CHAPTER 14

Working with ASP.NET Core

The examples in this chapter show the same simple authentication and authorization policy implemented in different ways. This may seem repetitive, but I start with a completely custom solution and finish with one that is expressed in just a few code statements, with each iteration using less custom code and more of the built-in features that ASP.NET Core provides.

In this part of the book, I reintroduce terms and concepts that are also described in Part 1 so that you can more easily find the information you need if you have to refer to a chapter in the future.

The objective of this chapter is to show you how requests flow through ASP.NET Core, which provides an important foundation for understanding how Identity works, as described in later chapters. Table 14-1 puts the creation of custom authorization and authentication in context.

Table 14-1. *Putting Custom Authorization and Authentication in Context*

Question	Answer
What is it?	Custom authentication and authorization work directly with the request pipeline to inspect and evaluate requests.
Why is it useful?	The request pipeline is at the heart of ASP.NET Core, and its features provide the foundation for ASP.NET Core Identity.
How is it used?	Middleware is added to the pipeline that can inspect and modify requests and generate responses. You can write custom middleware for every aspect of an application's operation, but Microsoft provides built-in features, which means this is not required for most projects.
Are there any pitfalls or limitations?	There is no real advantage in writing custom code, other than to understand how ASP.NET Core functions.
Are there any alternatives?	Custom authentication and authorization should not be used in real projects because Microsoft provides ready-to-use alternatives.

A. Freeman, *Pro ASP.NET Core Identity*, https://doi.org/10.1007/978-1-4842-6858-2_14

Table 14-2 summarizes the chapter.

Table 14-2.. *Chapter Summary*

Problem	Solution	Listing
Handle requests in ASP.NET Core	Add a middleware component to the request pipeline.	1–2
Authenticate a request	Create a `ClaimsPrincipal` object that describes the user's identity.	4–6
Supplement the information known about a user	Add a middleware component that expresses the supplemental information as claims.	7–9
Determine if a request should be granted access to the resource it targets	Assess the claims associated with the request to see if they meet the expected user characteristics.	10–11
Remove the need for the credentials to be required for every request	Create a middleware component that creates a token—such as a cookie—that can be presented in subsequent requests.	12–14, 18–27
Define authorization policies in endpoints	Use the `Authorize` attribute and create an implementation of the `IAuthorizationHandler` interface.	15–17

Preparing for This Chapter

This chapter uses the `ExampleApp` project created in Chapter 13. Most of the features I describe in this chapter are provided by the ASP.NET Core platform and not Razor Pages or the MVC Framework. To maintain focus, I disable all but the essential features and reintroduce them later.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from `https://github.com/Apress/pro-asp.net-core-identity`. See Chapter 1 for how to get help if you have problems running the examples.

Add a class file named `SecretEndpoint.cs` to the `ExampleApp` folder and add the code shown in Listing 14-1.

Listing 14-1. The Contents of the SecretEndpoint.cs File in the ExampleApp Folder

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

namespace ExampleApp {

    public class SecretEndpoint {

        public static async Task Endpoint(HttpContext context) {
            await context.Response.WriteAsync("This is the secret message");
        }
    }
}
```

Replace the contents of the Startup.cs file with the code shown in Listing 14-2, which enables only the most basic ASP.NET Core features.

Listing 14-2. Configuring the Application in the Startup.cs File in the ExampleApp Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace ExampleApp {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

            app.UseRouting();

            app.UseEndpoints(endpoints => {
                endpoints.MapGet("/", async context => {
                    await context.Response.WriteAsync("Hello World!");
                });
                endpoints.MapGet("/secret", SecretEndpoint.Endpoint)
                    .WithDisplayName("secret");
            });
        }
    }
}
```

Use the command prompt to run the command shown in Listing 14-3, which will compile and start the project.

Listing 14-3. Running the Example Project

```
dotnet run
```

Once the ASP.NET Core HTTP server has started, open a new browser window and request http://localhost:5000 and http://localhost:5000/secret. You will see the responses shown in Figure 14-1.

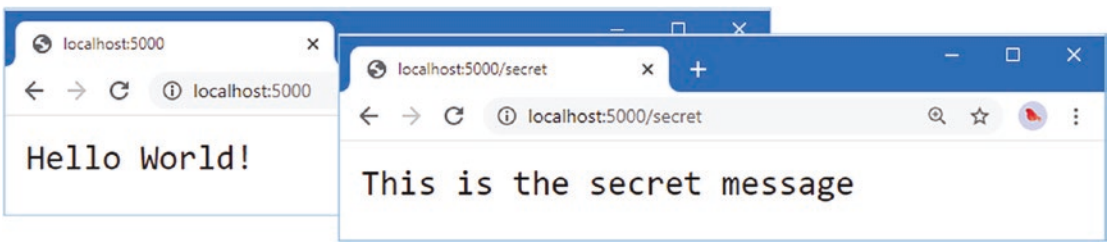

Figure 14-1. *Running the example project*

Understanding the ASP.NET Core Request Flow

The ASP.NET Core platform creates three objects when it receives an HTTP request: an `HttpRequest` object that describes the request, an `HttpResponse` object that describes the response that will be returned to the client, and an `HttpContext` object that provides access to ASP.NET Core features.

To process the request, ASP.NET Core passes the `HttpContext`, `HttpRequest`, and `HttpResponse` objects to its *middleware components*. Middleware components use the objects to inspect or modify the request and contribute to the response. The middleware components are arranged in a sequence, known as the *request pipeline*, and the ASP.NET Core objects are passed to the ASP.NET Core objects in order. Once the request gets to the last component in the pipeline, the objects are passed back along the line until they return to the start, at which point ASP.NET Core uses the `HttpResponse` object to send the response to the client, as shown in Figure 14-2.

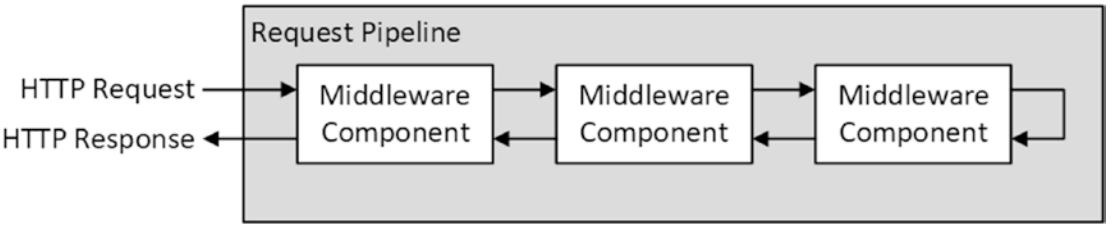

Figure 14-2. *ASP.NET Core request handling*

Understanding the Endpoint Routing Middleware

Middleware components can be as simple as a single function or as complex as a complete framework. The changes to the `Startup` class in Listing 14-2 defines a pipeline with two related middleware components.

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace ExampleApp {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

            app.UseRouting();

            app.UseEndpoints(endpoints => {
                endpoints.MapGet("/", async context => {
                    await context.Response.WriteAsync("Hello World!");
                });
```

```
                endpoints.MapGet("/secret", SecretEndpoint.Endpoint)
                    .WithDisplayName("secret");
            });
        }
    }
}
```

The highlighted statements enable *endpoint routing*, which matches requests to *endpoints*, which are functions or classes that generate responses for specific URLs. The `UseRouting` method sets up a middleware component that inspects the HTTP request and attempts to match its URL using one of the endpoints specified by the `UseEndpoints` method, which uses the selected endpoint to produce a response.

■ **Note** When I added an endpoint in Listing 14-2, I used the `WithDisplayName` method, which is a useful feature that makes it easier to identify the endpoint that has been selected to produce a response.

There are two endpoints in the example application. The default endpoint responds to requests for the `/` URL with a `Hello, World` message and was added to the project when it was created. The other endpoint also responds with a text message.

