CHAPTER 15

Authorizing Requests

In the previous chapter, I focused on the ASP.NET Core request pipeline and the middleware components that authenticate requests and authorize access to endpoints. In this chapter, I dig deeper into the authorization features, explaining the different ways that access can be restricted. These are, once again, features that are provided by ASP.NET Core, and I describe them without using ASP.NET Core Identity. Table 15-1 puts the ASP.NET Core authorization features in context.

Table 15-1. *Putting Request Authorization in Context*

Question	Answer
What is it?	Request authorization is the process of restricting access to endpoints so they can only be accessed by selected users.
Why is it useful?	Request authorization is the complement to authorization and acts as the gatekeeper to the resources managed by the application.
How is it used?	Access control policies are defined and applied to endpoints. When a request is processed, the policy is evaluated to determine if the user associated with the request is entitled to access the endpoint.
Are there any pitfalls or limitations?	Policies are most effective when they are simple and easy to understand. The more complex a policy becomes, the more likely it is to allow unintended access.
Are there any alternatives?	Request authorization is a fundamental feature and is required in any application that contains any endpoint that should not be available to all users.

Table 15-2 summarizes the chapter.

Table 15-2. *Chapter Summary*

Problem	Solution	Listing
Define and enforce a custom authorization requirement	Implement the `IAuthorizationRequirement` and `IAuthorizationHandler` interfaces.	8, 9
Apply an authorization policy	Use the `AuthorizationOptions` options pattern to specify a policy described with the `AuthorizationPolicy` class.	10, 11
Use a built-in requirement	Use one of the built-in requirement classes.	12
Combine requirements	Arrange individual requirements in an array.	13
Restrict access to specified authentication schemes	Specify the schemes when creating the policy.	14, 15
Apply policies to endpoints	Use the `Authorize` attribute.	16, 18–22
Specify the default policy	Assign a policy to the `DefaultPolicy` configuration option.	17
Restrict access with multiple policies	Specify multiple policies when applying the `Authorize` attribute.	23
Create exceptions to policies	Use the `AllowAnonymous` attribute.	24
Apply policies to Razor Pages	Use page conventions.	25, 26
Apply policies to controllers	Use filters and the application model.	27–29

Preparing for This Chapter

This chapter uses the `ExampleApp` project created in Chapter 14. The sections that follow describe the preparations required for this chapter.

■ **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.

Creating an Authorization Reporter

The ASP.NET Core authorization features are easier to understand if you can see their effects without having to repeatedly sign in and make requests as individual users. In this chapter, I replace the standard ASP.NET Core authorization middleware with a component that tests the authorization policy for the target endpoint using the users and claims defined in the `UsersAndClaims` class.

In Listing 15-1, I have added a convenience method to the `UsersAndClaims` class that will produce a sequence of `ClaimsPrincipal` objects and defined an array of authentication scheme names.

Listing 15-1. Adding a Method in 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 string[] Schemes = new string[] { "TestScheme" };

        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);

        public static IEnumerable<ClaimsPrincipal> GetUsers() {
            foreach (string scheme in Schemes) {
                foreach (var kvp in Claims) {
                    ClaimsIdentity ident = new ClaimsIdentity(scheme);
                    ident.AddClaim(new Claim(ClaimTypes.Name, kvp.Key));
                    ident.AddClaims(kvp.Value);
                    yield return new ClaimsPrincipal(ident);
                }
            }
        }
    }
}
```

Add a class file named AuthorizationReporter.cs to the ExampleApp/Custom folder and use it to define the middleware component shown in Listing 15-2.

Listing 15-2. The Contents of the AuthorizationReporter.cs File in the Custom Folder

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Http;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Security.Claims;
using System.Threading.Tasks;

namespace ExampleApp.Custom {
    public class AuthorizationReporter {
        private string[] schemes = new string[] { "TestScheme" };
        private RequestDelegate next;
        private IAuthorizationPolicyProvider policyProvider;
        private IAuthorizationService authorizationService;
        public AuthorizationReporter(RequestDelegate requestDelegate,
                IAuthorizationPolicyProvider provider,
                IAuthorizationService service) {
            next = requestDelegate;
            policyProvider = provider;
            authorizationService = service;
        }

        public async Task Invoke(HttpContext context) {
            Endpoint ep = context.GetEndpoint();
            if (ep != null) {
                Dictionary<(string, string), bool> results
                    = new Dictionary<(string, string), bool>();
                bool allowAnon = ep.Metadata.GetMetadata<IAllowAnonymous>() != null;
                IEnumerable<IAuthorizeData> authData =
                    ep?.Metadata.GetOrderedMetadata<IAuthorizeData>()
                        ?? Array.Empty<IAuthorizeData>();
                AuthorizationPolicy policy = await
                    AuthorizationPolicy.CombineAsync(policyProvider, authData);
                foreach (ClaimsPrincipal cp in GetUsers()) {
                    results[(cp.Identity.Name ?? "(No User)",
                        cp.Identity.AuthenticationType)] =
                            allowAnon || policy == null
                                || await AuthorizeUser(cp, policy);
                }
                context.Items["authReport"] = results;
                await ep.RequestDelegate(context);
            } else {
                await next(context);
            }
        }

        private IEnumerable<ClaimsPrincipal> GetUsers() =>
            UsersAndClaims.GetUsers()
                .Concat(new[] { new ClaimsPrincipal(new ClaimsIdentity()) });

        private async Task<bool> AuthorizeUser(ClaimsPrincipal cp,
                AuthorizationPolicy policy) {
            return UserSchemeMatchesPolicySchemes(cp, policy)
                && (await authorizationService.AuthorizeAsync(cp, policy)).Succeeded;
        }

        private bool UserSchemeMatchesPolicySchemes(ClaimsPrincipal cp,
                AuthorizationPolicy policy) {
            return policy.AuthenticationSchemes?.Count() == 0 ||
                cp.Identities.Select(id => id.AuthenticationType)
                    .Any(auth => policy.AuthenticationSchemes
                        .Any(scheme => scheme == auth));
        }
    }
}
```

This component uses features that you are unlikely to need in real projects, but that are interesting anyway and help explain the authorization process. The authorization requirements for an endpoint are available through the `HttpContext.GetEndpoint().Metadata` property and are expressed using the `IAuthorizeData` interface. To get an endpoint's requirements, I use the `GetOrderedMetadata<IAuthorizeData>` method, like this:

```
...
ep?.Metadata.GetOrderedMetadata<IAuthorizeData>() ?? Array.Empty<IAuthorizeData>();
...
```

You will see where the `IAuthorizeData` objects are created later in this chapter, but for now, it is enough to know that each one represents a requirement that must be met before a request is authorized.

The collection of individual `IAuthorizeData` objects is converted into a combined authorization policy for the endpoint using the static `AuthorizationPolicy.CombineAsync` method, using the `IAuthorizationPolicyProvider` service on which the middleware component declares a constructor dependency.

```
...
AuthorizationPolicy policy = await
    AuthorizationPolicy.CombineAsync(policyProvider, authData);
...
```

The authorization policy can be tested using the `IAuthorizationService` service, like this:

```
...
await authorizationService.AuthorizeAsync(cp, policy);
...
```

The arguments to the `AuthorizeAsync` method are the `ClaimsPrincipal` that requires authorization and the `AuthorizationPolicy` object produced from the endpoint's middleware. After evaluating the authorization policy, the endpoint is invoked to produce a response.

```
...
await ep.RequestDelegate(context);
...
```

This means that the endpoint will receive the request even if no users would have normally been authorized and also means that the request will be passed directly to the endpoint, skipping over any other middleware in the request pipeline. For these reasons, you should only use the code in Listing 15-2 to understand how authorization works and not deploy it in real projects.

■ **Tip** The code in Listing 15-2 is based on the way that the built-in ASP.NET Core middleware authorizes requests. ASP.NET Core and ASP.NET Core Identity are open source, and exploring the source code is a good way of understanding how features are implemented.

Creating the Report View

To create a partial view that will display the authorization results, add a Razor View named _AuthorizationReport.cshtml to the ExampleApp/Pages/Shared folder with the content shown in Listing 15-3. (If you are using Visual Studio, create the file using the Razor View – Empty item template.)

