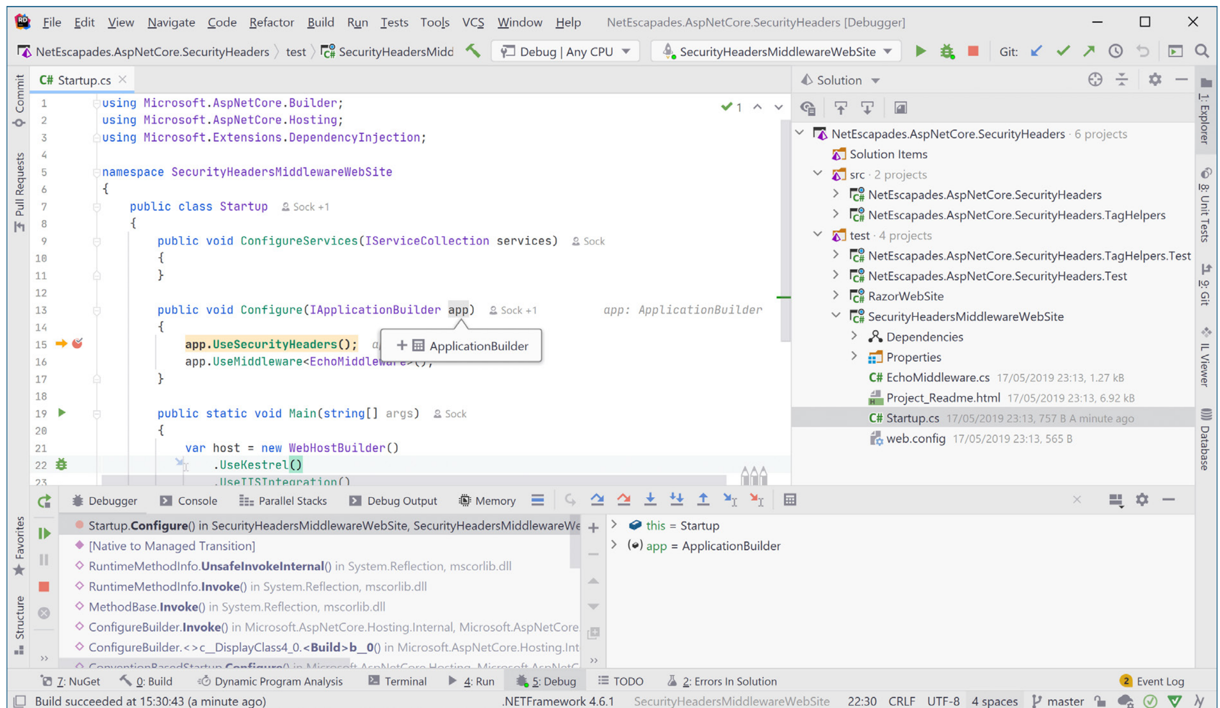


Figure A.3 Rider is a cross-platform .NET IDE from JetBrains. It is based on the ReSharper plugin for Visual Studio, so includes many of the same refactoring features, as well as a debugger, test runner, and all the other integration features you would expect from a full-featured IDE.

To install Rider visit https://www.jetbrains.com/rider/ and click Download. Rider comes with a 30-day free trial, after which you will need to purchase a license. If you already have a ReSharper license, you may already have a license for Rider. They also offer discounts or free licenses for various users, such as students and startups, so it's worth looking into.

A.2.3 Visual Studio for Mac (macOS)

Despite the branding, Visual Studio for Mac is a completely different product to Visual Studio. Rebranded and extended from its Xamarin Studio precursor, you can now use Visual Studio for Mac to build ASP.NET Core applications on macOS. Visual Studio for Mac generally has fewer features than Visual Studio or Rider, but it offers a native IDE, and is under active development.

To install Visual Studio for Mac, visit https://visualstudio.microsoft.com/vs/mac/, click Download Visual Studio for Mac, and download and run the installer.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

A.2.4 Visual Studio Code (Windows, Linux, macOS)

Sometimes, you don't want a full-fledged IDE. Maybe you want to quickly view or edit a file, or you don't like the sometimes-unpredictable performance of Visual Studio. In those cases, a simple editor may be all you want or need, and Visual Studio Code is a great choice. Visual Studio Code (figure A.4) is an open source, lightweight editor that provides editing, IntelliSense, and debugging for a wide range of languages, including C# and ASP.NET Core.

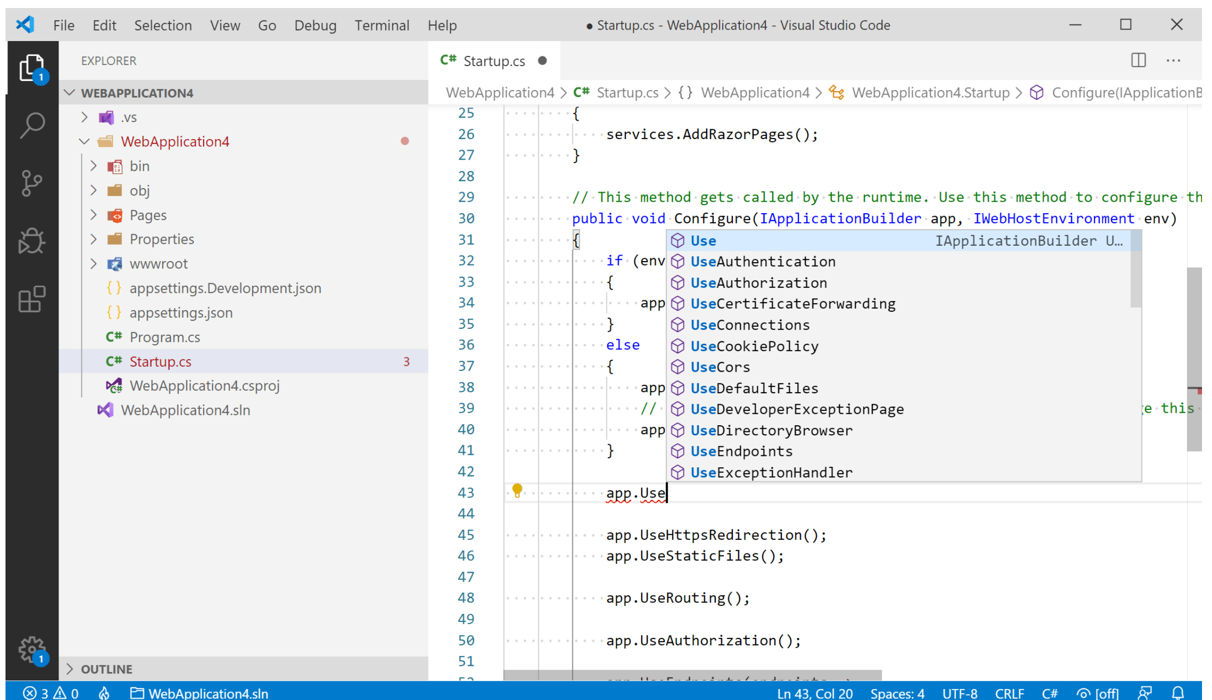


Figure A.4 Visual Studio Code provides cross-platform IntelliSense and debugging.

To install Visual Studio Code, visit https://code.visualstudio.com/, click Download, and run the downloaded installer. The first time you open a folder containing a C# project or solution file with Visual Studio Code, you'll be prompted to install a C# extension. This provides the IntelliSense and integration between Visual Studio Code and the .NET SDK.

The extension model of VS Code is one of its biggest assets, as you can add a huge amount of additional functionality. Whether you're working with Azure, AWS, or any other technology, be sure to check the extension marketplace at https://marketplace.visualstudio.com/vscode to see what's available. If you search for ".NET Core", you'll also find a huge array of extensions that can bring VS Code closer to that full-blown IDE experience if you wish.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

In this book, I use Visual Studio for most of the examples, but you'll be able to follow along using any of the tools I've mentioned. The book assumes you've successfully installed .NET 5 and an editor on your computer.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Appendix B

Understanding the .NET ecosystem

This appendix covers

- **The history of .NET leading to the development of .NET Core**
- **The position of .NET 5 in the .NET ecosystem**
- **Sharing code between projects with .NET Standard**
- **The future of .NET Standard with .NET 5**

The .NET ecosystem has changed a lot since .NET was first introduced, but the development of .NET Core and .NET 5 has resulted in a particularly large degree of churn and the introduction of many new concepts.