```
...
app.UseEndpoints(endpoints => {
    endpoints.MapGet("/", async context => {
        await context.Response.WriteAsync("Hello World!");
    });
    endpoints.MapGet("/secret", SecretEndpoint.Endpoint)
        .WithDisplayName("secret");
});
...
```

In later examples, I replace the default endpoint with more complex ASP.NET Core features, but simple text responses are enough for now. The use of endpoint routing produces a request pipeline with three distinct phases, as shown in Figure 14-3.

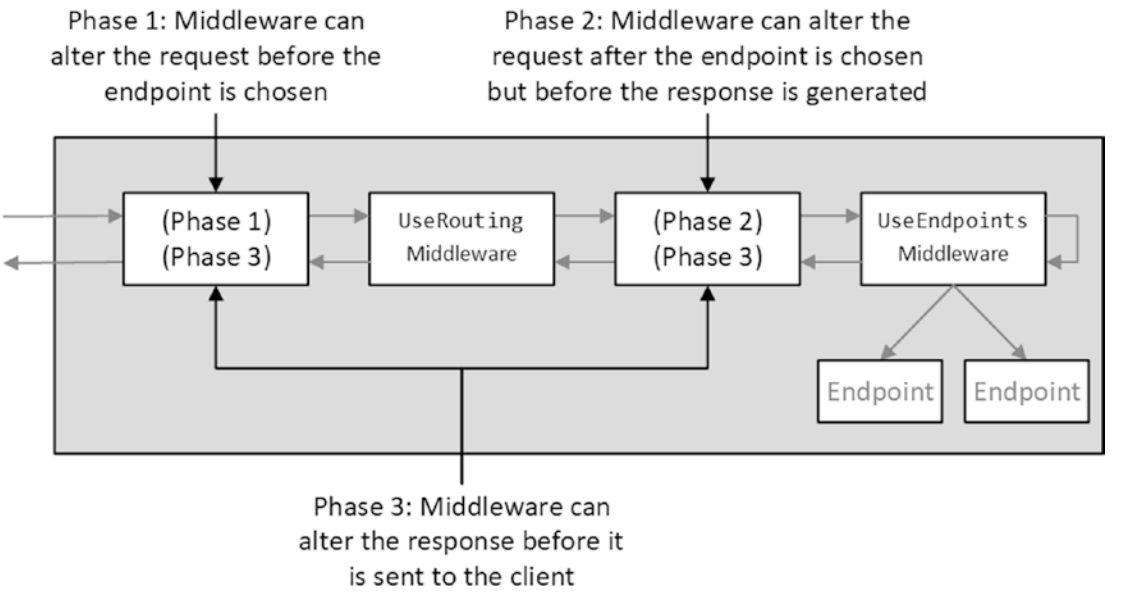

Figure 14-3. *The effect of endpoint routing on the request pipeline*

Because HTTP requests are passed along the pipeline in both directions, middleware can work alongside the endpoint routing system and provides opportunities for adding authentication and authorization to ASP.NET Core applications, as you will see shortly.

An important aspect of the ASP.NET Core request flow is that middleware components can short-circuit the pipeline and generate a response without passing on the request to the remaining components, as shown in Figure 14-4.

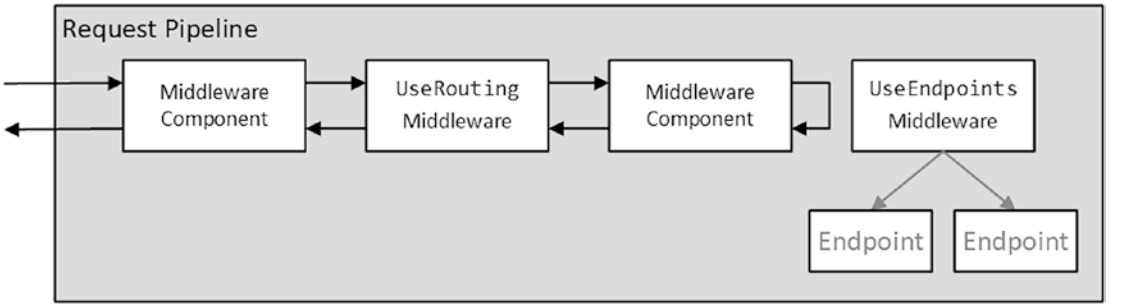

Figure 14-4. *Short-circuiting the request pipeline*

Authenticating and Authorizing Request Flow

There is a natural fit in the request flow for authentication and authorization. Providing credentials to an application can be a complex process. In most web applications, credentials are offered once and result in a token, often an HTTP cookie, that can be presented in subsequent requests. This is known as *signing in*, and invalidating the token is known as *signing out*.

Once the user is signed in, the client includes the token in requests, which allows the user to be identified as the source of the request without presenting credentials again, which is known as *authenticating the request*. Request authentication is performed as early as possible in the request pipeline, in Phase 1, so that subsequent middleware components can use or add to the information the request contains about the user.

Once request authentication has established the identity of the user, authentication middleware can be used in Phase 2 to control access to the endpoint that has been selected to produce a request. The authentication middleware has three possible responses:

1. If the user is allowed access, then do nothing, and the request will continue along the pipeline to the endpoint, which will produce a response.
2. If the request is not authenticated, short-circuit the pipeline with an HTTP 401 status code, which challenges the user to provide credentials. This is known as a *challenge* response.
3. If the request is authenticated but the user is not allowed access, short-circuit the pipeline with an HTTP 403 status code. This is known as a *forbidden* response.

Figure 14-5 shows the authentication and authorization middleware and endpoints in the request pipeline.

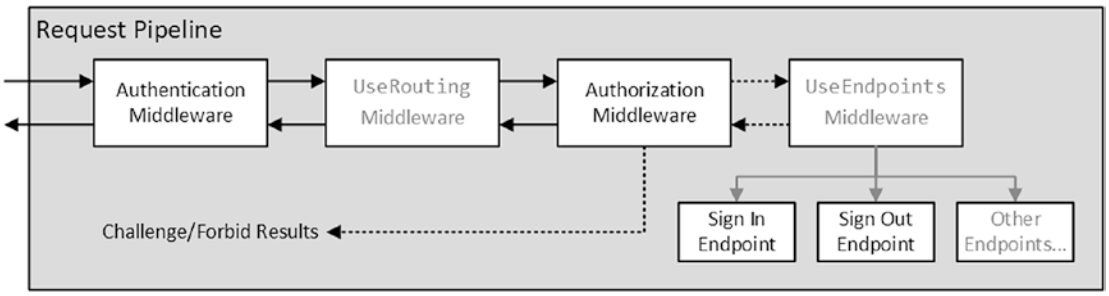

Figure 14-5. *Authentication and authorization in the request pipeline*

In many applications, the challenge and forbid responses are not just HTTP status codes but redirections to a web page that allows authentication or displays a meaningful error message. I start with status code responses and introduce HTML responses later in this chapter.

Understanding Claims

The middleware responsible for authenticating requests needs to describe a user so the authorization middleware can decide whether to send a challenge or forbidden response. ASP.NET Core provides a set of standard classes that are not specific to a single approach to authentication and that are used to consistently describe a user throughout an application.

These classes use *claims-based authentication*, which can be confusing at first because it is designed to be open and flexible and, as a result, can feel vague and ill-defined. A user is represented by a `ClaimsPrincipal` object. Each user can have multiple identities, which are represented by `ClaimsIdentity` objects. An identity contains one or more pieces of information about the user, each of which is represented by a `Claim`.

Create a `ExampleApp/Custom` folder, add to it a class file named `CustomAuthentication.cs`, and use it to define the middleware component shown in Listing 14-4.

Listing 14-4. The Contents of the CustomAuthentication.cs File in the Custom Folder

```
using Microsoft.AspNetCore.Http;
using System.Security.Claims;
using System.Threading.Tasks;

namespace ExampleApp.Custom {

    public class CustomAuthentication {
        private RequestDelegate next;

        public CustomAuthentication(RequestDelegate requestDelegate)
                => next = requestDelegate;

        public async Task Invoke(HttpContext context) {
            string user = context.Request.Query["user"];
            if (user != null) {
                Claim claim = new Claim(ClaimTypes.Name, user);
                ClaimsIdentity ident = new ClaimsIdentity("QueryStringValue");
                ident.AddClaim(claim);
                context.User = new ClaimsPrincipal(ident);
            }
            await next(context);
        }
    }
}
```

This middleware component handles sign-in and authentication as a single step, which means that the user needs to present their credentials for every request. This would normally be an unreasonable burden on the user, but this component does something that is only sensible in an example project: it trusts the user to provide their username in the request query string, like this:

```
...
string user = context.Request.Query["user"];
...
```

If the request contains the query string value, it is used to authenticate the request.

■ **Caution** It should be obvious that this is not a suitable way to identify users in real projects. This part of the book is about explaining the features ASP.NET Core provides. See Part 1 for real workflows that are project-ready.

Several objects are required to authenticate the request. Claims are used to describe every piece of available information about a user, including the user's name. To that end, a `Claim` object is created.