Listing 15-3. The Contents of the _AuthorizationReport.cshtml File in the Pages/Shared Folder

```
@{
    var data = Context.Items["authReport"] as Dictionary<(string, string), bool>
        ?? new Dictionary<(string, string), bool>();
}

<div class="m-2 p-2 border bg-light">
    <h5 class="text-center">Authorization Summary</h5>
    <table class="table table-sm table-bordered table-striped">
        <thead><tr><th>User</th><th>Scheme</th><th>Result</th></tr></thead>
        <tbody>
            @if (data.Count() == 0) {
                <tr><td colspan="3" class="text-center">No Data</td></tr>
            }
            @foreach (var result in data) {
                <tr>
                    <td>@result.Key.Item1</td>
                    <td>@result.Key.Item2</td>
                    <td class="text-white @(result.Value
                            ? "bg-success" : "bg-danger")">
                        @(result.Value ? "Access Granted" : "Access Denied")
                    </td>
                </tr>
            }
        </tbody>
    </table>
</div>
```

The partial view retrieves the authorization data from the HttpContext object and uses it to display a table containing the results of the tests performed by the middleware component.

Add the element shown in Listing 15-4 to the _Layout.cshtml File in the Pages/Shared folder to include the partial view.

Listing 15-4. Including a Partial View in the _Layout.cshtml File in the Pages/Shared Folder

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>ExampleApp</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
```

```
<body>
    <div>
        @RenderBody()
    </div>
    <partial name="_AuthorizationReport" />
</body>
</html>
```

Creating an Endpoint

I need an HTML endpoint so that I can include the partial view in the response that it generates. Add a Razor Page named `Secret.cshtml` to the `Pages` folder with the content shown in Listing 15-5.

Listing 15-5. The Contents of the Secret.cshtml File in the Pages Folder

```
@page

<h4 class="bg-info text-center text-white m-2 p-2">
    This is the secret message
</h4>
```

The Razor Page displays a simple HTML message, echoing the endpoint used in Chapter 14.

Configuring the Request Pipeline

In Listing 15-6, I have added the new middleware to the request pipeline, using it to replace the built-authorization in middleware that ASP.NET Core provides. I also enabled the MVC Framework and removed the text-only endpoint used in Chapter 14 so that the `/secret` URL will be handled by the Razor Page created in Listing 15-5.

Listing 15-6. Configuring the Application in the Startup.cs File in the ExampleApp Folder

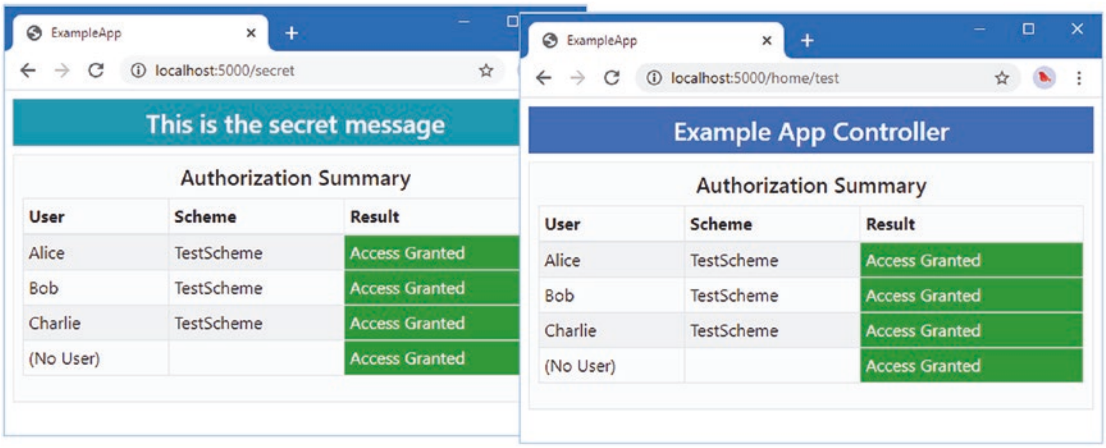
```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ExampleApp.Custom;
using Microsoft.AspNetCore.Authentication.Cookies;

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();
            services.AddControllersWithViews();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

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

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

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

Use a command prompt to run the command shown in Listing 15-7 in the ExampleApp folder.

Listing 15-7. Starting the Example Application

```
dotnet run
```

Once ASP.NET Core has started, use a browser to request `http://localhost:5000/secret` and `http://localhost:5000/home/test`, and you will receive the responses shown in Figure 15-1.



This is the secret message

Authorization Summary

User	Scheme	Result
Alice	TestScheme	Access Granted
Bob	TestScheme	Access Granted
Charlie	TestScheme	Access Granted
(No User)		Access Granted

Example App Controller

Authorization Summary

User	Scheme	Result
Alice	TestScheme	Access Granted
Bob	TestScheme	Access Granted
Charlie	TestScheme	Access Granted
(No User)		Access Granted

Figure 15-1. *Running the example application*

No authorization policy has been specified for the Razor Page or the `Test` action method, so the summary shows that all users and unauthenticated requests will be granted access.

Understanding Policies and Requirements

The key building blocks for ASP.NET Core authorization are *policies*, which consist of individual *requirements*. For a request to be authorized, it must meet all the requirements in the policy for the target endpoint. In this section, I create and apply a custom requirement and show you how it is applied before describing the built-in requirements that ASP.NET Core provides.

Defining the Custom Requirement and Handler

Requirements are expressed using classes that implement the `IAuthorizationRequirement` interface. The `IAuthorizationRequirement` interface defines no members and is implemented to describe the constraints for a specific requirement. To demonstrate, add a class file named `CustomRequirement.cs` to the `Custom` folder and use it to define the class shown in Listing 15-8.

Listing 15-8. The Contents of the CustomRequirement.cs File in the Custom Folder

```
using Microsoft.AspNetCore.Authorization;

namespace ExampleApp.Custom {

    public class CustomRequirement: IAuthorizationRequirement {

        public string Name { get; set; }

    }
}
```

This class implements the IAuthorizationRequirement interface and defines a Name property that will allow a username to be specified. To enforce the requirement, an implementation of the IAuthorizationHandler interface is required. Add a class file named CustomRequirementsHandler.cs to the Custom folder and add the code shown in Listing 15-9.

Listing 15-9. The Contents of the CustomRequirementsHandler.cs File in the Custom Folder

```
using Microsoft.AspNetCore.Authorization;
using System;
using System.Linq;
using System.Threading.Tasks;

namespace ExampleApp.Custom {
    public class CustomRequirementHandler : IAuthorizationHandler {

        public Task HandleAsync(AuthorizationHandlerContext context) {
            foreach (CustomRequirement req in
                context.PendingRequirements.OfType<CustomRequirement>().ToList()) {
                if (context.User.Identities.Any(ident => string.Equals(ident.Name,
                        req.Name, StringComparison.OrdinalIgnoreCase))) {
                    context.Succeed(req);
                }
            }
            return Task.CompletedTask;
        }
    }
}
```

The IAuthorizationHandler interface defines the HandleAsync method, which accepts an AuthorizationHandlerContext context object. AuthorizationHandlerContext defines the members described in Table 15-3.

Table 15-3. *The Members Defined by the AuthorizationHandlerContext Class*

Name	Description
User	This property returns the ClaimsPrincipal object for the request that requires authorization.
Resource	This property returns the target of the request, which will be an endpoint for the examples in this chapter.
Requirements	This property returns a sequence of all the requirements for the resource/endpoint.
PendingRequirements	This property returns a sequence of the requirements that have not been marked as satisfied.
Succeed(requirement)	This method tells ASP.NET Core that the specified requirement has been satisfied.
Fail()	This method tells ASP.NET Core that a requirement has not been satisfied.

The idea is that an authorization handler will process one or more of the requirements in the policy and, assuming the requirement is satisfied, mark them as succeeded. A request will be authorized if all the requirements succeed. Authorization fails if any handler calls the `Fail` method. Authorization will also fail if there are outstanding requirements for which a handler has not invoked the `Succeed` method.

The custom handler in Listing 15-9 gets the list of pending requirements and filters them for the `CustomRequirement` type like this:

```
...
foreach (CustomRequirement req in
    context.PendingRequirements.OfType<CustomRequirement>().ToList()) {
...
```

The `OfType<T>` method is a LINQ extension method, and I use the `ToList` method to force evaluation of the LINQ query because calling the `Succeed` method will alter the `PendingRequirements` sequence, which causes an error if the sequence is used directly in a `foreach` loop. For each `CustomRequirement` object, I can read the value of the `Name` property and compare it to the names contained in the user's identities, calling the `Succeed` method if there is a match.

```
...
if (context.User.Identities.Any(ident =>
    string.Compare(ident.Name, req.Name, true) == 0)) {
        context.Succeed(req);
}
...
```

Creating and Applying the Policy

The next step is to use the custom requirement to create an authorization policy. Add a class named `AuthorizationPolicies.cs` to the `Custom` folder with the code shown in Listing 15-10.