This churn isn't surprising given Microsoft's newfound transparency regarding the development process and building in the open on GitHub. Unfortunately, it can be confusing for developers new to .NET, and even to seasoned veterans! In this appendix, I try to straighten out some of the terms that developers new to .NET often find confusing, as well as provide some context for the changes.

In this appendix I discuss the history the .NET ecosystem, how it has evolved, and the issues Microsoft was attempting to solve. As part of this, I'll discuss the similarities and differences between .NET 5, .NET Core, and .NET Framework.

.NET Core wasn't developed in isolation, and one of its primary design goals was to improve the ability to share code between multiple frameworks. In section B.2, I describe how this was achieved in pre-.NET Core/.NET 5 days, using Portable Class Libraries (PCLs) and the successor approach using .NET Standard. Finally, in section B.3, I discuss what .NET 5 means for .NET Standard.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

B.1 The evolution of .NET into .NET 5

In this section I discuss the history or .NET 5 and .NET Core and why they were created. You'll learn about the various .NET platforms around in the early 2010s and why their sharding prompted the development of .NET Core as a new cross-platform runtime. Finally, you'll learn how .NET 5 has grown out of .NET Core, and the future of .NET.

B.1.1 Exploring the .NET platforms that prompted .NET Core

If you're a .NET developer, chances are you're already familiar with the .NET Framework. The .NET Framework, version 4.8 at the time of writing, is a Windows-only development platform that you can use for both desktop and web development. It's installed by default on Windows and was historically used for most desktop and server .NET development.

If you're a mobile developer, you might also be familiar with the Xamarin framework, which, until recently, used the cross-platform Mono implementation of the .NET Framework. This is an alternative *platform* to the .NET Framework that you can use to build mobile applications on Windows, Android, and iOS.

Historically, these two platforms were completely separate, but they consisted of many similar components and implemented similar APIs. Each platform contained libraries and app-models specific to their platform, but they use similar fundamental libraries and types, as shown in figure B.1.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

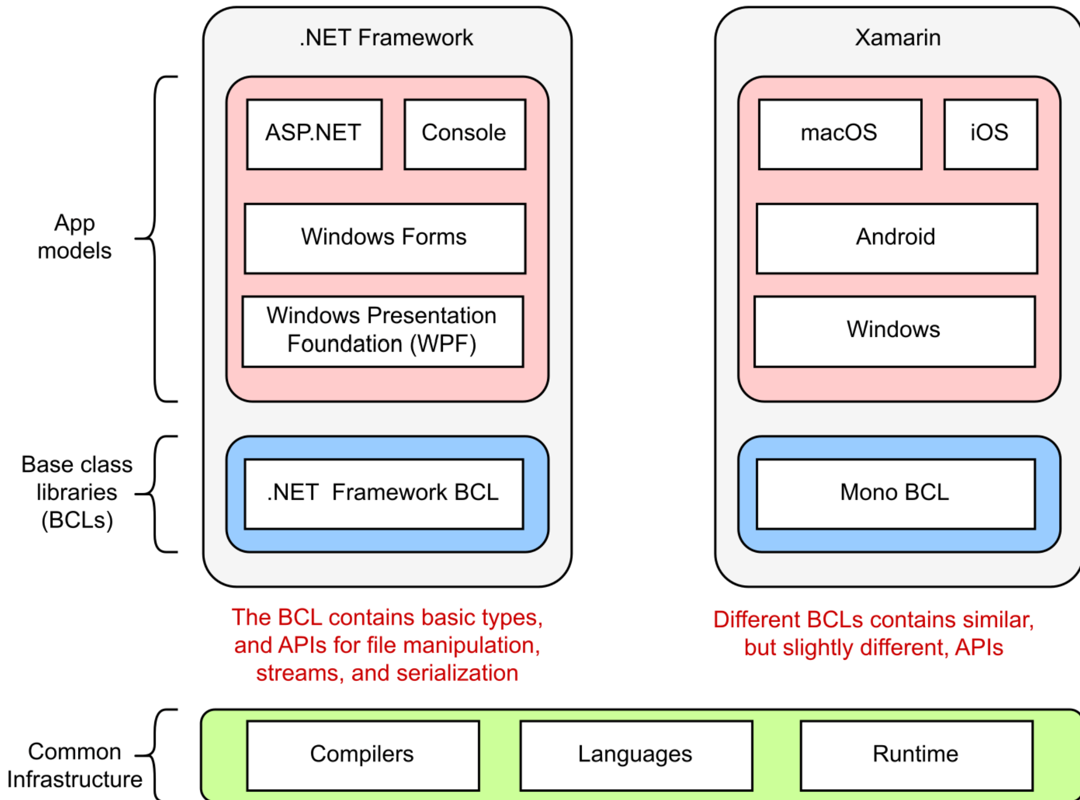


Figure B.1 The layers that make up the .NET Framework. Each builds on the capabilities of the layer below, with the highest layer providing the app models that you'll use to build your applications.

At the bottom of each stack is the tooling that allows you to compile and run .NET applications such as the compiler and the common language runtime (CLR). At the top of each stack, you have the app-specific libraries that you use to build applications for your platform. For example, you could build a Windows Forms app on the .NET Framework, but not using the Xamarin platform, and vice-versa for an iOS app.

In the middle of each stack you have the Base Class Libraries (BCL). These are the fundamental .NET types you use in your apps on a daily basis: the `int` and `string` types, the file-reading APIs, the `Task` APIs, and so on. Although both .NET platforms have many similar

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

types, they're fundamentally different, so you can't guarantee a type will exist on both platforms, or that it will expose the same methods and properties.

I've only discussed two platforms so far, the .NET Framework and Xamarin, but .NET has *many* different implementations, of which these are only two. Windows also has the Windows 8/8.1 platform and the Universal Windows Platform (UWP). On phones, in addition to Xamarin, there's the Windows Phone 8.1 and Silverlight Phone platforms. The list goes on and on (Unity, .NET Compact Framework (CF), .NET Micro …)!

Each of these platforms uses a slightly different set of APIs (classes and methods) in their BCL. Platforms have a certain number of similar APIs between them in their BCLs, but the intersection is patchy. On top of that, the libraries that make up the BCL of a platform are fundamentally not interoperable. Any source code written for a given set of APIs must be specifically recompiled for each target platform.

Several years ago, Microsoft realized this sharding of .NET was a problem. Developers had to know a *slightly* different set of APIs for each platform, and sharing code so that it could be used on more than one platform was a pain.

On top of that, the primary web development platform of the .NET Framework was showing its age. The software industry was moving toward small, lightweight, cloud-native frameworks that you could deploy in cross-platform environments. The centrally installed Windows-only .NET Framework was not designed for these scenarios. Microsoft set about developing a new framework, called "Project K" during development, which ultimately became .NET Core.

B.1.2 Introducing .NET Core

The .NET Core platform was Microsoft's solution to the centrally installed, Windows-only .NET Framework. .NET Core is highly modular, can be deployed side-by-side with other .NET Core installations (or alternatively, distributed with the app), and is cross-platform. The term .NET Core is somewhat overloaded, in that it was used through development as a general umbrella term to describe a variety of changes. The .NET Core platform consists of

- *A cross-platform BCL*—The BCL libraries of the .NET Core platform, historically called CoreFX, contain all the primitive types and libraries for building .NET Core applications.
- *A new cross-platform runtime*—The runtime for .NET Core, called CoreCLR, which executes .NET Core applications.
- *The .NET CLI tooling*—The `dotnet` tool used for building and publishing apps.
- *The ASP.NET Core libraries*—The "app-layer" libraries, used to build ASP.NET Core applications.

These components make up the .NET Core platform and find their analogs to the various components that make up the .NET Framework and Xamarin platforms you saw in figure B.1. By creating a new platform, Microsoft was able to maintain backward compatibility for apps that used the .NET Framework, which allowed new apps to be developed using .NET Core and take advantage of its cross-platform and isolated deployment story.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

NOTE You might be thinking, "Wait, they had too many .NET platforms, so they created *another* one?" If so, you're on the ball. But luckily, with .NET Core, came .NET Standard.