```
...
Claim claim = new Claim(ClaimTypes.Name, user);
...
```

Claim objects are created with a type and a value. The ClaimTypes class provides a series of constant string values that are used to specify the claim type, which denotes the type of information the claim represents. In this case, the claim contains the username, so I have used the ClaimTypes.Name property as the type. There lots of claim types available—and you can easily create your own—but Table 14-3 lists some commonly used ClaimTypes properties.

Table 14-3. *Commonly Used Claim Types*

Name	Description
Name	This claim type denotes the user's name and is typically used for the user account name.
Role	This claim types denotes a role, which is often used for access control.
Email	This claim types denotes an email address.
GivenName	This claim type denotes the user's given name.
Surname	This claim type denotes the user's surname.

The next step is to create an identity for the user, like this:

```
...
ClaimsIdentity ident = new ClaimsIdentity("QueryStringValue");
ident.AddClaim(claim);
...
```

Individual claims are grouped together in an identity. There are no definitive rules about what an identity represents. That means there can be one identity for all a user's claims or, perhaps, multiple identities to represent how a user is known to multiple back-end systems, depending on what makes the most sense for an application.

■ **Tip** A sensible approach for most web applications is to start with a single identity and use it to group together all the user's claims. You can always add identities if need be, but that won't be required for most projects.

The ClaimsIdentity constructor accepts a string that denotes the authentication type, which I have set to QueryStringValue for this example. There is only one Claim object for the user, which I associate with the identity using the AddClaim method.

The final step is to create a ClaimsPrincipal object using the identity as an argument and assign the ClaimsPrincipal to the User property of the HttpContext object, like this:

```
...
context.User = new ClaimsPrincipal(ident);
...
```

When the middleware component in Listing 14-5 processes a request, it will check for the query string value and, if it is present, authenticate the request by creating a ClaimsPrincipal object that is passed along the request pipeline with the HttpContext object so that details of the user's claims can be processed by other middleware components.

To demonstrate how claims are handled, add a class file named ClaimsReporter.cs to the ExampleApp/Custom folder and use it to define the middleware component shown in Listing 14-5, which writes out the claims associated with a request.

Listing 14-5. The Contents of the ClaimsReporter.cs File in the Custom Folder

```
using Microsoft.AspNetCore.Http;
using System;
using System.IO;
using System.Linq;
using System.Security.Claims;
using System.Threading.Tasks;

namespace ExampleApp.Custom {
    public class ClaimsReporter {
        private RequestDelegate next;

        public ClaimsReporter(RequestDelegate requestDelegate)
                => next = requestDelegate;

        public async Task Invoke(HttpContext context) {

            ClaimsPrincipal p = context.User;

            Console.WriteLine($"User: {p.Identity.Name}");
            Console.WriteLine($"Authenticated: {p.Identity.IsAuthenticated}");
            Console.WriteLine("Authentication Type "
                + p.Identity.AuthenticationType);

            Console.WriteLine($"Identities: {p.Identities.Count()}");
            foreach (ClaimsIdentity ident in p.Identities) {
                Console.WriteLine($"Auth type: {ident.AuthenticationType},"
                    + $" {ident.Claims.Count()} claims");
                foreach (Claim claim in ident.Claims) {
                    Console.WriteLine($"Type: {GetName(claim.Type)}, "
                        +$"Value: {claim.Value}, Issuer: {claim.Issuer}");
                }
            }
            await next(context);
        }

        private string GetName(string claimType) =>
            Path.GetFileName(new Uri(claimType).LocalPath);
    }
}
```

The code in Listing 14-5 looks more complex than it is because there are several template strings. When the middleware component receives the request, it uses the HttpContext.User property to get the ClaimsPrincipal object and writes out details to the console. The ClaimsPrincipal object defines an Identity property that returns the first identity associated with the user. The Identity property returns an object that implements the IIdentity interface, which provides basic information about the identity using the properties described in Table 14-4.

Table 14-4. *The IIdentity Properties*

Name	Description
`Name`	Returns the value of the identity's name claim
`IsAuthenticated`	Returns true if the identity has been authenticated
`AuthenticationType`	Returns the string that identities the source of the identity and its claims

I use the `IIdentity` properties to display a summary of the first identity.

```
...
Console.WriteLine($"User: {p.Identity.Name}");
Console.WriteLine($"Authenticated: {p.Identity.IsAuthenticated}");
Console.WriteLine("Authentication Type " + p.Identity.AuthenticationType);
...
```

The `ClaimsPrincipal` class also provides access to all the identities associated with the user and the complete set of claims, and it provides some convenience members that make access control easier, as I demonstrate later in this chapter.

```
...
Console.WriteLine($"Identities: {p.Identities.Count()}");
foreach (ClaimsIdentity ident in p.Identities) {
    Console.WriteLine($"Auth type: {ident.AuthenticationType},"
         + $" {ident.Claims.Count()} claims");
    foreach (Claim claim in ident.Claims) {
        Console.WriteLine($"Type: {GetName(claim.Type)}, "
            +$"Value: {claim.Value}, Issuer: {claim.Issuer}");
    }
}
...
```

Listing 14-6 adds both new middleware components to the request pipeline.

Listing 14-6. Adding Middleware Components in the Startup.cs File in the ExampleApp Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ExampleApp.Custom;

namespace ExampleApp {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }
```

```
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

            app.UseMiddleware<CustomAuthentication>();

            app.UseRouting();

            app.UseMiddleware<ClaimsReporter>();

            app.UseEndpoints(endpoints => {
                endpoints.MapGet("/", async context => {
                    await context.Response.WriteAsync("Hello World!");
                });
                endpoints.MapGet("/secret", SecretEndpoint.Endpoint)
                    .WithDisplayName("secret");
            });
        }
    }
}
```

Restart ASP.NET Core and request `http://localhost:5000/?user=Alice`, and you will see the following messages displayed at the command prompt:

```
...
User: Alice
Authenticated: True
Authentication Type QueryStringValue
Identities: 1
Auth type: QueryStringValue, 1 claims
Type: name, Value: Alice, Issuer: LOCAL AUTHORITY
...
```

The properties provided by the `IIdentity` interface are mapped onto claims so that the `IIdentity.Name` property locates the claim I created in Listing 14-6, for example. By default, the source of each claim is `LOCAL AUTHORITY`, indicating that the data originated within the application. You will see examples of claims from other sources when I show you how to authenticate using third-party services in Chapter 23.

UNDERSTANDING CLAIM TYPE URLS

The values of the `ClaimTypes` properties used to create claims are URLs so that the value of the `ClaimTypes.Name` property, for example, is this URL:

`http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name`

Unfortunately, this URL doesn't work, and it is difficult to get definitive descriptions of claim types and how they should be used. Consequently, claims are used inconsistently by applications, and custom claims are created freely. This can be a problem if you are trying to write code that is agnostic about the authentication systems it supports but isn't a problem in most web application projects where the development team can tailor the application to the specific claims in use.

To display the claim type in Listing 14-6, I used the .NET `Uri` and `Path` classes to process the URL and just write out the last segment of the URL path, like this:

```
...
private string GetName(string claimType) =>
            Path.GetFileName(new Uri(claimType).LocalPath);
...
```

In real projects, you will usually look for claims using the `ClaimTypes` property names, which is more convenient than dealing with the cumbersome URLs, as demonstrated in Part 1.

The `IIdentity.IsAuthenticated` property returns `true` if the `ClaimsIdentity` has been created with an authentication type constructor argument, like this:

```
...
ClaimsIdentity ident = new ClaimsIdentity("QueryStringValue");
...
```

It is important not to assume that a request has been authenticated just because the `HttpContext.User` property returns an object. ASP.NET Core creates a `ClaimsPrincipal` object with a `ClaimsIdentity` for all `HttpContext` objects by default, although no claims are created and no authentication type is specified, so that the `IsAuthenticated` property returns `false`.

You can see details of the default `ClaimsPrincipal` object by requesting `http://localhost:5000`, which will produce the following console output:

```
...
User:
Authenticated: False
Authentication Type
Identities: 1
Auth type: , 0 claims
...
```

This response denotes a request that has not been authenticated and for which no user information is available.

Adding Claims to a Request

Some applications have access to additional information about users that can be expressed as claims so it can be used by other middleware components. This additional information can be added to the `ClaimsPrincipal` associated with a request, either as claims added to an existing identity or as an additional identity.

Add a class file to the `ExampleApp` folder named `UsersAndClaims.cs` and add the code shown in Listing 14-7.

Listing 14-7. The Contents of the UsersAndClaims.cs File in the ExampleApp Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Security.Claims;
```

```
namespace ExampleApp {
    public static class UsersAndClaims {

        public static Dictionary<string, IEnumerable<string>> UserData
            = new Dictionary<string, IEnumerable<string>> {
               { "Alice", new [] { "User", "Administrator" } },
               { "Bob", new [] { "User" } },
               { "Charlie", new [] { "User"} }
            };

        public static string[] Users => UserData.Keys.ToArray();

        public static Dictionary<string, IEnumerable<Claim>> Claims =>
            UserData.ToDictionary(kvp => kvp.Key,
                kvp => kvp.Value.Select(role => new Claim(ClaimTypes.Role, role)),
                StringComparer.InvariantCultureIgnoreCase);
    }
}
```

Next, add a class file named RoleMemberships.cs to the ExampleApp/Custom folder and use it to define the middleware component shown in Listing 14-8.

Listing 14-8. The Contents of the RoleMemberships.cs File in the Custom Folder