Listing 15-10. The Contents of the AuthorizationPolicies.cs File in the Custom Folder

```
using Microsoft.AspNetCore.Authorization;
using System.Linq;

namespace ExampleApp.Custom {

    public static class AuthorizationPolicies {

        public static void AddPolicies(AuthorizationOptions opts) {
            opts.FallbackPolicy = new AuthorizationPolicy(
               new[] {
                   new CustomRequirement() { Name = "Bob" }
               }, Enumerable.Empty<string>());
        }
    }
}
```

Policies are created using the AuthorizationPolicy class, whose constructor accepts a sequence of requirements and a sequence of authentication scheme names. The policy will grant access when all the requirements are satisfied for users who have been authenticated using one of the specified schemes, which I describe later in this chapter. For the moment, I have used an empty array, which will not restrict the policy.

The AuthorizationOptions class is used to configure ASP.NET Core authorization and defines the members described in Table 15-4.

Table 15-4. *AuthorizationOptions Members*

Name	Description
DefaultPolicy	This property defines the policy that will be applied by default when authorization is required but no policy has been selected, such as when the Authorize attribute is applied with no arguments. Any authorized user will be granted access unless a new default policy is defined, as described later in this chapter.
FallbackPolicy	This property defines the policy that is applied when no other policy has been defined. There is no fallback policy by default, which means that all requests will be authorized when there is no explicitly defined policy.
InvokeHandlersAfterFailure	This property determines whether a single failed requirement prevents subsequent requirements from being evaluated. The default value is true, which means that all requirements are evaluated. A false value means that a failure short-circuits the requirement process.

AddPolicy(name, policy)
AddPolicy(name, builder)
	This method adds a new policy, either using an AuthorizationPolicy object or using a builder function.

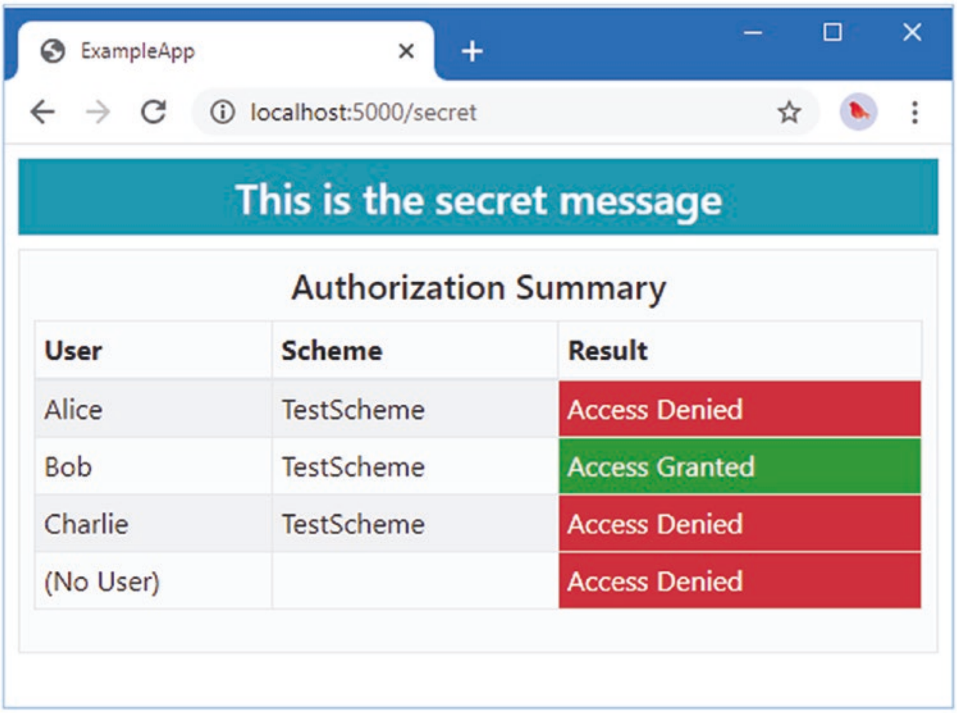
GetPolicy(name)	This method retrieves a policy using its name.

For this example, I have assigned my policy to the FallbackPolicy property, which means that it will be used to authorize requests for which no explicit authorization has been defined and which sets a baseline for the minimum authorization required for all requests the application receives. The final step is to apply the policy and register the custom requirements handler as a service, as shown in Listing 15-11.

Listing 15-11. Setting up the Policy 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;
using Microsoft.AspNetCore.Authorization;

namespace ExampleApp {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {

            services.AddTransient<IAuthorizationHandler, CustomRequirementHandler>();
            services.AddAuthentication(opts => {
                opts.DefaultScheme
                    = CookieAuthenticationDefaults.AuthenticationScheme;
            }).AddCookie(opts => {
                opts.LoginPath = "/signin";
                opts.AccessDeniedPath = "/signin/403";
            });
            services.AddAuthorization(opts => {
                AuthorizationPolicies.AddPolicies(opts);
            });
            services.AddRazorPages();
            services.AddControllersWithViews();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

            app.UseStaticFiles();
            app.UseAuthentication();
            app.UseRouting();
            app.UseMiddleware<AuthorizationReporter>();

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

Restart ASP.NET Core and request `http://localhost:5000/secret`. No specific authorization policy has been specified for the `Secret` Razor Page, so the fallback policy is applied, producing the result shown in Figure 15-2. The results show that the custom policy authorizes only authenticated requests for the username Bob.



This is the secret message

Authorization Summary

User	Scheme	Result
Alice	TestScheme	Access Denied
Bob	TestScheme	Access Granted
Charlie	TestScheme	Access Denied
(No User)		Access Denied

Figure 15-2. *Using an authorization policy*

Using the Built-In Requirements

There are a set of useful built-in requirements and handlers for the most common authorization requirements, as described in Table 15-5, and they can be used instead of custom classes.

Table 15-5. *Useful Built-In Authorization Requirement Classes*

Name	Description
`NameAuthorizationRequirement`	This requirement is for a case-sensitive name match.
`RolesAuthorizationRequirement`	This requirement is for a role.
`ClaimsAuthorizationRequirement`	This requirement is for a claim type and a range of acceptable values.
`AssertionRequirement`	This requirement evaluates a function using an `AuthorizationHandlerContext` object and is satisfied if the result is `true`.
`DenyAnonymousAuthorization`	This requirement is satisfied by any authenticated user.

Simple policies can be expressed using the name and role requirements. In Listing 15-12, I have replaced my custom requirement with the built-in equivalent.

Listing 15-12. Using a Built-In Requirement in the AuthorizationPolicies.cs in the Custom Folder

```
using Microsoft.AspNetCore.Authorization;
using System.Linq;
using Microsoft.AspNetCore.Authorization.Infrastructure;

namespace ExampleApp.Custom {

    public static class AuthorizationPolicies {

        public static void AddPolicies(AuthorizationOptions opts) {
            opts.FallbackPolicy = new AuthorizationPolicy(
               new IAuthorizationRequirement[] {
                   new NameAuthorizationRequirement("Bob"),
               }, Enumerable.Empty<string>());
        }
    }
}
```

This requirement has the same effect as my custom equivalent, albeit the name comparison is case-sensitive. Restart ASP.NET Core and request `http://localhost:5000/secret`, and you will see the response shown in Figure 15-2.

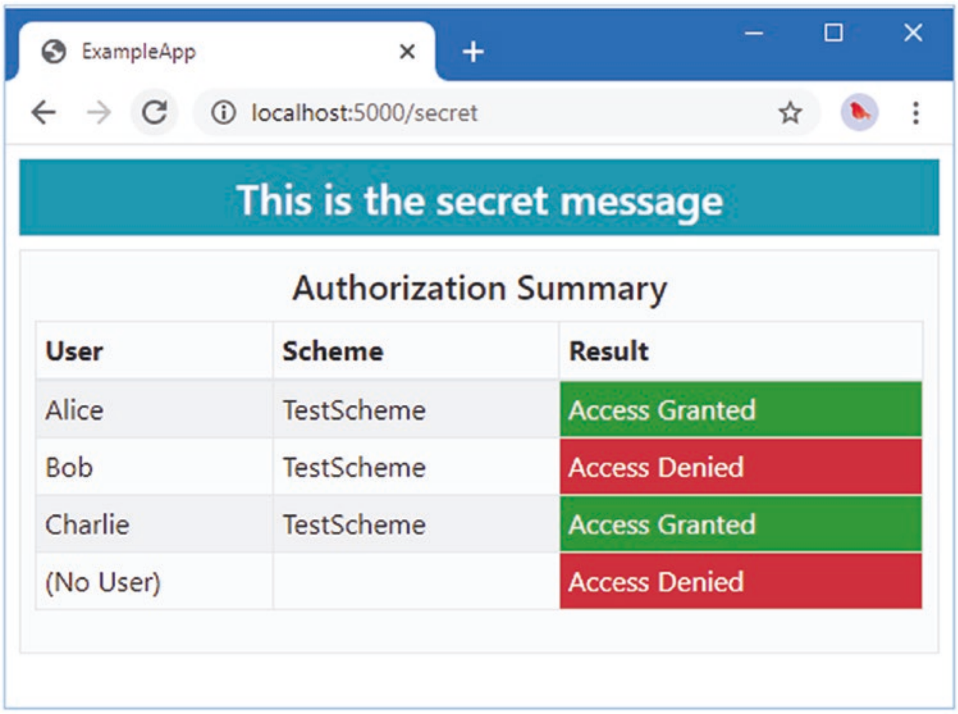
Combining Requirements

A policy will grant access only if all its requirements are satisfied, which means that complex policies can be built up using combinations of simple requirements, as shown in Listing 15-13.