On its own, .NET Core would've meant *yet another* BCL of APIs for .NET developers to learn. But as part of the development of .NET Core, Microsoft introduced .NET Standard. .NET Standard, as the name suggests, ensures a *standard set* of APIs is available on every .NET platform. You no longer had to learn the specific set of APIs available for the flavor of .NET you were using; if you could use the .NET Standard APIs, you knew you'd be fine on multiple platforms. I'll talk more about .NET Standard in section B.2.

.NET Standard was a good stop-gap solution for writing code that could work on multiple platforms, but it didn't address one fundamental issue: there were still multiple platforms. Every platform had its *own separate code* that must be maintained by Microsoft, despite being *almost* identical in many places.

Microsoft was innovating quickly in .NET Core, introducing new C# features such as Async Enumerables and `Span<T>`, as well as providing many performance improvements.[113] Unfortunately, none of the other platforms could take advantage of these without significant work. Microsoft's vision for tackling this head-on was "One .NET".

B.1.3 .NET 5: the first step in the One .NET vision

In May 2019, Microsoft announced[114] that the next major version of .NET Core after 3.0 would be .NET 5. This was the first step in their attempt to unify the .NET platform.

Previously, as I discussed in section B.1.1, you had to use .NET Framework to build Windows Desktop apps, Xamarin to build iOS or Android apps, and .NET Core to build cross-platform web apps. Each app-model was tied to the underlying platform and used a distinct BCL. The "One .NET" vision which started with.NET 5 is to have a *single* .NET platform, with a *single* BCL, which can be used with every app model: Windows Desktop apps, iOS or Android apps, as well as cross-platform web apps, as shown in figure B.2.

---

[113] There is a blog post on the .NET blog detailing the vast low-level improvements made to .NET Core. These are fascinating if you are into that sort of thing! You can find the .NET 5 blog post here: https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-5/.

[114] You can find the announcement blog post here: https://devblogs.microsoft.com/dotnet/introducing-net-5/. This contains a lot of detail of future plans, so I strongly suggest reading it.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

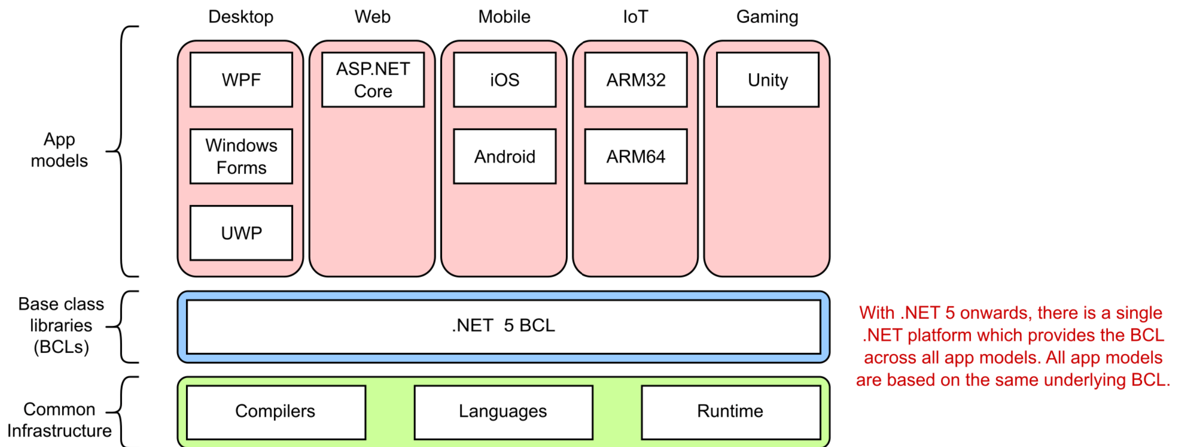


Figure B.2. .NET 5 provides a single platform for running multiple app models. Instead of each app model requiring a separate .NET platform, with a separate BCL, all app models will be able to use the same underlying .NET 5 platform and BCL.

Practically speaking .NET 5 really is "just" the next version of .NET Core. There are very few breaking changes moving an ASP.NET Core 3.1 application to .NET 5[115], and for the most part, the upgrade is very easy. .NET 5 adds additional features (such as gRPC and Blazor), but fundamentally not much has changed for most ASP.NET Core applications.

> **NOTE** **A common point of confusion is the name: .NET 5. The "Core" moniker was dropped to try and signify "there is only one version of .NET now". Also, version 4 was skipped, to avoid confusion between the new version and .NET Framework version 4. Hopefully this naming decision will pay off in the long run, even if it's confusing now!**

.NET 5 represents the first step on the road to One .NET. The hope is that basing all future development effort on one platform will reduce duplication of effort and provide both greater stability and progress for the platform. With that in mind, Microsoft have committed to a regular release cadence, so you can easily plan how to keep your apps up to date as new versions of .NET are released.

---

[115] You can see the list of breaking changes at https://docs.microsoft.com/en-us/dotnet/core/compatibility/3.1-5.0.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

B.1.4 The future: .NET 6 and beyond

As with many open source projects, developing in the open is often associated with a faster release cycle than with the traditional .NET Framework. This was certainly the case with .NET Core, with new releases (major and minor) coming regularly for the first few years of development.

While many developers like this faster cadence and the new features it brings, it can lead to some uncertainty. Is it worth spending time upgrading to the latest version now if a new version is going to be released next week?

To counteract the potential churn, and give users confidence in continued support, each .NET Core (and subsequently, .NET) release falls into one of two *support tracks*:

- *Long Term Support (LTS).* These releases are supported for three years from their first release.
- *Current*. These releases are supported until three months after the *next* LTS or current release.

Having two supported tracks leaves you with a simple choice: if you want more features, and are happy to commit to updating your app more frequently, choose Current releases; if you want fewer updates but also fewer features, choose LTS[116].

The two-track approach went some way to alleviating uncertainty, but it still left users unsure *exactly* when a new release would occur, and hence how long the current version would be supported.

With .NET 5, Microsoft committed to a well-defined release cycle consisting of shipping a new major version of .NET every year, alternating between LTS releases and Current releases, as shown in figure B.3. Minor updates are not intended to be common but will occur in interstitial months if required.

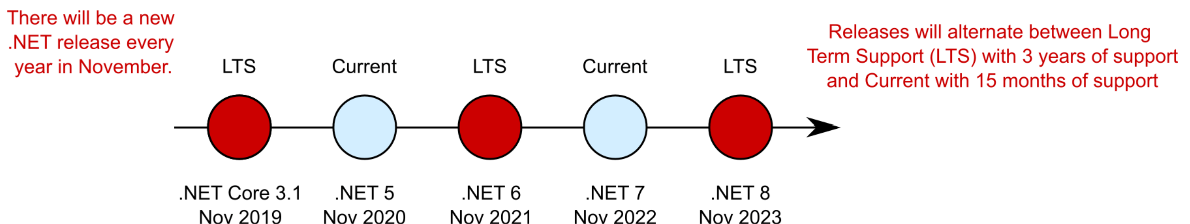


Figure B.3 The timeline for releases of new .NET versions. A new .NET version will be released every year in November. Releases will alternate between Long Term Support (LTS) versions and Current release versions.

---

[116] For more details on .NET support policies, see https://dotnet.microsoft.com/platform/support/policy/dotnet-core.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

With this timeline, you know how long a version of .NET will be supported. If you use a Current track release (such as .NET 5) you know you will be supported until 3 months after the release of .NET 6 in November 2021. As an LTS release, .NET 6 will be supported until November 2024.

The unification of multiple .NET platforms in .NET 5 means that there will be less need in future to share code between multiple platforms: that's one of the big selling points of One .NET. Nevertheless, you will no doubt need to share code with existing legacy applications for many years, so code sharing is still a concern.

As I described in section B.1.2 .NET Standard was introduced with .NET Core as a way of sharing code between .NET Core applications and existing legacy applications. Before I dig into the details of .NET Standard, I'll briefly discuss its predecessor, Portable Class Libraries, and why they're now obsolete thanks to .NET Standard.

B.2 Sharing code between projects

In this section I discuss the history of sharing code between .NET platforms using Portable Class Libraries. I then introduce .NET Standard as an alternative solution that was introduced with .NET Core.