```
using Microsoft.AspNetCore.Http;
using System.Security.Claims;
using System.Security.Principal;
using System.Threading.Tasks;

namespace ExampleApp.Custom {

    public class RoleMemberships {
        private RequestDelegate next;

        public RoleMemberships(RequestDelegate requestDelegate)
                => next = requestDelegate;

        public async Task Invoke(HttpContext context) {
            IIdentity mainIdent = context.User.Identity;
            if (mainIdent.IsAuthenticated
                    && UsersAndClaims.Claims.ContainsKey(mainIdent.Name)) {
                ClaimsIdentity ident = new ClaimsIdentity("Roles");
                ident.AddClaim(new Claim(ClaimTypes.Name, mainIdent.Name));
                ident.AddClaims(UsersAndClaims.Claims[mainIdent.Name]);
                context.User.AddIdentity(ident);
            }
            await next(context);
        }
    }
}
```

This middleware component uses the map of usernames to roles defined in Listing 14-7. When an authenticated request is received for a user in the map, a new identity is created with role claims, denoting that the user has been granted a specific role. Listing 14-9 adds the new middleware component to the request pipeline.

Listing 14-9. Adding a Middleware Component in the Startup.cs File in the ExampleApp Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ExampleApp.Custom;

namespace ExampleApp {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

            app.UseMiddleware<CustomAuthentication>();
            app.UseMiddleware<RoleMemberships>();
            app.UseRouting();

            app.UseMiddleware<ClaimsReporter>();

            app.UseEndpoints(endpoints => {
                endpoints.MapGet("/", async context => {
                    await context.Response.WriteAsync("Hello World!");
                });
                endpoints.MapGet("/secret", SecretEndpoint.Endpoint)
                    .WithDisplayName("secret");
            });
        }
    }
}
```

Restart ASP.NET Core and request `http://localhost:5000/?user=Alice`. The messages written to the console show two identities and a total of four claims.

```
...
User: Alice
Authenticated: True
Authentication Type QueryStringValue
Identities: 2
Auth type: QueryStringValue, 1 claims
Type: name, Value: Alice, Issuer: LOCAL AUTHORITY
Auth type: Roles, 3 claims
Type: name, Value: Alice, Issuer: LOCAL AUTHORITY
Type: role, Value: User, Issuer: LOCAL AUTHORITY
Type: role, Value: Administrator, Issuer: LOCAL AUTHORITY
...
```

Assessing Claims

The term *claim* reflects the fact that ASP.NET Core doesn't validate the data associated with a user and just passes it along the request pipeline. It is the responsibility of other middleware components to assess claims and decide if they should be trusted. Using query string authentication performed in the example application is dangerously insecure, for example, because it relies on users to honestly identify themselves and so claims arising from that source—and claims that are created based on the original claims—should be treated with suspicion.

Initially, however, I am going to continue as though that were not the case and assume that those claims are trustworthy and can be used for access control, just to demonstrate how claims can be used. Add a class file named `CustomAuthorization.cs` to the `ExampleApp/Custom` folder and use it to define the middleware component shown in Listing 14-10.

Listing 14-10. The Contents of the CustomAuthorization.cs File in the Custom Folder

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

namespace ExampleApp.Custom {

    public class CustomAuthorization {
        private RequestDelegate next;

        public CustomAuthorization(RequestDelegate requestDelegate)
                => next = requestDelegate;

        public async Task Invoke(HttpContext context) {
            if (context.GetEndpoint()?.DisplayName == "secret") {
                if (context.User.Identity.IsAuthenticated) {
                    if (context.User.IsInRole("Administrator")) {
                        await next(context);
                    } else {
                        Forbid(context);
                    }
                } else {
                    Challenge(context);
                }
            } else {
                await next(context);
            }
        }

        public void Challenge(HttpContext context)
                => context.Response.StatusCode = StatusCodes.Status401Unauthorized;

        public void Forbid(HttpContext context)
                => context.Response.StatusCode = StatusCodes.Status403Forbidden;
    }
}
```

There are four possible outcomes when this middleware component processes a request. If the request isn't for the protected endpoint, then it is forwarded along the pipeline because there no authorization is required.

```
...
if (context.GetEndpoint()?.DisplayName == "secret") {
...
```

I use the null conditional operator (the ? character), also known as the *safe navigation operator*, to inspect the `DisplayName` property of the object returned by the `GetEndpoint` method, which returns the string assigned to the endpoint using the `WithDisplayName` method I used in Listing 14-11. No authorization is required if there is no endpoint or the endpoint's display name isn't `secret` and the request can be passed on.

If there is an endpoint and it has the right name, then I can check to see if the request has been authenticated.

```
...
if (context.User.Identity.IsAuthenticated) {
...
```

I am not checking the source of the authentication in this example. This is a commonly used approach in web applications that use ASP.NET Core Identity and where there is a single trustworthy source of authentication.

If the request has been authenticated, I check to see there is a claim for the `Administrator` role.

```
...
if (context.User.IsInRole("Administrator")) {
...
```

The `IsInRole` method is one of several useful methods defined by the `ClaimsPrincipal` class that work on all claims, regardless of which identity they are associated with, as described in Table 14-5.

Table 14-5. *Useful ClaimsPrincipal Methods*

Name	Description
`FindAll(type)`	This method locates all claims with the specified type. There is a version of this method that accepts a predicate function to select claims.
`FindFirst(type)`	This method locates the first claim with the specified type. There is a version of this method that accepts a predicate function to select a claim.
`HasClaim(type, value)`	This method returns true if there is a claim with the specified type and value. There is a version of this method that accepts a predicate function to check claims.
`IsInRole(role)`	This method returns true if there is a claim for the specified role. Identities can be configured with different claim types for roles, but the default type is `ClaimTypes.Role`.

If there is an `Administrator` role claim present, then the request is forwarded along the pipeline so that the protected endpoint can generate a response.

The remaining outcomes short-circuit the request pipeline and denote an authorization failure. A *challenge response* is sent when the request has not been authenticated.

```
...
public void Challenge(HttpContext context)
        => context.Response.StatusCode = StatusCodes.Status401Unauthorized;
...
```

A *forbid response* is sent when the request is authenticated but there is no `Administrator` role claim.

```
...
public void Forbid(HttpContext context)
    => context.Response.StatusCode = StatusCodes.Status403Forbidden;
...
```

The separate `Challenge` and `Forbid` methods will help make later examples clearer as I improve the way that the example application deals with authentication and authorization.

UNDERSTANDING THE 401 AND 403 HTTP STATUS CODES

The way I have used the 401 and 403 status codes matches the built-in features that ASP.NET Core provides, which you will see in later examples. Not all frameworks and applications use these status codes in the same way, and you will often find the 401 HTTP status code is used both for challenge and forbidden responses.

When sending a 401 response, the HTTP specification requires the use of the `WWW-Authenticate` header, which tells clients how they should authenticate users. I did not set this header, partly for simplicity and partly because the application does not authenticate users in a standard way.

Listing 14-11 adds the authorization middleware to the request pipeline.

Listing 14-11. Adding Middleware in the Startup.cs File in the ExampleApp Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ExampleApp.Custom;

namespace ExampleApp {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

            app.UseMiddleware<CustomAuthentication>();
            app.UseMiddleware<RoleMemberships>();
            app.UseRouting();
```

```
            app.UseMiddleware<ClaimsReporter>();
            app.UseMiddleware<CustomAuthorization>();

            app.UseEndpoints(endpoints => {
                endpoints.MapGet("/", async context => {
                    await context.Response.WriteAsync("Hello World!");
                });
                endpoints.MapGet("/secret", SecretEndpoint.Endpoint)
                    .WithDisplayName("secret");
            });
        }
    }
}
```

Restart ASP.NET Core and request `http://localhost:5000/secret?user=alice`. The custom middleware components will authenticate the request and add an `Administrator` role claim, which will allow the request to be authorized.

Request `http://localhost:5000/secret?user=bob`, and you will receive a forbidden response because the request is authenticated but doesn't have the required role claim. Request `http://localhost:5000/secret`, and you will receive the challenge response because the request is not authenticated. Figure 14-6 shows all three responses.

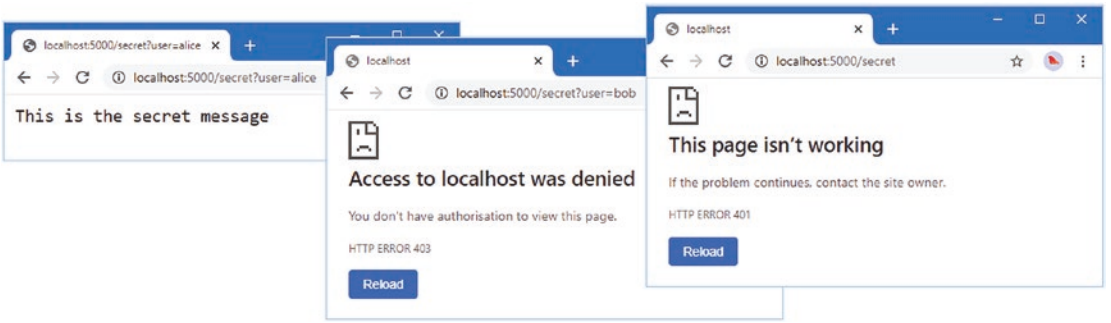

Figure 14-6. *Using custom middleware*

Improving the Authentication and Authorization

The example application has working authentication and authorization, but it is awkward to use. In the sections that follow, I improve the implementation and take advantage of more of the features that ASP.NET Core provides.