Listing 15-13. Combining Requirements in the AuthorizationPolicies.cs File in the Custom Folder

```
using Microsoft.AspNetCore.Authorization;
using System.Linq;
using Microsoft.AspNetCore.Authorization.Infrastructure;

namespace ExampleApp.Custom {

    public static class AuthorizationPolicies {

        public static void AddPolicies(AuthorizationOptions opts) {
            opts.FallbackPolicy = new AuthorizationPolicy(
               new IAuthorizationRequirement[] {
                   new RolesAuthorizationRequirement(
                       new [] { "User", "Administrator" }),
                   new AssertionRequirement(context =>
                       !string.Equals(context.User.Identity.Name, "Bob"))
               }, Enumerable.Empty<string>());
        }
    }
}
```

This policy contains two requirements. The roles requirement will be met by users who are assigned the User or Administrator roles. The assertion requirement is met by any user whose name is not Bob. Restart ASP.NET Core and request http://localhost:500/secret, and you will see that access is granted to both Alice and Charlie, but not to Bob or unauthenticated requests, as shown in Figure 15-3.



This is the secret message

Authorization Summary

User	Scheme	Result
Alice	TestScheme	Access Granted
Bob	TestScheme	Access Denied
Charlie	TestScheme	Access Granted
(No User)		Access Denied

Figure 15-3. *Combining requirements in an authorization policy*

Restricting Access to a Specific Authorization Scheme

The two arguments used to create an AuthorizationPolicy object are a sequence of requirements and a sequence of authentication schemes. I used an empty array in earlier examples, which doesn't restrict the schemes that will be used, but this is a helpful feature in applications that support multiple authentication schemes and need to restrict access to users authenticated with a subset of them. In Listing 15-14, I added a new scheme that will be used by the custom authorization testing middleware.

Listing 15-14. Adding a Scheme in 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 string[] Schemes
            = new string[] { "TestScheme", "OtherScheme" };
        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);

        public static IEnumerable<ClaimsPrincipal> GetUsers() {
            foreach (string scheme in Schemes) {
                foreach (var kvp in Claims) {
                    ClaimsIdentity ident = new ClaimsIdentity(scheme);
                    ident.AddClaim(new Claim(ClaimTypes.Name, kvp.Key));
                    ident.AddClaims(kvp.Value);
                    yield return new ClaimsPrincipal(ident);
                }
            }
        }
    }
}
```

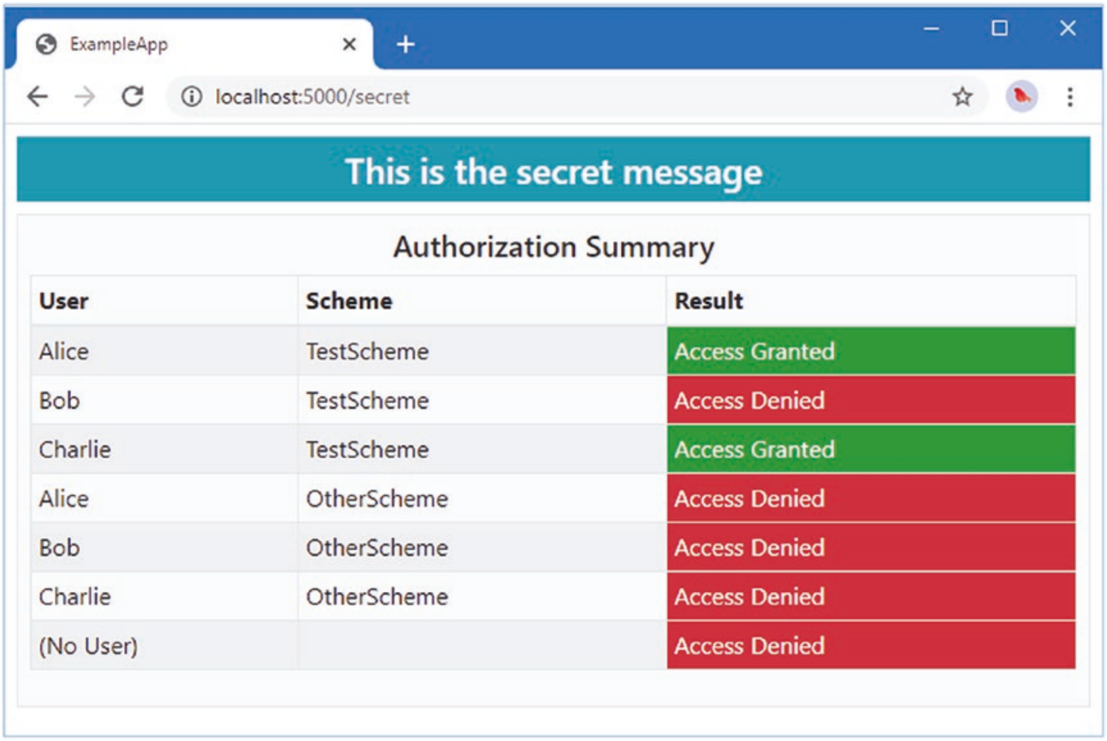
In Listing 15-15, I have changed the authorization policy to specify a scheme.

Listing 15-15. Specifying a Scheme in the AuthorizationPolicies.cs File in the Custom Folder

```
using Microsoft.AspNetCore.Authorization;
using System.Linq;
using Microsoft.AspNetCore.Authorization.Infrastructure;

namespace ExampleApp.Custom {

    public static class AuthorizationPolicies {

        public static void AddPolicies(AuthorizationOptions opts) {
            opts.FallbackPolicy = new AuthorizationPolicy(
               new IAuthorizationRequirement[] {
                   new RolesAuthorizationRequirement(
                       new [] { "User", "Administrator" }),
                   new AssertionRequirement(context =>
                       !string.Equals(context.User.Identity.Name, "Bob"))
               }, new string[] { "TestScheme" });
        }
    }
}
```

The effect is that the policy requires all its requirements to be met and for the request to have been authenticated using the `TestScheme` scheme. Restart ASP.NET Core and request `http://localhost:5000/secret`, and you will see that authorization fails for the `ClaimsPrincipal` objects whose authentication scheme is `OtherScheme`, even when their claims meet all the policy requirements, as shown in Figure 15-4. Alice and Charlie are authorized when they have been authenticated by `TestScheme` but not when they have been authenticated by `OtherScheme`.

■ **Note** In a real application, the built-in authorization middleware will trigger the standard authorization request flow described in Chapter 14. If the user has been authenticated using one of the policy's schemes, then a forbidden response is sent; otherwise, the challenge response is sent.



This is the secret message

Authorization Summary

User	Scheme	Result
Alice	TestScheme	Access Granted
Bob	TestScheme	Access Denied
Charlie	TestScheme	Access Granted
Alice	OtherScheme	Access Denied
Bob	OtherScheme	Access Denied
Charlie	OtherScheme	Access Denied
(No User)		Access Denied

Figure 15-4. *Specifying an authentication scheme*

Targeting Authorization Policies

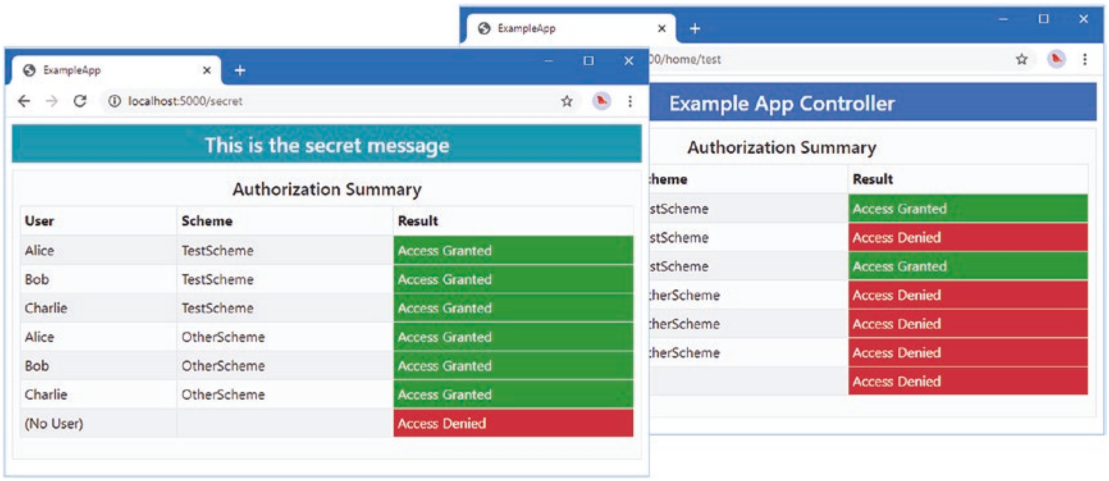
Changing the fallback policy makes it easy to see how the authorization building blocks fit together, but it has limited use in real projects where targeted access controls are required.

The next step up in authorization granularity is to use the default authorization policy, which is applied where access control is applied without using a specific policy. In Listing 15-16, I have applied the `Authorize` attribute to the `Secret` Razor Page, which tells ASP.NET Core that access controls are required.

Listing 15-16. Applying an Attribute in the Secret.cshtml File in the Pages Folder