With so many different .NET implementations, the .NET ecosystem needed a way to share code between libraries, long before .NET Core was envisaged. What if you wanted to use the same classes in both your ASP.NET .NET Framework project and your Silverlight project? You'd have to create a separate project for each platform, copy and paste files between them, and recompile your code for each platform. The result was two different libraries from the same code. Portable Class Libraries (PCLs) were the initial solution to this problem.

B.2.1 Finding a common intersection with Portable Class Libraries

PCLs were introduced to make the process of compiling and sharing code between multiple platforms simpler. When creating a library, developers could specify the platforms they wanted to support, and the project would only have access to the set of APIs common among all of them. Each additional platform supported would reduce the API surface to only those APIs available in *all* the selected platforms, as shown in figure B.4.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

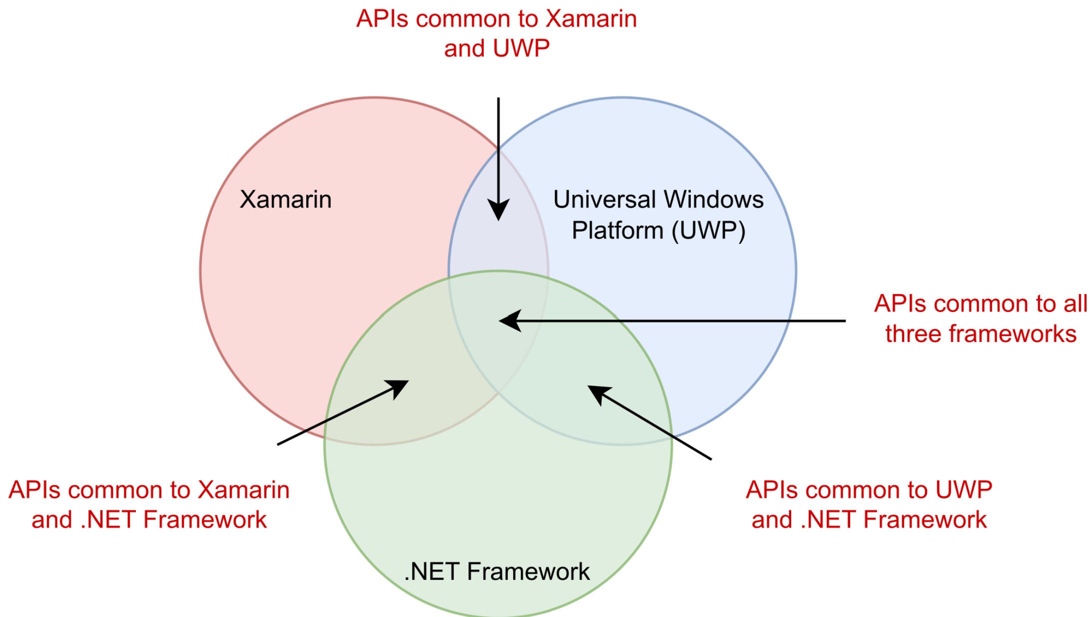


Figure B.4 Each additional framework that must be supported by a PCL reduces the APIs available to your application. If you support multiple frameworks you have vastly fewer APIs available to you.

To create a PCL library, you'd create a library that targeted a specific PCL "profile". This profile contained a precomputed list of APIs known to be available on the associated platforms. That way, you could create one library that you could share across your selected platforms. You could have a single project and a single resulting package—no copy and paste or duplicate projects required.

This approach was a definite improvement over the previous option, but creating PCLs was often tricky. There were inherent tooling complexities to contend with and understanding the APIs available for each different combination of platforms that made up a PCL profile was difficult.[117]

On top of these issues, every additional platform you targeted would reduce the BCL API surface available for you to use in your library. For example, the .NET Framework might contain APIs A, B, and C. But if Xamarin only has API A and Windows Phone only has API C, then your library can't use any of them, as shown in figure B.5.

[117] See here for the full horrifying list: https://portablelibraryprofiles.stephencleary.com/.

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

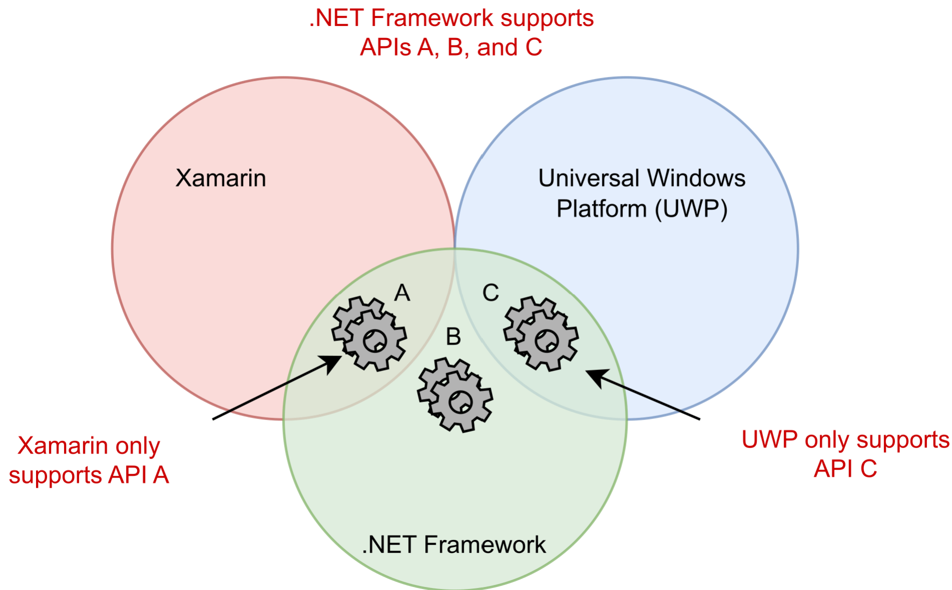


Figure B.5 Each platform exposes slightly different APIs. When creating PCLs, only those APIs that are available in all the targeted platforms are available. In this case, none of the APIs, A, B, or C, is available in all targeted platforms, so none of them can be used in the PCL.

One additional issue with PCL libraries was that they were inherently tied to the underlying platforms they targeted. In order to work with a new target platform, you'd have to recompile the PCL, even if no source code changes were required.

Say you're using a PCL library that supports Windows Phone 8.1 and .NET Framework 4.5. If Microsoft were to release a new platform, let's say, .NET Fridge, which exposes the same API as Windows Phone 8.1, you wouldn't be able to use the existing library with your new .NET Fridge application. Instead, you'd have to wait for the library author to recompile their PCL to support the new platform, and who knows when that would be!

PCLs had their day, and they solved a definite problem, but for modern development .NET Standard provides a much cleaner approach.

B.2.2 .NET Standard: a common interface for .NET

As part of the development of .NET Core, Microsoft announced .NET Standard as the successor to PCL libraries. .NET Standard takes the PCL relationship between platform support and APIs available, and flips it on its head:

- *PCLs*—A PCL profile targets a specific set of *platforms*. The APIs available to a PCL library are the common APIs shared by all the platforms in the profile.
- *.NET Standard*—A .NET Standard version defines a specific set of *APIs*. These APIs are

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

always available in a .NET Standard library. Any platform that implements all these APIs supports that version of .NET Standard.

.NET Standard isn't something you download. Instead, it's a list of APIs that a .NET Standard-compatible platform must implement.[118] You can create libraries that target .NET Standard, and you can use that library in any app that targets a .NET Standard-compatible platform.

.NET Standard has multiple *versions*, each of which is a superset of the previous versions. For example, .NET Standard 1.2 includes all the APIs from .NET Standard 1.1, which in turn includes all the APIs from .NET Standard 1.0, as shown in figure B.6.