Signing In and Out of the Application

Even in my rudimentary system, providing credentials through the query string for every request is awkward. In most projects, there is an endpoint that allows the user to sign in to the application and receive a token in return. This token, which is often an HTTP cookie, is included in subsequent HTTP requests and used to authenticate requests. Another endpoint allows the user to sign out by invalidating the token so that it cannot be used again.

To allow users to sign in and out of the example application, add a class named CustomSignInAndSignOut.cs to the ExampleApp/Custom folder and add the code shown in Listing 14-12.

Listing 14-12. The Contents of the CustomSignInandSignOut.cs File in the Custom Folder

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

namespace ExampleApp.Custom {
    public class CustomSignInAndSignOut {

        public static async Task SignIn(HttpContext context) {
            string user = context.Request.Query["user"];
            if (user != null) {
                context.Response.Cookies.Append("authUser", user);
                await context.Response
                    .WriteAsync($"Authenticated user: {user}");
            } else {
                context.Response.StatusCode = StatusCodes.Status401Unauthorized;
            }
        }

        public static async Task SignOut(HttpContext context) {
            context.Response.Cookies.Delete("authUser");
            await context.Response.WriteAsync("Signed out");
        }
    }
}
```

The SignIn method will be an endpoint that gets the username from the query string and adds a cookie to the response containing the name, which the client will then include in subsequent requests. This is no more secure than earlier examples because it still trusts the user to honestly identify themselves, but it is more convenient and easier to use.

The SignOut method in Listing 14-12 will also be used as an endpoint. This method deletes the cookie so that subsequent requests cannot be authenticated. Listing 14-13 registers the SignIn and SignOut methods as endpoints in the request pipeline.

Listing 14-13. Adding Endpoints in the Startup.cs File in the ExampleApp Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ExampleApp.Custom;

namespace ExampleApp {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }
```

```
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

            app.UseMiddleware<CustomAuthentication>();
            app.UseMiddleware<RoleMemberships>();
            app.UseRouting();

            app.UseMiddleware<ClaimsReporter>();
            app.UseMiddleware<CustomAuthorization>();

            app.UseEndpoints(endpoints => {
                endpoints.MapGet("/", async context => {
                    await context.Response.WriteAsync("Hello World!");
                });
                endpoints.MapGet("/secret", SecretEndpoint.Endpoint)
                    .WithDisplayName("secret");
                endpoints.Map("/signin", CustomSignInAndSignOut.SignIn);
                endpoints.Map("/signout", CustomSignInAndSignOut.SignOut);
            });
        }
    }
}
```

Listing 14-14 shows the changes to the custom authentication code to use cookies, rather than the query string, to authenticate requests.

Listing 14-14. Using Cookies for Authentication in the CustomAuthentication.cs File in the Custom Folder

```
using Microsoft.AspNetCore.Http;
using System.Security.Claims;
using System.Threading.Tasks;

namespace ExampleApp.Custom {

    public class CustomAuthentication {
        private RequestDelegate next;

        public CustomAuthentication(RequestDelegate requestDelegate)
                => next = requestDelegate;

        public async Task Invoke(HttpContext context) {
            //string user = context.Request.Query["user"];
            string user = context.Request.Cookies["authUser"];
            if (user != null) {
                Claim claim = new Claim(ClaimTypes.Name, user);
                ClaimsIdentity ident = new ClaimsIdentity("QueryStringValue");
                ident.AddClaim(claim);
                context.User = new ClaimsPrincipal(ident);
            }
            await next(context);
        }
    }
}
```

Restart ASP.NET Core, and request `http://localhost:5000/signin?user=alice` to sign in as Alice. Once you are signed in, request `http://localhost:5000/secret`, and your request will be authenticated and authorized, as shown in Figure 14-7.

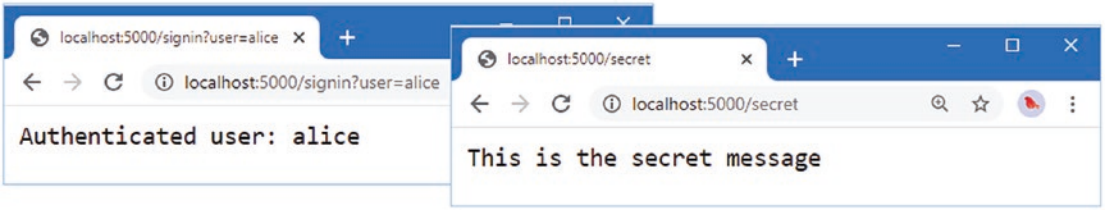

Figure 14-7. *Signing in to the application*

Next, request `http://localhost:5000/signin?user=bob` and then `http://localhost:5000/secret`, which will produce a forbidden result because the request won't have the required role claim. Finally, request `http://localhost:5000/signout` and request `http://localhost:5000/secret`, which will produce the challenge result because the request won't contain the cookie required for authentication. Figure 14-8 shows the forbidden and challenge responses. You will be able to see details of the identities and claims associated with all these requests in the messages displayed at the command prompt.

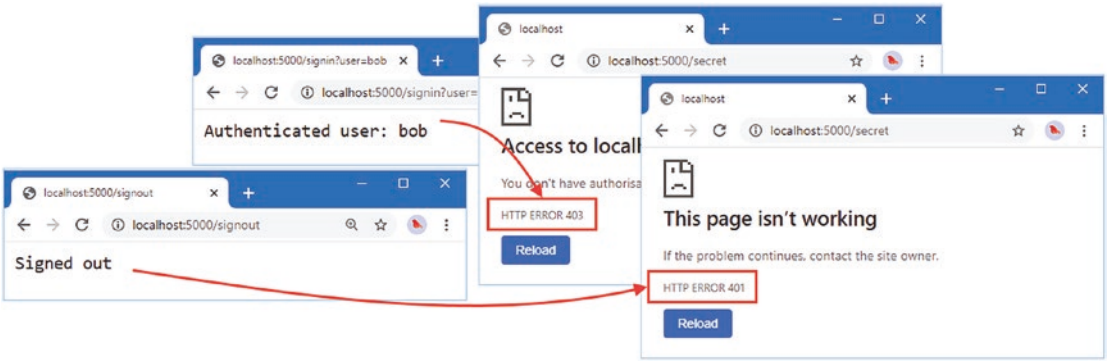

Figure 14-8. *Forbidden and challenge responses*

Defining Authorization Policy in the Endpoint

The examples so far in this chapter have taken advantage of the ASP.NET Core pipeline to authenticate and authorize requests using middleware. This allowed me to describe the different components in detail, but it led me to hard-code my authorization policy into a middleware component, which is not a flexible approach for real projects.

ASP.NET Core provides built-in middleware that allows authorization policies to be defined by applying attributes to endpoints. In Listing 14-15, I applied an attribute to the protected endpoint to describe its access control restrictions.

Listing 14-15. Applying an Attribute in the SecretEndpoint.cs File in the ExampleApp Folder

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

using Microsoft.AspNetCore.Authorization;

namespace ExampleApp {

    public class SecretEndpoint {

        [Authorize(Roles = "Administrator")]
        public static async Task Endpoint(HttpContext context) {
            await context.Response.WriteAsync("This is the secret message");
        }
    }
}
```

You will be familiar with the `Authorize` attribute if you have applied access control to Razor Pages or MVC Framework action methods. The `Authorize` attribute can be applied directly to endpoints, as I have done here, and the `Roles` argument specifies the role claims required for authorization. I describe other features provided by the `Authorize` attribute in Chapter 15.

Implementing the Authentication Handler Interface

ASP.NET Core supports a set of interfaces that work with the features provided by the built-in middleware components. In this section, I am going to implement the `IAuthorizationHandler` interface, which allows me to add a custom authentication scheme that will work with the `Authorize` attribute I applied in Listing 14-15. Add a class called `AuthHandler.cs` to the `ExampleApp/Custom` folder and use it to define the class shown in Listing 14-16.

Listing 14-16. The Contents of the AuthHandler.cs File in the Custom Folder

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Http;
using System.Security.Claims;
using System.Threading.Tasks;

namespace ExampleApp.Custom {
    public class AuthHandler : IAuthenticationHandler {
        private HttpContext context;
        private AuthenticationScheme scheme;

        public Task InitializeAsync(AuthenticationScheme authScheme,
                HttpContext httpContext) {
            context = httpContext;
            scheme = authScheme;
            return Task.CompletedTask;
        }
```

```
        public Task<AuthenticateResult> AuthenticateAsync() {
            AuthenticateResult result;
            string user = context.Request.Cookies["authUser"];
            if (user != null) {
                Claim claim = new Claim(ClaimTypes.Name, user);
                ClaimsIdentity ident = new ClaimsIdentity(scheme.Name);
                ident.AddClaim(claim);
                result = AuthenticateResult.Success(
                    new AuthenticationTicket(new ClaimsPrincipal(ident),
                        scheme.Name));
            } else {
                result = AuthenticateResult.NoResult();
            }
            return Task.FromResult(result);
        }

        public Task ChallengeAsync(AuthenticationProperties properties) {
            context.Response.StatusCode = StatusCodes.Status401Unauthorized;
            return Task.CompletedTask;
        }

        public Task ForbidAsync(AuthenticationProperties properties) {
            context.Response.StatusCode = StatusCodes.Status403Forbidden;
            return Task.CompletedTask;
        }
    }
}
```

The `InitializeAsync` method is called to prepare the handler to authenticate a request, providing it with the name of the authentication scheme and the `HttpContext` object that provides access to the request and response objects. The scheme can be used by classes that provide multiple authentication techniques but is used just to get the name given to the authentication handler when authorization is configured, as demonstrated shortly.