```
@page

@using Microsoft.AspNetCore.Authorization

@attribute [Authorize]

<h4 class="bg-info text-center text-white m-2 p-2">
    This is the secret message
</h4>
```

The `Authorize` attribute denotes that authorization is required. I explain how to configure the `Authorize` attribute shortly, but when it is applied without arguments, ASP.NET Core uses the default authorization policy, which applied the `DenyAnonymousAuthorization` requirement described in Table 15-5. To see the effect of the default policy, restart ASP.NET Core and request `http://localhost:5000/secret`, and you will see that all authenticated users are authorized. The fallback policy is still used when there is no other policy available, which you can see by requesting `http://localhost:5000/home/test`. This request targets the `Test` action on the `Home` controller, which has not been decorated with the attribute. Figure 15-5 shows both responses.



This is the secret message

Authorization Summary

User	Scheme	Result
Alice	TestScheme	Access Granted
Bob	TestScheme	Access Granted
Charlie	TestScheme	Access Granted
Alice	OtherScheme	Access Granted
Bob	OtherScheme	Access Granted
Charlie	OtherScheme	Access Granted
(No User)		Access Denied

Example App Controller

Authorization Summary

heme	Result
stScheme	Access Granted
stScheme	Access Denied
stScheme	Access Granted
herScheme	Access Denied
herScheme	Access Denied
herScheme	Access Denied
	Access Denied

Figure 15-5. *The default authorization policy*

Behind the scenes, the `Authorize` attribute implements the `IAuthorizeData` interface, which is used by the ASP.NET Core authorization middleware to discover the authorization requirements for an endpoint. I used the same approach for the custom middleware in Listing 15-2.

```
...
IEnumerable<IAuthorizeData> authData =
    ep?.Metadata.GetOrderedMetadata<IAuthorizeData>()
        ?? Array.Empty<IAuthorizeData>();
AuthorizationPolicy policy = await
    AuthorizationPolicy.CombineAsync(policyProvider, authData);
...
```

This is the link between the Authorize attribute you may have already used in projects and the authorization flow described in this chapter.

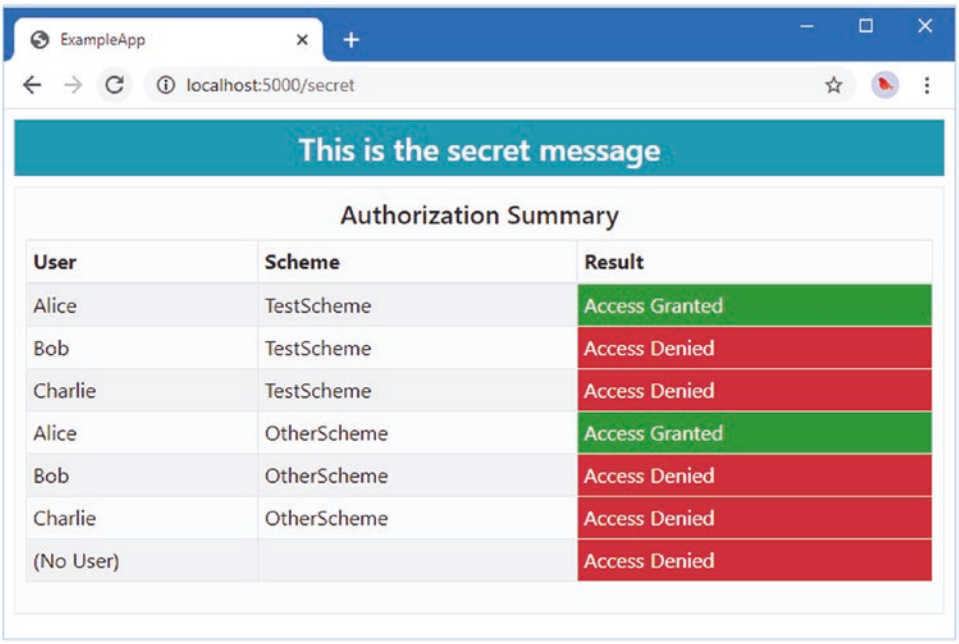
Changing the Default Authorization Policy

The default policy can be changed by assigning a policy to the AuthorizationOptions.DefaultPolicy, allowing different behavior to be defined for the Authorize attribute, as shown in Listing 15-17.

Listing 15-17. Changing the Default Policy in the AuthorizationPolicies.cs File in the Custom Folder

```
using Microsoft.AspNetCore.Authorization;
using System.Linq;
using Microsoft.AspNetCore.Authorization.Infrastructure;

namespace ExampleApp.Custom {

    public static class AuthorizationPolicies {

        public static void AddPolicies(AuthorizationOptions opts) {
            opts.FallbackPolicy = new AuthorizationPolicy(
               new IAuthorizationRequirement[] {
                   new RolesAuthorizationRequirement(
                       new [] { "User", "Administrator" }),
                   new AssertionRequirement(context =>
                       !string.Equals(context.User.Identity.Name, "Bob"))
               }, new string[] { "TestScheme" });

            opts.DefaultPolicy = new AuthorizationPolicy(
                new IAuthorizationRequirement[] {
                    new RolesAuthorizationRequirement(
                        new string[] { "Administrator"})
                }, Enumerable.Empty<string>());
        }
    }
}
```

The new policy requires the Administrator role without restriction on the scheme used to authenticate the user. Restart ASP.NET Core and request http://localhost:5000/secret to see the effect of the new default policy, which is shown in Figure 15-6.



This is the secret message

Authorization Summary

User	Scheme	Result
Alice	TestScheme	Access Granted
Bob	TestScheme	Access Denied
Charlie	TestScheme	Access Denied
Alice	OtherScheme	Access Granted
Bob	OtherScheme	Access Denied
Charlie	OtherScheme	Access Denied
(No User)		Access Denied

***Figure 15-6.** Changing the default authorization policy*

Configuring Targeted Authorization Polices

The policy applied by the `Authorize` attribute can be selected using the properties described in Table 15-6.

***Table 15-6.** The Authorize Attribute Properties*

Name	Description
`AuthenticationSchemes`	This property is used to specify a comma-separated list of allowed authentication schemes.
`Policy`	This property is used to specify a policy by name.
`Roles`	This property is used to specify a comma-separated list of allowed roles.

Direct support is provided for role-based authorization, which is the most commonly used approach. Roles that should be granted access are specified as a comma-separated list in a string, as shown in Listing 15-18.

■ **Caution** Role names are not validated because ASP.NET Core doesn't know what role claims are going to be created in advance. If you misspell a role name, you will find that you don't get the authorization policy you expected.

Listing 15-18. Specifying Roles in the Secret.cshtml File in the Pages Folder

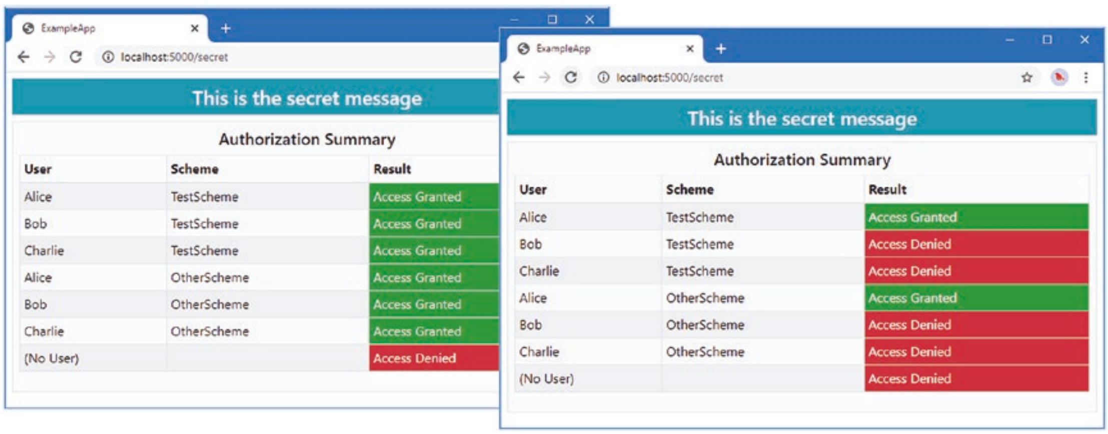
```
@page