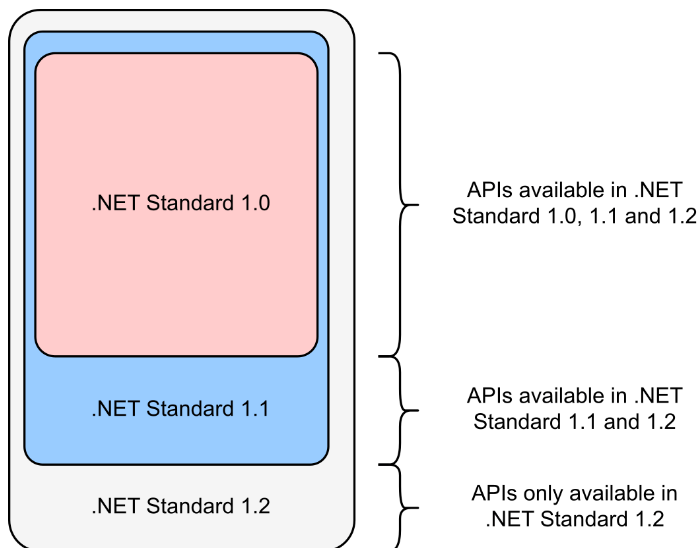


Figure B.6 Each version of .NET Standard includes all the APIs from previous versions. The smaller the version of .NET Standard, the smaller the number of APIs.

When you create a .NET Standard library, you target a specific version of .NET Standard and can reference any library that targets that version or earlier. If you're writing a library that targets .NET Standard 1.2, you can reference packages that target .NET Standard 1.2, 1.1, or 1.0. Your package can in turn be referenced by any library that targets .NET Standard 1.2 or later or any library that targets a *platform that implements* .NET Standard 1.2 or later.

[118] It is, literally, a list of APIs. You can view the APIs included in each version of .NET Standard on GitHub here: https://github.com/dotnet/standard/tree/master/docs/versions. For example you can see the APIs include in .NET Standard 1.0 here: https://github.com/dotnet/standard/blob/master/docs/versions/netstandard1.0.md.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

A platform implements a specific version of .NET Standard if it contains all the APIs required by that version of .NET Standard. By extension, a platform that supports a specific version of .NET Standard also supports all previous versions of .NET Standard. For example, UWP version 10 supports .NET Standard 1.4, which means it also supports .NET Standard versions 1.0-1.3, as shown in figure B.7.

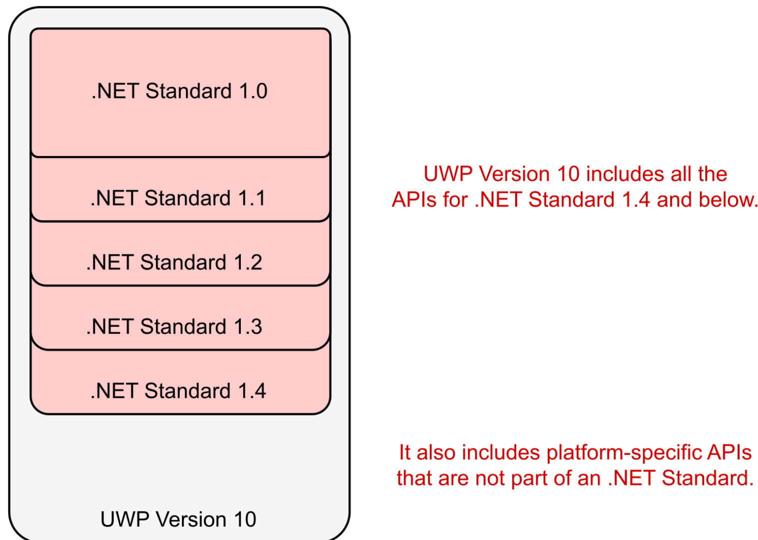


Figure B.7 The UWP Platform version 10 supports .NET Standard 1.4. That means it contains all the APIs required by the .NET Standard specification version 1.4. That means it also contains all the APIs in earlier versions of .NET Standard. It also contains additional platform-specific APIs that aren't part of any version of .NET Standard.

Each version of a platform supports a different version of .NET Standard. .NET Framework 4.5 supports .NET Standard 1.1, but .NET Framework 4.7.1 supports .NET Standard 2.0. Table B.1 shows some of the versions supported by various .NET platforms. For a more complete list, see https://docs.microsoft.com/dotnet/standard/net-standard.

Table B.1 Highest supported .NET Standard version for various .NET platform versions. A blank cell means that version of .NET Standard isn't supported on the platform.

.NET Standard Version	1.0	1.1	1.2	1.3	1.4	1.5	1.6	2.0	2.1
.NET Core	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0	3.0
.NET Framework	4.5	4.5	4.5.1	4.6	4.6.1	4.6.2		4.7.1	

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Mono	4.6	4.6	4.6	4.6	4.6	4.6	4.6	5.4	6.4
Windows	8.0	8.0	8.1						
Windows Phone	8.1	8.1	8.1						

A version of this table is often used to explain .NET Standard, but for me, the relationship between .NET Standard and a .NET Platform all made sense when I saw an example that explained .NET Standard in terms of C# constructs.[119]

You can think of each version of .NET Standard as a series of inherited interfaces, and the .NET platforms as implementations of one of these interfaces. In the following listing, I use the last two rows of table B.1 to illustrate this, considering .NET Standard 1.0-1.2 and looking at the Windows 8.0 platform and Windows Phone 8.1.

Listing B.1 An interpretation of .NET Standard in C#

```
interface NETStandard1_0        #A
{
   void SomeMethod();
}

interface NETStandard1_1 : NETStandard1_0       #B
{
    void OtherMethod();          #C
}

interface NETStandard1_2 : NETStandard1_1         #D
{
    void YetAnotherMethod(); #E
}

class Windows8 : NETStandard1_1                      #F
{
   void SomeMethod () { /* Method implementation */ }     #G
   void OtherMethod() { /* Method implementation */ }     #G

   void ADifferentMethod() { /* Method implementation */      #H
}

class WindowsPhone81 : NETStandard1_2          #I
{
   void SomeMethod () { /* Method implementation */ }         #J
   void OtherMethod() { /* Method implementation */ }         #J
   void YetAnotherMethod () { /* Method implementation */ }   #J

   void ExtraMethod1() { /* Method implementation */ }       #K
```

[119] The example was originally provided by David Fowler from the ASP.NET team. You can view an updated version of this metaphor here: https://github.com/dotnet/standard/blob/master/docs/metaphor.md/.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

```
   void ExtraMethod2() { /* Method implementation */ }       #K
}
```

#A Defines the APIs available in .NET Standard 1.0
#B .NET Standard 1.1 inherits all the APIs from .NET Standard 1.0.
#C APIs available in .NET Standard 1.1 but not in 1.0
#D .NET Standard 1.2 inherits all the APIs from .NET Standard 1.1.
#E APIs available in .NET Standard 1.2 but not in 1.1 or 1.0
#F Windows 8.0 implements .NET Standard 1.1.
#G Implementations of the APIs required by .NET Standard 1.1 and 1.0
#H Additional APIs that aren't part of .NET Standard, but exist on the Windows 8.0 platform
#I Windows Phone 8.1 implements .NET Standard 1.2.
#J Implementations of the APIs required by .NET Standard 1.2, 1.1, and 1.0
#K Additional APIs that aren't part of .NET Standard, but exist on the Windows Phone 8.1 platform

In the same way that you write programs to use interfaces rather than specific implementations, you can target your *libraries* against a .NET Standard interface without worrying about the individual implementation details of the platform. You can then use your library with any platform that implements the required interface version.

One of the advantages you gain by targeting .NET Standard is the ability to target new platforms without having to recompile any of your libraries or wait for dependent library authors to recompile theirs. It also makes reasoning about the exact APIs available far simpler—the higher the .NET Standard version you target, the more APIs will be available to you.

> **WARNING** Even if a platform implements a given version of .NET Standard, the method implementation might throw a `PlatformNotSupportedException`. For example, some reflection APIs might not be available on all platforms. .NET 5 includes Roslyn Analyzer support to detect this situation and will warn you of the issue at build time. See https://devblogs.microsoft.com/dotnet/automatically-find-latent-bugs-in-your-code-with-net-5/ for more details about analyzers introduced in .NET 5.

Unfortunately, things are never as simple as you want them to be. Although .NET Standard 2.0 is a strict superset of .NET Standard 1.6, apps targeting .NET Framework 4.6.1 can reference .NET Standard 2.0 libraries, *even though it technically only implements .NET Standard 1.4, as shown in Table B.1*.[120]

> **WARNING** Even though .NET Framework 4.6.1 technically only implements .NET Standard 1.4, it can reference .NET Standard 2.0 libraries. This is a special case and applies only to versions 4.6.1-4.7.0. .NET Framework 4.7.1 implements .NET Standard 2.0, so it can reference .NET Standard 2.0 libraries natively.