The `AuthenticateAsync` method is called to authenticate a request, which it does by returning an `AuthenticateResult` object. The `AuthenticateResult` class defines static methods that produce objects to represent different authentication outcomes, as described in Table 14-6.

■ **Note** The methods defined by the `IAuthenticationHandler` interface are asynchronous. I don't need to perform asynchronous operations for the simple authentication in Listing 14-16, which is why the method results are produced with either `Task.FromResult` or `Task.CompletedTask`.

Table 14-6. *The Static Methods Defined by the AuthenticateResult Class*

Name	Description
`Success(ticket)`	This method creates a result indicating that authentication succeeded.
`Fail(message)`	This method creates a result indicating that authentication failed.
`NoResult()`	This method creates a result indicating no authentication was performed for this request. For the example application, this means that a request did not contain a cookie named `authUser`.

The argument to the `Success` method is an `AuthenticationTicket` object that describes the authenticated user and the name of the authentication scheme that was used:

```
...
result = AuthenticateResult.Success(new AuthenticationTicket(
    new ClaimsPrincipal(ident), scheme.Name));
...
```

Notice that no code in Listing 14-16 compares the user's claims with the policy defined with the `Authorize` attribute. When authorization fails, the built-in ASP.NET Core middleware will automatically invoke the `ChallengeAsync` or `ForbidAsync` method.

Configuring the Request Pipeline

In Listing 14-17, I have changed the configuration of the request pipeline to remove the custom authentication and authorization middleware I created earlier and to enable the built-in middleware that ASP.NET Core provides.

Listing 14-17. Configuring the Pipeline in the Startup.cs File in the ExampleApp Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ExampleApp.Custom;

namespace ExampleApp {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddAuthentication(opts => {
                opts.AddScheme<AuthHandler>("qsv", "QueryStringValue");
                opts.DefaultScheme = "qsv";
            });
            services.AddAuthorization();
        }
```

```
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

            //app.UseMiddleware<CustomAuthentication>();
            app.UseAuthentication();
            app.UseMiddleware<RoleMemberships>();
            app.UseRouting();

            app.UseMiddleware<ClaimsReporter>();
            //app.UseMiddleware<CustomAuthorization>();
            app.UseAuthorization();

            app.UseEndpoints(endpoints => {
                endpoints.MapGet("/", async context => {
                    await context.Response.WriteAsync("Hello World!");
                });
                endpoints.MapGet("/secret", SecretEndpoint.Endpoint)
                    .WithDisplayName("secret");
                endpoints.Map("/signin", CustomSignInAndSignOut.SignIn);
                endpoints.Map("/signout", CustomSignInAndSignOut.SignOut);

            });
        }
    }
}
```

Two sets of services are enabled in the `ConfigureServices` method using the `AddAuthentication` and `AddAuthorization` methods. The `AddAuthentication` method uses the options pattern to register the implementation of the `IAuthenticationHandler` interface and use it as the default.

```
...
services.AddAuthentication(opts => {
    opts.AddScheme<AuthHandler>("qsv", "QueryStringValue");
    opts.DefaultScheme = "qsv";
});
...
```

The `AddScheme` method defines a generic type parameter that specifies the `IAuthenticationHandler` implementation class and regular arguments that specify a name for the authentication scheme and a display name for the scheme, which I have set to `qsv` and `QueryStringValue`. (Names for authentication schemes are more important when you are using more than one in the same application.) The `DefaultScheme` property is used to configure the default authentication scheme, and I have specified the name given to the scheme when it was registered with the `AddScheme` method.

In addition to the services, two method calls are required to configure the built-in middleware. The `UseAuthentication` method authenticates requests using the scheme configured in the `ConfigureServices` method, and the `UseAuthorization` method enforces access control using the `Authorize` attribute.

Restart ASP.NET Core, request `http://localhost:5000/signin?user=alice`, and then request `http://localhost:5000/secret` to test the new code. There is no visual change, as Figure 14-9 shows, but the authorization is now expressed using the standard attribute and applied using the built-in middleware.

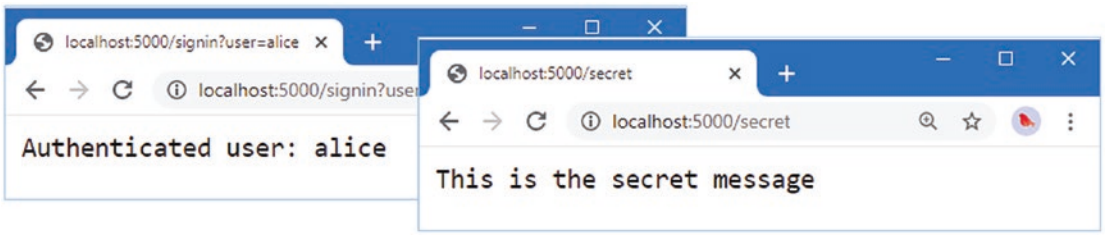

Figure 14-9. *Using the built-in middleware*

There is no change in the objects used to represent the user and their claims. This means that I can leave my `RoleMemberships` middleware in the pipeline to supply the application with role claims. The `ClaimsReporter` middleware is also still in the pipeline and will produce the following output when requesting the protected endpoint after signing in as Alice:

```
User: alice
Authenticated: True
Authentication Type qsv
Identities: 2
Auth type: qsv, 1 claims
Type: name, Value: alice, Issuer: LOCAL AUTHORITY
Auth type: Roles, 3 claims
Type: name, Value: alice, Issuer: LOCAL AUTHORITY
Type: role, Value: User, Issuer: LOCAL AUTHORITY
Type: role, Value: Administrator, Issuer: LOCAL AUTHORITY
```

Moving the Sign-In and Sign-Out Code

One drawback of my original approach is that the endpoints that deal with user requests to sign in and out are part of the authentication implementation, which makes it difficult to use a different authentication scheme. To address this problem, the ASP.NET Core authentication middleware supports the `IAuthenticationSignInHandler` interface, which extends the `IAuthenticationHandler` interface with methods for signing users into the application. Listing 14-18 updates the handler to implement the new interface and consolidate all the authentication code into a single class.

Listing 14-18. Consolidating Code in the AuthHandler.cs File in the Custom Folder

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Http;
using System.Security.Claims;
using System.Threading.Tasks;

namespace ExampleApp.Custom {
    public class AuthHandler : IAuthenticationSignInHandler {
        private HttpContext context;
        private AuthenticationScheme scheme;
```

```
        public Task InitializeAsync(AuthenticationScheme authScheme,
                HttpContext httpContext) {
            context = httpContext;
            scheme = authScheme;
            return Task.CompletedTask;
        }

        // ...other methods omitted for brevity...

        public Task SignInAsync(ClaimsPrincipal user,
                AuthenticationProperties properties) {
            context.Response.Cookies.Append("authUser", user.Identity.Name);
            return Task.CompletedTask;
        }

        public Task SignOutAsync(AuthenticationProperties properties) {
            context.Response.Cookies.Delete("authUser");
            return Task.CompletedTask;
        }
    }
}
```

The SignInAsync and SignOutAsync methods add and remove the cookie that the AuthenticateAsync method uses to authenticate requests. Consolidating the authentication code means that the endpoints no longer work directly with the cookie and can ask ASP.NET Core to sign the user in or out using extension methods for the HttpContext class, as shown in Listing 14-19.