@using Microsoft.AspNetCore.Authorization

@attribute [Authorize(Roles = "Administrator, User")]

<h4 class="bg-info text-center text-white m-2 p-2">
    This is the secret message
</h4>
```

A user who has a role claim for any of the roles specified by the `Authorize` attribute will be granted access. If you want to restrict access to users who have claims for all roles in a list, then multiple attributes can be used, as shown in Listing 15-19.

Listing 15-19. Using Multiple Attributes in the Secret.cshtml File in the Pages Folder

```
@page

@using Microsoft.AspNetCore.Authorization

@attribute [Authorize(Roles = "Administrator")]
@attribute [Authorize(Roles = "User")]

<h4 class="bg-info text-center text-white m-2 p-2">
    This is the secret message
</h4>
```

When the policy is created, it will contain a role requirement for each of the `Authorize` attributes, which means that only users who have both `Administrator` and `User` role claims will be granted access. Figure 15-7 shows the results of Listing 15-18 and Listing 15-19.



This is the secret message

Authorization Summary

User	Scheme	Result
Alice	TestScheme	Access Granted
Bob	TestScheme	Access Granted
Charlie	TestScheme	Access Granted
Alice	OtherScheme	Access Granted
Bob	OtherScheme	Access Granted
Charlie	OtherScheme	Access Granted
(No User)		Access Denied

This is the secret message

Authorization Summary

User	Scheme	Result
Alice	TestScheme	Access Granted
Bob	TestScheme	Access Denied
Charlie	TestScheme	Access Denied
Alice	OtherScheme	Access Granted
Bob	OtherScheme	Access Denied
Charlie	OtherScheme	Access Denied
(No User)		Access Denied

Figure 15-7. *Using the Authorize attribute to specify roles*

Using Named Policies

The Authorize attribute's Roles property is useful but makes it hard to change policies because every instance of the attribute has to be located so that new roles can be specified. The attribute's Policy attribute is more flexible because the authorization policy can be defined once and applied consistently throughout the application. When a change is required, the policy can be modified without needing to change the attributes. Listing 15-20 shows how a new policy is defined.

Listing 15-20. Defining a Policy in the AuthorizationPolicies.cs File in the Custom Folder

```
using Microsoft.AspNetCore.Authorization;
using System.Linq;
using Microsoft.AspNetCore.Authorization.Infrastructure;

namespace ExampleApp.Custom {

    public static class AuthorizationPolicies {

        public static void AddPolicies(AuthorizationOptions opts) {
            opts.FallbackPolicy = new AuthorizationPolicy(
               new IAuthorizationRequirement[] {
                   new RolesAuthorizationRequirement(
                       new [] { "User", "Administrator" }),
                   new AssertionRequirement(context =>
                       !string.Equals(context.User.Identity.Name, "Bob"))
               }, new string[] { "TestScheme" });

            opts.DefaultPolicy = new AuthorizationPolicy(
                new IAuthorizationRequirement[] {
                    new RolesAuthorizationRequirement(
                        new string[] { "Administrator"})
                }, Enumerable.Empty<string>());

            opts.AddPolicy("UsersExceptBob", new AuthorizationPolicy(
                new IAuthorizationRequirement[] {
                    new RolesAuthorizationRequirement(new[] { "User" }),
                    new AssertionRequirement(context =>
                        !string.Equals(context.User.Identity.Name, "Bob"))
                }, Enumerable.Empty<string>()));
        }
    }
}
```

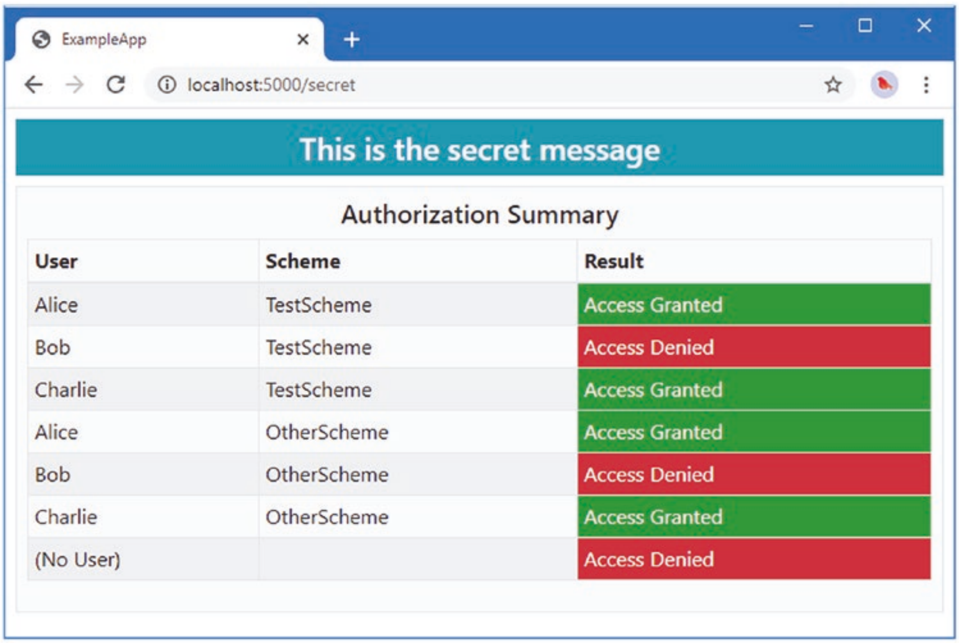
The AuthorizationOptions class provides the AddPolicy method, which accepts a name and an AuthorizationPolicy object. In this case, the name is UsersExceptBob, and the policy has a role requirement for User role and an assertion requirement that the username isn't Bob. In Listing 15-21, I have changed the Authorize attribute applied to the Secret Razor Page to use the Policy property.

Listing 15-21. Specifying a Property in the Secret.cshtml File in the Pages Folder

```
@page

@using Microsoft.AspNetCore.Authorization

@attribute [Authorize(Policy = "UsersExceptBob")]

<h4 class="bg-info text-center text-white m-2 p-2">
    This is the secret message
</h4>
```

Only a single policy can be specified, and, unlike the `Roles` property, the `Policy` property cannot be used with a comma-separated list. Restart ASP.NET Core and request `http://localhost:5000/secret` to see the effect of the new policy, as shown in Figure 15-8.

■ **Tip** The `AuthenticationSchemes` property on the `Authorize` attribute is used to specify one or more authentication schemes. These are added to the list defined by the policy, broadening the range of schemes that will be granted access.



This is the secret message

Authorization Summary

User	Scheme	Result
Alice	TestScheme	Access Granted
Bob	TestScheme	Access Denied
Charlie	TestScheme	Access Granted
Alice	OtherScheme	Access Granted
Bob	OtherScheme	Access Denied
Charlie	OtherScheme	Access Granted
(No User)		Access Denied

Figure 15-8. *Using a named authorization policy*

Creating Named Policies Using the Policy Builder

ASP.NET Core provides a more elegant way to create named policies, which allows requirements to be expressed using methods defined by the `AuthorizationPolicyBuilder` class, described in Table 15-7.

Table 15-7. *The AuthorizationPolicyBuilder Methods*

Name	Description
`AddAuthenticationSchemes(schemes)`	This method adds one or more authentication schemes to the set of schemes that will be accepted by the policy.
`RequireAssertion(func)`	This method adds an `AssertionRequirement` to the policy.
`RequireAuthenticatedUser()`	This method adds a `DenyAnonymousAuthorizationRequirement` to the policy, requiring requests to be authenticated.
`RequireClaim(type)`	This method adds a `ClaimsAuthorizationRequirement` to the policy, requiring a specific type of claim with any value.
`RequireClaim(type, values)`	This method adds a `ClaimsAuthorizationRequirement` to the policy, requiring a specific type of claim with one or more acceptable values.
`RequireRole(roles)`	This method adds a `RolesAuthorizationRequirement` to the policy.
`RequireUserName(name)`	This method adds a `NameAuthorizationRequirement` to the policy.
`AddRequirements(reqs)`	This method adds one or more `IAuthorizationRequirement` objects to the policy, which is useful for adding custom requirements.

The methods described in Table 15-7 all return a `AuthorizationPolicyBuilder` object, which allows calls to be chained together to create a policy, as shown in Listing 15-22.

Listing 15-22. Building an Authorization Policy in the AuthorizationPolicies.cs File in the Custom Folder