[120] The reasoning behind this move was laid out in a post on the .NET blog which I highly recommend reading: https://devblogs.microsoft.com/dotnet/introducing-net-standard/.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

The reasoning behind this move was to counteract a chicken-and-egg problem. One of the early complaints about .NET Core 1.x was how few APIs were available, which made porting projects to .NET Core tricky. Consequently, in .NET Core 2.0, Microsoft added thousands of APIs that were available in .NET Framework 4.6.1, the most widely installed .NET Framework version, and added these APIs to .NET Standard 2.0. The intention was for .NET Standard 2.0 to provide the same APIs as .NET Framework 4.6.1.

Unfortunately, .NET Framework 4.6.1 *doesn't* contain the APIs in .NET Standard 1.5 or 1.6. Given .NET Standard 2.0 is a strict superset of .NET Standard 1.6, .NET Framework 4.6.1 can't support .NET Standard 2.0 directly.

This left Microsoft with a problem. If the most popular version of the .NET Framework didn't support .NET Standard 2.0, no one would write .NET Standard 2.0 libraries, which would hamstring .NET Core 2.0 as well. Consequently, Microsoft took the decision to *allow .NET Framework 4.6.1 to reference .NET Standard 2.0 libraries*, as shown in figure B.8.

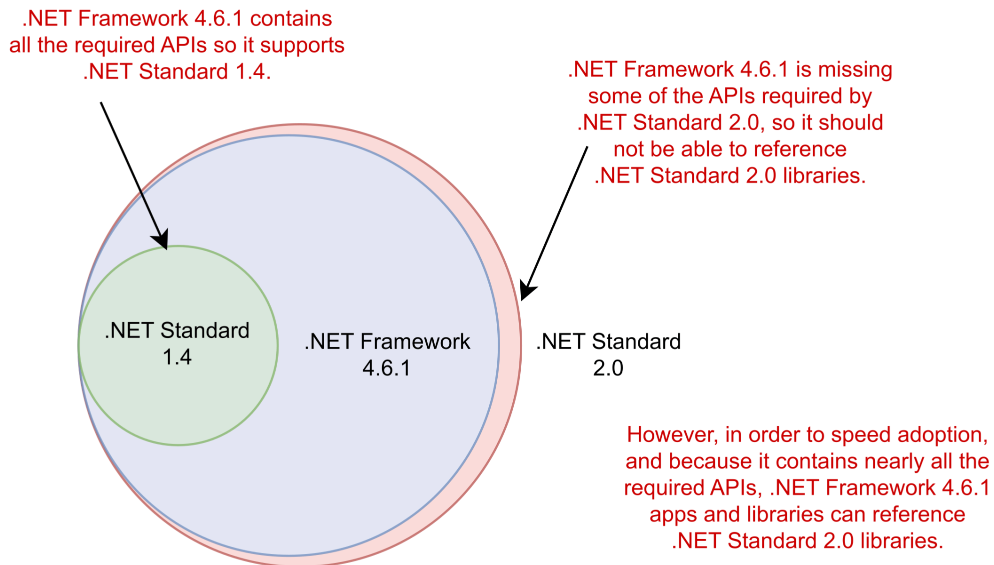


Figure B.8 NET Framework 4.6.1 doesn't contain the APIs required for .NET Standard 1.5, 1.6, or 2.0. But it contains *nearly* all the APIs required for .NET Standard 2.0. In order to speed up adoption and to make it easier to start using .NET Standard 2.0 libraries, you can reference .NET Standard 2.0 libraries for a .NET Framework 4.6.1 app.

All this leads to some fundamental technical difficulties. .NET Framework 4.6.1 can reference .NET Standard 2.0 libraries, even though it *technically* doesn't support them, but you must have the .NET Core 2.0 SDK installed to ensure everything works correctly.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

To blur things even further, libraries compiled against .NET Framework 4.6.1 can be referenced by .NET Standard libraries through the use of a compatibility shim, as I describe in the next section.

B.2.3 Fudging .NET Standard 2.0 support with the compatibility shim

Microsoft's plan for .NET Standard 2.0 was to make it easier to build .NET Core apps. If users built libraries targeting .NET Standard 2.0, then they could still use them in their .NET Framework 4.6.1 apps, but they could also use the libraries in their .NET Core apps.

The problem was that when .NET Standard 2.0 was first released, no libraries (NuGet packages) would implement it yet. Given .NET Standard libraries can only reference other .NET Standard libraries, you'd have to wait for all your dependencies to update to .NET Standard, who would have to wait for *their* dependencies first, and so on.

To speed things up, Microsoft created a compatibility shim. This shim allows a .NET Standard 2.0 library to *reference .NET Framework 4.6.1 libraries*. Ordinarily, this sort of reference wouldn't be possible; .NET Standard libraries can only reference .NET Standard libraries of an equal or lower version, as shown in figure B.9.[121]

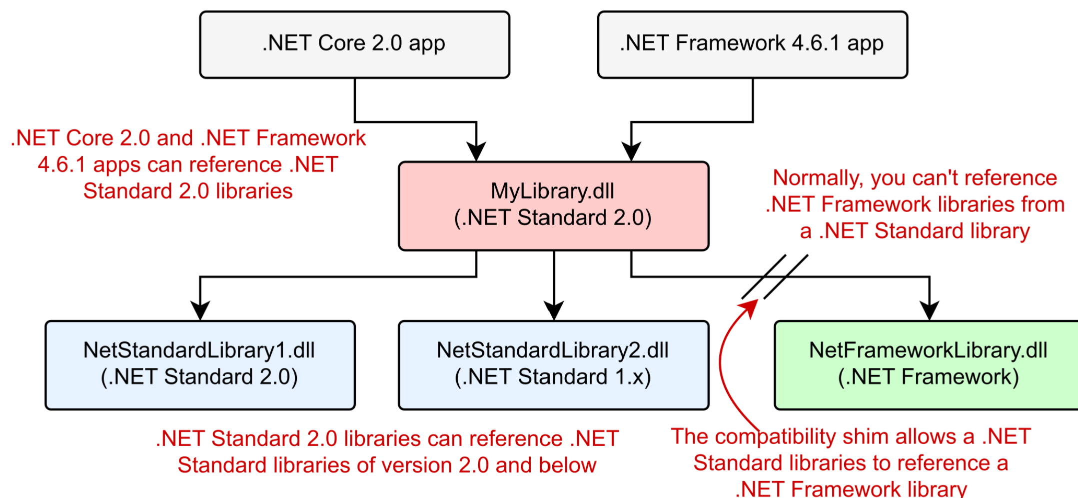


Figure B.9 By default, .NET Standard libraries can only reference other .NET Standard libraries, targeting the same .NET Standard version or lower. With the compatibility shim, .NET Standard libraries can also reference libraries compiled against .NET Framework 4.6.1.

[121] The process by which this magic is achieved is complicated. This article describes the process of assembly unification in detail: https://github.com/dotnet/standard/blob/master/docs/planning/netstandard-2.0/README.md

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

By enabling this shim, suddenly .NET Core 2.0 apps could use any of the many .NET Framework 4.6.1 (or lower) NuGet libraries available. As long as the referenced library stuck to APIs which are part of .NET Standard 2.0, you'd be able to reference .NET Framework libraries in your .NET Core 2+ apps or .NET Standard 2.0 libraries, even if your app runs cross-platform on Linux or macOS.

> **WARNING** If the library uses .NET Framework-specific APIs, you'll get an exception at runtime. There's no easy way of knowing whether a library is safe to use, short of examining the source code, so the .NET tooling will raise a warning every time you build. Be sure to thoroughly test your app if you rely on this shim.

If your head is spinning at this point, I don't blame you. This was a particularly confusing point in the evolution of .NET Standard, in which rules were being bent to fit the current environment. This inevitably led to various caveats and hand-waving, followed by bugs and fixes[122]! Luckily, if your development is focused on .NET 5, .NET Standard is not something you will generally have to worry about.

B.3 .NET 5 and the future of .NET Standard

In this section I discuss what .NET 5 means for the future of .NET Standard and the approach you should take for new applications targeting .NET 5.

> **NOTE** The advice in this section is based on official guidance from Microsoft regarding the future of .NET Standard here https://devblogs.microsoft.com/dotnet/the-future-of-net-standard/.