Listing 14-19. Signing In and Out in the CustomSignInAndSignOut.cs File in the Custom Folder

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;
using System.Security.Claims;

namespace ExampleApp.Custom {
    public class CustomSignInAndSignOut {

        public static async Task SignIn(HttpContext context) {
            string user = context.Request.Query["user"];
            if (user != null) {
                Claim claim = new Claim(ClaimTypes.Name, user);
                ClaimsIdentity ident = new ClaimsIdentity("qsv");
                ident.AddClaim(claim);
                await context.SignInAsync(new ClaimsPrincipal(ident));
                await context.Response
                    .WriteAsync($"Authenticated user: {user}");
            } else {
                await context.ChallengeAsync();
            }
        }
```

```
        public static async Task SignOut(HttpContext context) {
            await context.SignOutAsync();
            await context.Response.WriteAsync("Signed out");
        }
    }
}
```

ASP.NET Core provides a set of extension methods that provide indirect access to the signing and authentication features, as described in Table 14-7.

Table 14-7. *HttpContext Extension Methods for Accessing the Authentication Handler*

Name	Description
`AuthenticateAsync()` `AuthenticateAsync(scheme)`	This method authenticates a request. The default scheme will be used if no argument is provided.
`SignInAsync(principal)` `SignInAsync(principal, scheme)`	This method signs a user into the application. The default scheme is used if one is not specified.
`SignOutAsync()  SignOutAsync(scheme)`	This method signs a user out of the application. The default scheme is used if one is not specified.
`ChallengeAsync()` `ChallengeAsync(scheme)`	This method sends a challenge response. The default scheme is used if one is not specified.
`ForbidAsync()ForbidAsync(scheme)`	This method sends a forbidden response. The default scheme is used if one is not specified.

These methods use the authentication handlers set up in the `Startup` class without needing to know which handler will be used, allowing code to be written without dependencies on specific `IAuthenticationHandler` implementations. Using these methods means that the code in Listing 14-19 is responsible for processing the user's credentials—which is just a query string value in this example—without needing to worry about the token that will be used to authenticate subsequent requests.

Restart ASP.NET Core, request `http://localhost:5000/signin?user=alice`, and then request `http://localhost:5000/secret` to test the new code. There is no visual change, and you will see the results shown in Figure 14-9. However, the code that signs Alice into the application no longer deals directly with the cookie that is used to authenticate the request for the protected endpoint. The effect is that validation of the user's credentials has been separated from the scheme that creates the cookie and uses it to validate requests, as shown in Figure 14-10.

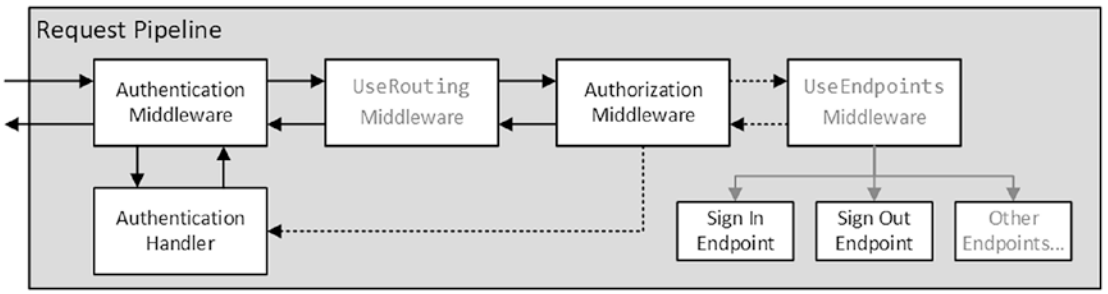

Figure 14-10. *Separating signing in from authentication*

Using HTML Responses

All the responses so far in this chapter have been plain text or just HTTP status codes, which has helped focus on the authentication and authorization processes but isn't what most web applications require. The next improvement to my example is to add an HTML response and provide a more user-friendly mechanism for signing in to and out of the application.

I am going to use Razor Pages for the HTML endpoints, although MVC controllers and views would work equally as well. Add a Razor Page named `SignIn.cshtml` to the `Pages` folder with the content shown in Listing 14-20.

Listing 14-20. The Contents of the SignIn.cshtml File in the Pages Folder

```
@page "{code:int?}"
@model ExampleApp.Pages.SignInModel
@using Microsoft.AspNetCore.Http

@if (Model.Code == StatusCodes.Status401Unauthorized) {
    <h3 class="bg-warning text-white text-center p-2">
        401 - Challenge Response
    </h3>
} else if (Model.Code == StatusCodes.Status403Forbidden) {
    <h3 class="bg-danger text-white text-center p-2">
        403 - Forbidden Response
    </h3>
}
<h4 class="bg-info text-white m-2 p-2">
    Current User: @Model.Username
</h4>

<div class="m-2">
    <form method="post">
        <div class="form-group">
            <label>User</label>
            <select class="form-control"
                    asp-for="Username" asp-items="@Model.Users">
            </select>
        </div>
        <button class="btn btn-info" type="submit">Sign In</button>
    </form>
</div>
```

This Razor Page will serve as the sign-in page as well as the challenge and forbidden responses. The user will be able to pick from a list of identities and sign in, which is a more convenient approach than the query strings used in earlier examples. (It is no more secure, however, but I'll show you how to deal with real credentials in later chapters, when I describe the sign-in features provided by ASP.NET Core Identity.) To define the page model for the Razor Page, add the code shown in Listing 14-21 to the `SignIn.cshtml.cs` file in the `Pages` folder. You will have to create this file if you are using Visual Studio Code.

Listing 14-21. The Contents of the SignIn.cshtml.cs File in the Pages Folder

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Security.Claims;
using System.Threading.Tasks;

namespace ExampleApp.Pages {
    public class SignInModel : PageModel {

        public SelectList Users => new SelectList(UsersAndClaims.Users,
            User.Identity.Name);

        public string Username { get; set; }

        public int? Code { get; set; }

        public void OnGet(int? code) {
            Code = code;
            Username = User.Identity.Name ?? "(No Signed In User)";
        }

        public async Task<ActionResult> OnPost(string username) {
            Claim claim = new Claim(ClaimTypes.Name, username);
            ClaimsIdentity ident = new ClaimsIdentity("simpleform");
            ident.AddClaim(claim);
            await HttpContext.SignInAsync(new ClaimsPrincipal(ident));
            return Redirect("/signin");
        }
    }
}
```

To let users sign out of the application, add a Razor Page named SignOut.cshtml to the Pages folder with the content shown in Listing 14-22.

Listing 14-22. The Contents of the SignOut.cshtml File in the Pages Folder

```
@page
@model ExampleApp.Pages.SignOutModel

<h4 class="bg-info text-white m-2 p-2">
    Current User: @Model.Username
</h4>
<div class="m-2">
    <form method="post" >
        <button class="btn btn-info" type="submit">Sign Out</button>
    </form>
</div>
```

When the user submits the form, they will be signed out of the application, which will remove the cookie used for request authentication. To define the page model for the Razor Page, add the code shown in Listing 14-23 to the SignOut.cshtml.cs file in the Pages folder. You will have to create this file if you are using Visual Studio Code.

Listing 14-23. The Contents of the SignOut.cshtml.cs File in the Pages Folder

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using System.Threading.Tasks;

namespace ExampleApp.Pages {
    public class SignOutModel : PageModel {
        public string Username { get; set; }

        public void OnGet() {
            Username = User.Identity.Name ?? "(No Signed In User)";
        }

        public async Task<ActionResult> OnPost() {
            await HttpContext.SignOutAsync();
            return RedirectToPage("SignIn");
        }
    }
}
```

When the page receives an HTTP POST request, the user is signed out through the SignoutAsync extension method, and the browser is redirected to the SignIn page.

In Listing 14-24, I have updated the authentication handler to redirect the user to the SignIn page instead of sending HTTP status codes for the challenge and forbidden responses.

Listing 14-24. Changing Responses in the AuthHandler.cs File in the Custom Folder

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Http;
using System.Security.Claims;
using System.Threading.Tasks;

namespace ExampleApp.Custom {
    public class AuthHandler : IAuthenticationSignInHandler {
        private HttpContext context;
        private AuthenticationScheme scheme;

        // ...other methods omitted for brevity...

        public Task ChallengeAsync(AuthenticationProperties properties) {
            //context.Response.StatusCode = StatusCodes.Status401Unauthorized;
            context.Response.Redirect("/signin/401");
            return Task.CompletedTask;
        }
```

```
        public Task ForbidAsync(AuthenticationProperties properties) {
            //context.Response.StatusCode = StatusCodes.Status403Forbidden;
            context.Response.Redirect("/signin/403");
            return Task.CompletedTask;
        }

        // ...other methods omitted for brevity...
    }
}
```

In Listing 14-25, I have enabled the services and middleware required for Razor Pages and serve the static CSS file from the Bootstrap package.