```
using Microsoft.AspNetCore.Authorization;
using System.Linq;
using Microsoft.AspNetCore.Authorization.Infrastructure;

namespace ExampleApp.Custom {

    public static class AuthorizationPolicies {

        public static void AddPolicies(AuthorizationOptions opts) {
            opts.FallbackPolicy = new AuthorizationPolicy(
               new IAuthorizationRequirement[] {
                   new RolesAuthorizationRequirement(
                       new [] { "User", "Administrator" }),
                   new AssertionRequirement(context =>
                       !string.Equals(context.User.Identity.Name, "Bob"))
               }, new string[] { "TestScheme" });

            opts.DefaultPolicy = new AuthorizationPolicy(
                new IAuthorizationRequirement[] {
                    new RolesAuthorizationRequirement(
                        new string[] { "Administrator"})
                }, Enumerable.Empty<string>());
            opts.AddPolicy("UsersExceptBob", builder => builder.RequireRole("User")
                .AddRequirements(new AssertionRequirement(context =>
                    !string.Equals(context.User.Identity.Name, "Bob")))
                .AddAuthenticationSchemes("OtherScheme"));
        }
    }
}
```

This policy has the same effect as the one defined in Listing 15-20 but has been created using the methods described in Table 15-7.

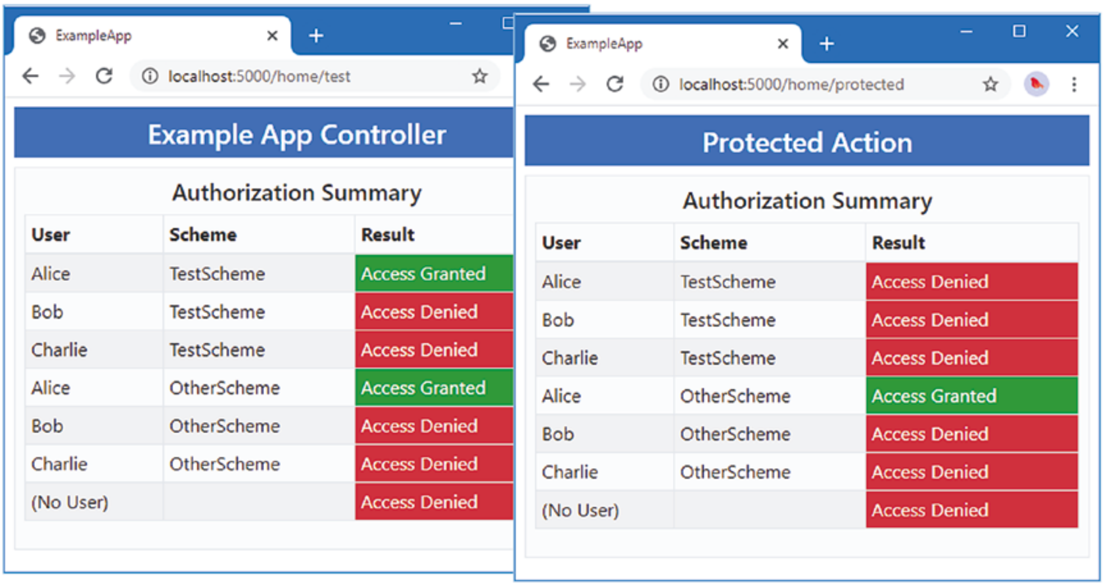
Combining Policies to Narrow Authorization

The Authorize attribute can be applied multiple times to set a wide authorization policy for an endpoint and narrow it for specific actions or handlers, as shown in Listing 15-23.

Listing 15-23. Applying the Attribute in the HomeController.cs File in the Controllers Folder

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

namespace ExampleApp.Controllers {

    [Authorize]
    public class HomeController: Controller {

        public IActionResult Test() => View();

        [Authorize(Roles = "User", AuthenticationSchemes = "OtherScheme")]
        public IActionResult Protected() => View("Test", "Protected Action");
    }
}
```

The Authorize attribute applied to the HomeController class applies the default policy to all the actions defined by the controller. I modified the default policy earlier, which means that access to the actions will only be granted to users with an Administrator role claim. This is the policy that will be applied to the Test action, but the Protected action has another Authorize attribute, which further narrows the policy by requiring the User role and the OtherScheme authentication scheme. To see the two policies that are created, restart ASP.NET Core and request http://localhost:5000/home/test and http://localhost:5000/home/protected. Access to the Test action is granted to Alice, regardless of how she is authenticated. Access to the Protected action is only granted to Alice when she is authenticated by the OtherScheme authentication scheme. Figure 15-9 shows both results.



Example App Controller

Authorization Summary

User	Scheme	Result
Alice	TestScheme	Access Granted
Bob	TestScheme	Access Denied
Charlie	TestScheme	Access Denied
Alice	OtherScheme	Access Granted
Bob	OtherScheme	Access Denied
Charlie	OtherScheme	Access Denied
(No User)		Access Denied

Protected Action

Authorization Summary

User	Scheme	Result
Alice	TestScheme	Access Denied
Bob	TestScheme	Access Denied
Charlie	TestScheme	Access Denied
Alice	OtherScheme	Access Granted
Bob	OtherScheme	Access Denied
Charlie	OtherScheme	Access Denied
(No User)		Access Denied

Figure 15-9. *Narrowing an authorization policy*

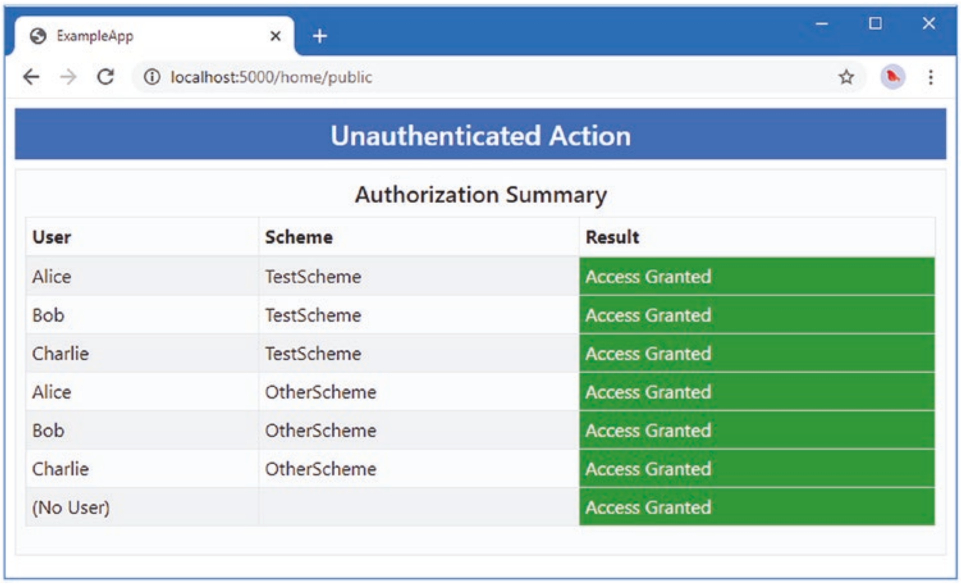
Creating Policy Exceptions

The AllowAnonymous attribute creates an exception to authorization policies to allow unauthenticated access. This is useful when the default or fallback policy would otherwise be applied or when the Authorize attribute has been used on a controller or Razor Page and you need to create an exception for a single action or handler method. In Listing 15-24, I have added an action method to the Home controller, to which the attribute is applied.