.NET Standard was necessary when .NET Core was a young framework, to ensure you still had access to the existing NuGet package ecosystem. .NET 5 is an evolution of .NET Core, so you can take advantage of that same ecosystem in .NET 5.

.NET 5 implements .NET Standard 2.1, the latest version of the standard, which is also implemented by .NET Core 3.0. That means .NET 5 applications can reference:

- Any NuGet package or library that implements .NET Standard 1.0-2.1.
- Any.NuGet package or library that implements .NET Core 1.x-3.x

.NET Standard was designed to handle code-sharing between multiple .NET platforms. But the release of .NET 5, and the "One .NET" vision, specifically aims to have only a *single* platform. Is .NET Standard still useful?

Yes and no. From .NET 5 onwards, no more versions of .NET Standard are planned, as subsequent versions of .NET (for example. .NET 7) will already be able to reference libraries targeting earlier versions of .NET (such as .NET 5 and .NET 6).

---

[122] You can find an example of one such issue here, but there were, unfortunately, many similar cases: https://github.com/dotnet/runtime/issues/29314.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

.NET Standard will remain useful when you need to share code between .NET 5+ applications and legacy (.NET Core, .NET Framework, Xamarin) applications. .NET Standard remains the mechanism for this cross-.NET platform code sharing.

B.4 Summary

- .NET has many different implementations, including the .NET Framework, Mono, and Unity. Each of these is a separate platform with separate Base Class Libraries (BCLs) and app models. .NET Core is another separate platform.
- Each platform has a BCL that provides fundamental .NET types and classes such as strings, file manipulation, and streams. Each platform has a slightly different BCL.
- .NET 5 is the first step in unifying these platforms, most notable Mono (and hence Xamarin) and .NET Core, under the One .NET vision. App models currently associated with other platforms will be made available on the new .NET 5+ platform.
- .NET will see a new major release every year. These will alternate between Long Term Support releases, which receive 3 years of support, and Current releases, which receive 15 months support.
- Portable Class Libraries (PCLs) attempted to solve the problem of sharing code between .NET platforms by allowing you to write code to the logical intersection of each platform's BCL. Each additional platform you targeted meant fewer BCL APIs in common.
- .NET Standard defines a standard set of APIs that are available across all platforms that support it. You can write libraries that target a specific version of .NET Standard and they'll be compatible with any platform that supports that version of .NET Standard.
- Each version of .NET Standard is a superset of the previous. For example, .NET Standard 1.2 includes all the APIs from .NET Standard 1.1, which in turn includes all the APIs from .NET Standard 1.0.
- Each version of a platform supports a specific version of .NET Standard. For example, .NET Framework 4.5.1 supports .NET Standard 1.2 (and hence also .NET Standard 1.1 and 1.0).
- .NET Framework 4.6.1 technically only supports .NET Standard 1.4. Thanks to a compatibility shim, you can reference .NET Standard 2.0 libraries from a .NET Framework 4.6.1 app. Similarly, you can reference a .NET Framework library from a .NET Standard 2.0 library, which wouldn't be possible without the shim.
- If you rely on the compatibility shim to reference a .NET Framework 4.6.1 library from a .NET Standard 2.0 library, and the referenced library uses .NET Framework-specific APIs, you'll get an exception at runtime.
- An app must target a .NET platform implementation, such as .NET 5 or .NET Core 3.1. It can't target .NET Standard.
- .NET 5 supports .NET Standard 2.1. It can reference any .NET Standard library, and any .NET Core library.

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

Appendix C
Useful references

In this appendix, I provide a number of links and references that I've found useful for learning about .NET Core/.NET 5, .NET Standard, and ASP.NET Core.

C.1 Relevant books

In this book, we touched on several topics and aspects of the .NET ecosystem that are somewhat peripheral to building ASP.NET Core applications. For a deeper understanding of those topics, I recommend the books in this section. They cover areas that you'll inevitably encounter when building ASP.NET Core applications:

- Khorikov, Vladimir. Unit Testing Principles, Patterns, and Practices. Manning, 2020. https://livebook.manning.com/book/unit-testing. Learn to refine your unit tests using modern best practices in this excellent book that contains examples in C#.
- Metzgar, Dustin. *.NET Core in Action*. Manning, 2018. https://livebook.manning.com/book/dotnet-core-in-action. ASP.NET Core apps are built using .NET Core and .NET 5. .NET Core in Action provides everything you need to know about running on the platform.
- Osherove, Roy. *The Art of Unit Testing*, second edition. Manning, 2013. https://livebook.manning.com/book/the-art-of-unit-testing-second-edition. In this book (*ASP.NET Core in Action*), I discuss the mechanics of unit testing ASP.NET Core applications. For a deeper discussion of how to create your tests, I recommend *The Art of Unit Testing*.
- Sainty, Chris. *Blazor in Action*. Manning, 2021. https://livebook.manning.com/book/blazor-in-action. Blazor is an exciting new framework that uses the power of industry standard WebAssembly to run .NET in the browser. With Blazor you can build single-page applications, just as you would with a JavaScript framework like Angular or React, but using the C# language and tooling that

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

you already know.

- Smith, Jon P. *Entity Framework Core in Action,* second edition. Manning, 2021. https://livebook.manning.com/book/entity-framework-core-in-action-second-edition. If you're using EF Core in your apps, I highly recommend *Entity Framework Core in Action*. It covers all the features and pitfalls of EF Core, as well as how to tune your app for performance.
- Van Deursen, Steven, and Mark Seemann, *Dependency Injection Principles, Practices, and Patterns*. Manning, 2019. https://livebook.manning.com/book/dependency-injection-principles-practices-patterns. Dependency injection is a core aspect of ASP.NET Core, so *Dependency Injection Principles, Practices, and Patterns* is especially relevant now. It introduces the patterns and antipatterns of dependency injection in the context of .NET and the C# language.

C.2 Announcement blog posts

When Microsoft releases a new version of ASP.NET Core or .NET Core, they typically write an announcement blog post. These posts provide a high-level overview of the topic, with many examples of new features. They're a great place to start if you want to quickly get acquainted with a topic:

- De la Torre, Cesar. "Web Applications with ASP.NET Core Architecture and Patterns guidance (Updated for .NET Core 2.0)," *.NET Blog* (blog), Microsoft, August 9, 2017, http://mng.bz/t415. Blog post introducing a free e-book on how to architecture modern web apps using ASP.NET Core and Azure.
- Fritz, Jeffrey T. "Announcing ASP.NET Core 2.0," *.NET Blog* (blog), Microsoft, August 14, 2017, http://mng.bz/0004. Announcement blog post for ASP.NET Core 2.0. Describes how to upgrade a project from 1.x to 2.0 and introduces some of the features specific to ASP.NET Core 2.0.
- Lander, Rich. "Announcing .NET Core 2.0," *.NET Blog* (blog), Microsoft, August 14, 2017, https://blogs.msdn.microsoft.com/dotnet/2017/08/14/ announcing-net-core-2-0/. Announcement blog post for .NET Core 2.0, describing the new features compared to .NET Core 1.x.
- Lander, Rich. "Introducing .NET 5". *.NET Blog (blog)*, Microsoft, May 6, 2019, https://devblogs.microsoft.com/dotnet/introducing-net-5/. The announcement blog post for .NET 5, describing the vision for the platform.
- Landwerth, Immo. "The future of .NET Standard," *.NET Blog* (blog), Microsoft, September 15, 2020, https://devblogs.microsoft.com/dotnet/the-future-of-net-standard/. A discussion on what .NET 5 means for the future of .NET Standard, including guidance for library authors.
- Landwerth, Immo. ".NET Standard—Demystifying .NET Core and .NET Standard," *Microsoft Developer Network*, Microsoft, September, 2017, https://msdn.microsoft.com/en-us/magazine/mt842506.aspx. Long post explaining

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

introducing .NET Core and explaining where .NET Standard fits in the .NET -ecosystem.

- Microsoft, ".NET Core and .NET 5 Support Policy,"
- Microsoft Docs, ".NET Core and .NET 5 Support Policy." Microsoft. https://dotnet.microsoft.com/platform/support/policy/dotnet-core. Microsoft's official support policy for .NET Core and .NET 5.