Listing 14-25. Configuring Razor Pages in the Startup.cs File in the ExampleApp Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ExampleApp.Custom;

namespace ExampleApp {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddAuthentication(opts => {
                opts.AddScheme<AuthHandler>("qsv", "QueryStringValue");
                opts.DefaultScheme = "qsv";
            });
            services.AddAuthorization();
            services.AddRazorPages();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

            app.UseStaticFiles();
            //app.UseMiddleware<CustomAuthentication>();
            app.UseAuthentication();
            app.UseMiddleware<RoleMemberships>();
            app.UseRouting();

            app.UseMiddleware<ClaimsReporter>();
            //app.UseMiddleware<CustomAuthorization>();
            app.UseAuthorization();

            app.UseEndpoints(endpoints => {
                endpoints.MapGet("/", async context => {
                    await context.Response.WriteAsync("Hello World!");
                });
```

```
                endpoints.MapGet("/secret", SecretEndpoint.Endpoint)
                    .WithDisplayName("secret");
                //endpoints.Map("/signin", CustomSignInAndSignOut.SignIn);
                //endpoints.Map("/signout", CustomSignInAndSignOut.SignOut);
                endpoints.MapRazorPages();
            });
        }
    }
}
```

Restart ASP.NET Core, request `http://localhost:5000/signin`, and sign in as Alice. (You may find that you are already authenticated with a cookie from an earlier example, but this will be replaced as soon as you sign in again.)

Request `http://localhost:5000/secret`, and you will be granted access to the protected endpoint. Repeat the process as Bob or without signing into the application, and the browser will be redirected to the `SignIn` page, as shown in Figure 14-11.

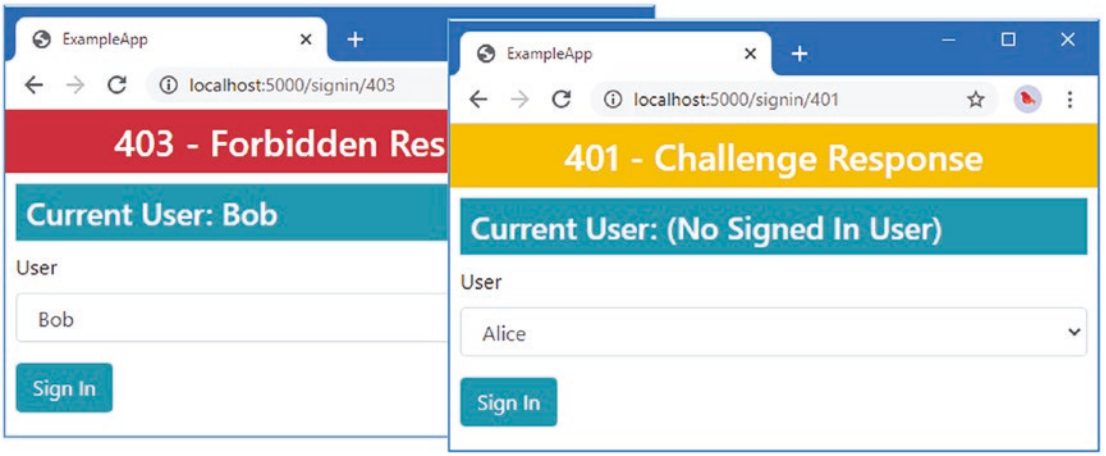

Figure 14-11. *Generating HTML responses*

Using the Built-In Cookie Authentication Handler

Creating a custom authentication handler is useful for this chapter, but a key goal with dealing with access control is to write as little custom code as possible. One consequence of implementing the `IAuthenticationSignInHandler` interface is that the code that validates user credentials is completely separate from the code that signs users into the application and authenticates requests. And so, to finish this chapter, I am going to replace my custom cookie-based authentication handler with the one that Microsoft provides with ASP.NET Core. Not only has the Microsoft handler been thoroughly tested, but it also manages cookies better and is more flexible and configurable. In Listing 14-26, I have changed the configuration of the application's services to use the built-in cookie authentication handler.

Listing 14-26. Using a Built-In Authentication Handler in the Startup.cs File in the ExampleApp Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ExampleApp.Custom;
using Microsoft.AspNetCore.Authentication.Cookies;

namespace ExampleApp {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddAuthentication(opts => {
                opts.DefaultScheme
                    = CookieAuthenticationDefaults.AuthenticationScheme;
            }).AddCookie(opts => {
                opts.LoginPath = "/signin";
                opts.AccessDeniedPath = "/signin/403";
            });
            services.AddAuthorization();
            services.AddRazorPages();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            // ...statements omitted for brevity...
        }
    }
}
```

I have set the `DefaultScheme` option to the `AuthenticationScheme` constant defined by the `CookieAuthenticationDefaults` class, which provides the name of the cookie handler. The `AddCookie` extension method sets up the built-in cookie authentication handler, which is configured using the options pattern, which is applied to the `CookieAuthenticationOptions` class.

■ **Note** One key benefit of using the built-in cookie authentication handler is that it serializes a user's claims into the encrypted cookie used for authentication or, if one is available, stores the claims in the ASP.NET Core session data store. Examples in later chapters, such as Chapter 20, use multiple instances of the cookie authentication handler to store cookies for different aspects of complex sign-in and authentication scenarios.

The `CookieAuthenticationOptions` class defines properties for managing the authentication cookie, the most useful of which are described in Table 14-8.

Table 14-8. *Useful CookieAuthenticationOptions Properties*

Name	Description
`AccessDeniedPath`	This property defines the URL path the client will be directed to for the forbid response. The default value is `/Account/AccessDenied`.
`LoginPath`	This property defines the URL path the client will be directed to for the challenge response. The default value is `/Account/Login`.
`ReturnUrlParameter`	This property defines the name of the query string parameter that will be used to store the path requested before a challenge response and can be used to redirect the client after a successful sign-in. The default value is `returnUrl`.
`ExpireTimeSpan`	This property defines the lifespan of the authentication cookie. The default value is 14 days.
`SlidingExpiration`	This property is used to control whether the authentication cookie is automatically reissued with a new expiry date. The default value is `true`.

For this chapter, I have set `LoginPath` and `AccessDeniedPath` to URLs that will be handled by the `SignIn` Razor Page.

Although I have not set the `ReturnUrlParameter` option in Listing 14-26, I can use the feature to make the login process smoother, as shown in Listing 14-27.

Listing 14-27. Supporting the Return URL in the SignIn.cshtml.cs File in the Pages Folder

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Security.Claims;
using System.Threading.Tasks;

namespace ExampleApp.Pages {
    public class SignInModel : PageModel {

        public SelectList Users => new SelectList(UsersAndClaims.Users,
            User.Identity.Name);

        public string Username { get; set; }

        public int? Code { get; set; }

        public void OnGet(int? code) {
            Code = code;
            Username = User.Identity.Name ?? "(No Signed In User)";
        }

        public async Task<ActionResult> OnPost(string username,
                [FromQuery]string returnUrl) {
            Claim claim = new Claim(ClaimTypes.Name, username);
            ClaimsIdentity ident = new ClaimsIdentity("simpleform");
            ident.AddClaim(claim);
```

```
            await HttpContext.SignInAsync(new ClaimsPrincipal(ident));
            return Redirect(returnUrl ?? "/signin");
        }
    }
}
```

The new parameter for the POST handler method will receive the URL that the client requested before a challenge response is issued. To see the effect of the changes to the application, restart ASP.NET Core, request the `http://localhost:5000/signout` page, and click the Sign Out button to make sure no user is signed in.

Next, request `http://localhost:5000/secret`. You will be redirected to the `SignIn` Razor Page, but the URL will include a `returnUrl` query string parameter that contains the URL you requested. Authenticate as `Alice`, and you will be automatically redirected to the protected endpoint, as shown in Figure 14-12.

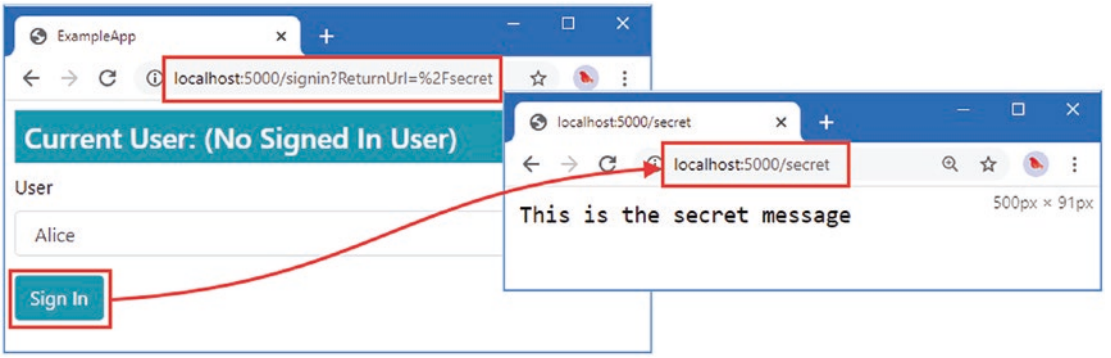

Figure 14-12. *Using the ASP.NET Core cookie authentication handler*

Summary

In this chapter, I explained the way that ASP.NET Core handles HTTP requests through its pipeline and how middleware components can provide authentication and authorization. I explained the different request flows, including the challenge and forbidden responses, and explained how signing into an application can generate a token that is used to authenticate subsequent requests. Once I established the basic features, I gradually refined my authentication and authorization process using built-in ASP.NET Core features, until only the code required to validate user credentials remained. In the next chapter, I explain how requests are authorized.