C.3 Microsoft documentation

Historically, Microsoft documentation has been poor, but with ASP.NET Core there has been a massive push to ensure the docs are useful and current. You can find walkthroughs, targeted documentation for specific features, documentation for supported APIs, and even an in-browser C# compiler:

- Microsoft Docs, ".NET API Browser." Microsoft Docs. https://docs.microsoft.com/dotnet/api/. This is an API browser, which can be used to work out which .NET APIs are available on which .NET platforms.
- Miller, Rowan, Brice Lambson, Maria Wenzel, Diego Vega, and Martin Milan. "Entity Framework Core Quick Overview." Microsoft Docs. September 20, 2020. https://docs.microsoft.com/ef/core/. This is the official documentation for EF Core.
- Microsoft. "Introduction to ASP.NET Core." Microsoft Docs. https://docs.microsoft.com/aspnet/core/. This is the official documentation for ASP.NET Core.
- Microsoft Docs, "Cross-platform targeting", Microsoft Docs. https://docs.microsoft.com/dotnet/standard/library-guidance/cross-platform-targeting. The official guidance on choosing a target framework for your libraries.

C.4 Security-related links

Security is an important aspect of modern web development. This section contains some of the references I refer to regularly, which describe some best practices for web development, as well as practices to avoid:

- Allen, Brock, and Dominick Baier. "IdentityServer4 1.0.0 documentation." https://identityserver4.readthedocs.io/. Documentation for IdentityServer, the OpenID Connect and OAuth 2.0 framework for ASP.NET Core.
- Baier, Dominick. *Dominick Baier on Identity & Access Control* (blog). https://leastprivilege.com/. The personal blog of Dominick Baier, co-author of IdentityServer. A great resource when working with authentication and authorization in ASP.NET Core.
- Microsoft Docs. "Overview of ASP.NET Core Security." Microsoft Docs. October 24, 2018. https://docs.microsoft.com/aspnet/core/security/. The home page of the official ASP.NET Core documentation for all things security related.
- Helme, Scott. *Scott Helme* (blog). https://scotthelme.co.uk/. Scott Helme's blog, with advice on security standards, especially security headers you can add to your

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

application.

- Helme, Scott. "SecurityHeaders.io—Analyse your HTTP response headers.". https://securityheaders.com/. Test your website's security headers, and get advice on why and how you should add them to your app.
- Hunt, Troy. *Troy Hunt* (blog). https://www.troyhunt.com. Personal blog of Troy Hunt with security-related advice for web developers, particularly .NET developers.

C.5 ASP.NET Core GitHub repositories

ASP.NET Core is entirely open source and developed on GitHub. One of the best ways I've found to learn about the framework is to browse the source code itself. This section contains the main repositories for ASP.NET Core, .NET Core, and EF Core:

- .NET Foundation. ".NET Runtime." The .NET CoreCLR runtime and BCL libraries, as well as extension libraries. https://github.com/dotnet/runtime.
- .NET Foundation. ".NET SDK and CLI." The .NET command line interface (CLI), assets for building the .NET SDK, and project templates. https://github.com/dotnet/sdk.
- .NET Foundation. "ASP.NET Core" The framework libraries that make up ASP.NET Core. https://github.com/dotnet/aspnetcore.
- .NET Foundation. "Entity Framework Core." The EF Core library. https://github.com/dotnet/efcore.

C.6 Tooling and services

This section contains links to tools and services you can use to build ASP.NET Core projects:

- .NET SDK: https://dotnet.microsoft.com/download.
- Cloudflare is a global content delivery network you can use to add caching and HTTPS to your applications for free https://www.cloudflare.com/.
- Let's Encrypt is a free, automated, and open Certificate Authority. You can use it to obtain free SSL certificates to secure your application: https://letsencrypt.org/.
- Rehan Saeed, Muhammed. ".NET Boxed" https://github.com/Dotnet-Boxed/Templates. A comprehensive collection of templates to get started with ASP.NET Core, preconfigured with many best practices.
- Visual Studio, Visual Studio for Mac, and Visual Studio Code: https://www.visualstudio.com/.
- JetBrains Rider: https://www.jetbrains.com/rider/.

C.7 ASP.NET Core blogs

This section contains blogs that focus on ASP.NET Core. Whether you're trying to get an overview of a general topic, or trying to solve a specific problem, it can be useful to have multiple viewpoints on a topic.

- .NET Team. *.NET Blog (blog)*. Microsoft, https://blogs.msdn.microsoft.com/dotnet. The

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

.NET team's blog, lots of great links.

- Alcock, Chris. *The Morning Brew (blog)*. http://blog.cwa.me.uk/. A collection of .NET-related blog posts, curated daily.
- Boden, Damien. *Software Engineering (blog)*. https://damienbod.com/. Excellent blog by Microsoft MVP Damien Boden on ASP.NET Core, lots of posts about ASP.NET Core with Angular.
- Brind, Mike. *Mikesdotnetting (blog).* https://www.mikesdotnetting.com/. Mike Brind has many posts on ASP.NET Core, especially focused on ASP.NET Core Razor Pages.
- Hanselman, Scott. *Scott Hanselman (blog)*. https://www.hanselman.com/blog/. Renowned speaker Scott Hanselman's personal blog. A highly diverse blog focused predominantly on .NET.
- Lock, Andrew. *.NET Escapades (blog).* https://andrewlock.net. My personal blog focused on ASP.NET Core.
- Pine, David. *IEvangelist (blog)*. http://davidpine.net/. Personal blog of Microsoft MVP David Pine, with lots of posts on ASP.NET Core.
- Rehan Saeed, Muhammed. *Muhammed Rehan Saeed (blog)*. https://rehansaeed.com. Personal blog of Muhammad Rehan Saeed, Microsoft MVP and author of the .NET Boxed project (linked in section C.6).
- Strahl, Rick. *Rick Strahl's Web Log (blog)*. https://weblog.west-wind.com/. Excellent blog by Rick Strahl covering a wide variety of ASP.NET Core topics.
- Gordon, Steve. *Steve Gordon – Code with Steve (blog)*. https://www.stevejgordon.co.uk/. Personal blog of Steve Gordon focused on .NET. Often focused on writing high-performance code with .NET.
- Wojcieszyn, Filip. *StrathWeb (blog)*. https://www.strathweb.com. Lots of posts on ASP.NET Core and ASP.NET from Filip, a Microsoft MVP.
- Abuhakmeh, Khalid. *Abuhakmeh (blog)*. https://khalidabuhakmeh.com/. A wide variety of posts from Khalid, focused on .NET and software development in general.

C.8 Video links

If you prefer video for learning a subject, I recommend checking out the links in this section. In particular, the ASP.NET Core community standup provides great insight into the changes you'll see in future ASP.NET Core versions, straight from the team building the framework.

- .NET Foundation. ".NET Community Standup." https://dotnet.microsoft.com/platform/community/standup. Weekly videos with the ASP.NET Core team discussing development of the framework. Also includes standups with the .NET team, the Xamarin team, and the EF Core team.
- Landwerth, Immo. ".NET Standard—Introduction." YouTube video, 10:16 minutes. Posted November 28, 2016 https://www.youtube.com/watch?v=YI4MurjfMn8. The first video in an excellent series on .NET standard.
- Microsoft. "Channel 9: Videos for developers from the people building Microsoft

©Manning Publications Co. To comment go to liveBook

Licensed to Angela Lutz <angelalutz1297@yahoo.com>

products and services." https://channel9.msdn.com/. Microsoft's official video channel. Contains a huge number of videos on .NET and ASP.NET Core, among many others.

- Wildermuth, Shawn. "Building a Web App with ASP.NET Core, MVC, Entity Framework Core, Bootstrap, and Angular." Pluralsight course, 9:52 hours. Posted October 7, 2019. https://www.pluralsight.com/courses/aspnetcore-mvc-efcore-bootstrap-angular-web. Shawn Wildermuth's course on building an ASP.NET Core application.
- Gordon, Steve, "Integration Testing ASP.NET Core Applications: Best Practices". Pluralsight course, 3:25 hours. Posted July 15, 2020. https://www.pluralsight.com/courses/integration-testing-asp-dot-net-core-applications-best-practices. One of several courses from Steve Gordon providing guidance and advice on building ASP.NET Core applications

©Manning Publications Co. To comment go to liveBook