Listing 15-24. Adding an Action in the HomeController.cs File in the Controllers Folder

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

namespace ExampleApp.Controllers {

    [Authorize]
    public class HomeController: Controller {

        public IActionResult Test() => View();

        [Authorize(Roles = "User", AuthenticationSchemes = "OtherScheme")]
        public IActionResult Protected() => View("Test", "Protected Action");

        [AllowAnonymous]
        public IActionResult Public() => View("Test", "Unauthenticated Action");
    }
}
```

The AllowAnonymous attribute is given special treatment by the built-in authorization middleware and short-circuits the normal policy evaluation process to grant access to all requests, regardless of whether they are authenticated. Restart ASP.NET Core, and request http://localhost:5000/home/public to see the effect, which is shown in Figure 15-10.



Unauthenticated Action

Authorization Summary

User	Scheme	Result
Alice	TestScheme	Access Granted
Bob	TestScheme	Access Granted
Charlie	TestScheme	Access Granted
Alice	OtherScheme	Access Granted
Bob	OtherScheme	Access Granted
Charlie	OtherScheme	Access Granted
(No User)		Access Granted

Figure 15-10. *Allowing anonymous access*

PERFORMING AUTHORIZATION IN PAGES AND VIEWS

You can use authorization in views to alter the HTML content for different groups of users. This requires using dependency injection to receive an IAuthorizationService service and using the AuthorizeAsync method in the view to apply an authorization policy, like this:

```
@page
@using Microsoft.AspNetCore.Authorization
@inject IAuthorizationService AuthorizationService

<div>This is content for all users</div>

@if ((await AuthorizationService.AuthorizeAsync(User,
        "UsersExceptBob")).Succeeded) {
    <div>This is the protected content</div>
}
```

I found this technique awkward to use, and it is easy to misapply policies. But, if you do use this feature, then you must ensure that any action methods or handlers that are targeted by the protected content have the same level of authorization to prevent unauthorized users from crafting HTTP requests to perform restricted operations.

Applying Policies Using Razor Page Conventions

If you are using Razor Pages, you can apply authorization policies using the options pattern when configuring services in the `Startup` class. In Listing 15-25, I have defined a new policy that denies access to users who have an `Administrator` role claim.

Listing 15-25. Defining a Policy in the AuthorizationPolicies.cs File in the Custom Folder

```
using Microsoft.AspNetCore.Authorization;
using System.Linq;
using Microsoft.AspNetCore.Authorization.Infrastructure;

namespace ExampleApp.Custom {

    public static class AuthorizationPolicies {

        public static void AddPolicies(AuthorizationOptions opts) {

            opts.FallbackPolicy = new AuthorizationPolicy(
               new IAuthorizationRequirement[] {
                   new RolesAuthorizationRequirement(
                       new [] { "User", "Administrator" }),
                   new AssertionRequirement(context =>
                       !string.Equals(context.User.Identity.Name, "Bob"))
               }, new string[] { "TestScheme" });

            opts.DefaultPolicy = new AuthorizationPolicy(
                new IAuthorizationRequirement[] {
                    new RolesAuthorizationRequirement(
                        new string[] { "Administrator"})
                }, Enumerable.Empty<string>());

            opts.AddPolicy("UsersExceptBob", builder => builder.RequireRole("User")
                .AddRequirements(new AssertionRequirement(context =>
                    !string.Equals(context.User.Identity.Name, "Bob")))
                .AddAuthenticationSchemes("OtherScheme"));

            opts.AddPolicy("NotAdmins", builder =>
                builder.AddRequirements(new AssertionRequirement(context =>
                    !context.User.IsInRole("Administrator"))));
        }
    }
}
```

In Listing 15-26, I have used the Razor Pages conventions feature to apply the new policy to the `Secret` Razor Page.

Listing 15-26. Applying a Policy in the Startup.cs File in the ExampleApp Folder

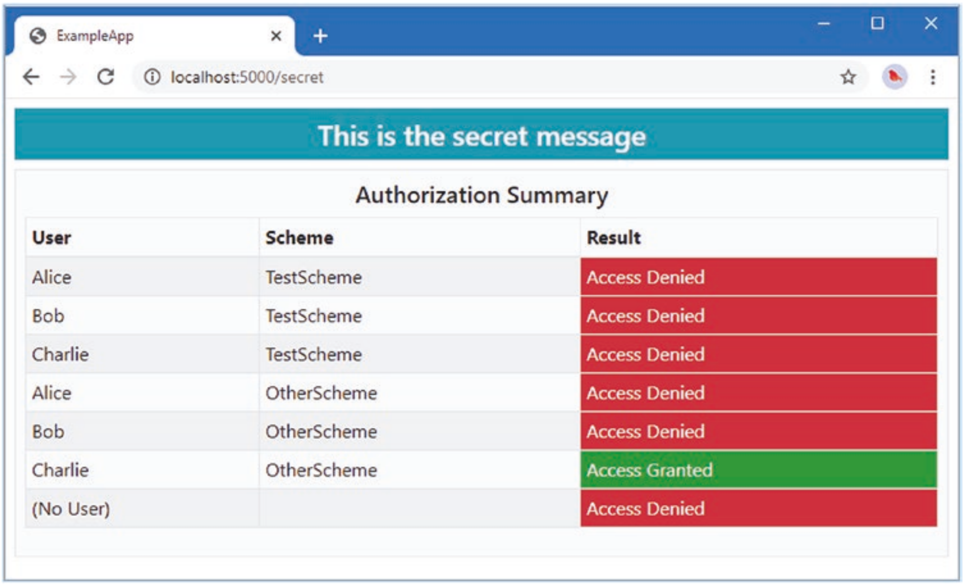
```
...
public void ConfigureServices(IServiceCollection services) {

    services.AddTransient<IAuthorizationHandler, CustomRequirementHandler>();

    services.AddAuthentication(opts => {
        opts.DefaultScheme
            = CookieAuthenticationDefaults.AuthenticationScheme;
    }).AddCookie(opts => {
        opts.LoginPath = "/signin";
        opts.AccessDeniedPath = "/signin/403";
    });
    services.AddAuthorization(opts => {
        AuthorizationPolicies.AddPolicies(opts);
    });
    services.AddRazorPages(opts => {
        opts.Conventions.AuthorizePage("/Secret", "NotAdmins");
    });
    services.AddControllersWithViews();
}
...
```

The RazorPagesOptions class is used to configure Razor Pages and its Conventions property returns a PageConventionCollection object for which authorization extension methods are available, as described in Table 15-8.

Table 15-8. *The Razor Pages Authorization Extension Methods*

Name	Description
AuthorizePage(page, policy)	This method applies an authorization policy to a specific page.
AuthorizePage(page)	This method applies the default policy to a specific page.
AuthorizeFolder(name, policy)	This method applies an authorization policy to all the pages in a single folder.
AuthorizeFolder(name)	This method applies the default policy to all the pages in a single folder.
AllowAnonymousToPage(page)	This method grants anonymous access to a specific page.
AllowAnonymousToFolder(name)	This method grants anonymous access to all the pages in a single folder.

In Listing 15-26, I used the AuthorizePage method to apply the NotAdmins policy to the Secret Razor Page. To see the effect, restart ASP.NET Core and request http://localhost:5000/secret. The response, shown in Figure 15-11, shows that the NotAdmins policy is combined with the policy UsersExceptBob policy applied to the page by the Authorize attribute, with the result that only Charlie can access the Razor Page and only when authenticated using OtherScheme.



This is the secret message

Authorization Summary

User	Scheme	Result
Alice	TestScheme	Access Denied
Bob	TestScheme	Access Denied
Charlie	TestScheme	Access Denied
Alice	OtherScheme	Access Denied
Bob	OtherScheme	Access Denied
Charlie	OtherScheme	Access Granted
(No User)		Access Denied

Figure 15-11. *Using an authorization convention*

Applying Policies Using MVC Framework Filters

The MVC Framework doesn't provide equivalent methods to those shown in Table 15-8, and the main alternative is to use a global filter to apply an authorization policy. The drawback with this approach is that the policy is applied to all requests that are processed by an MVC action method. However, with a little effort, it is possible to be more selective by taking advantage of the *application model* features provided to customize the core behavior of the MVC Framework, which is similar to the approach taken by the Razor Pages methods used in the previous section. In preparation for using the application model, I have disabled the customization of the default authorization policy, as shown in Listing 15-27, which will make it easier to see the effect of the new code in this section.

Listing 15-27. Disabling the Custom Default Policy in the AuthorizationPolicies.cs File in the Custom Folder

```
using Microsoft.AspNetCore.Authorization;
using System.Linq;
using Microsoft.AspNetCore.Authorization.Infrastructure;

namespace ExampleApp.Custom {

    public static class AuthorizationPolicies {

        public static void AddPolicies(AuthorizationOptions opts) {
            opts.FallbackPolicy = new AuthorizationPolicy(
               new IAuthorizationRequirement[] {
                   new RolesAuthorizationRequirement(
                       new [] { "User", "Administrator" }),
                   new AssertionRequirement(context =>
                       !string.Equals(context.User.Identity.Name, "Bob"))
               }, new string[] { "TestScheme" });
            //opts.DefaultPolicy = new AuthorizationPolicy(
            //    new IAuthorizationRequirement[] {
            //        new RolesAuthorizationRequirement(
            //            new string[] { "Administrator"})
            //    }, Enumerable.Empty<string>());

            opts.AddPolicy("UsersExceptBob", builder => builder.RequireRole("User")
                .AddRequirements(new AssertionRequirement(context =>
                    !string.Equals(context.User.Identity.Name, "Bob")))
                .AddAuthenticationSchemes("OtherScheme"));

            opts.AddPolicy("NotAdmins", builder =>
                builder.AddRequirements(new AssertionRequirement(context =>
                    !context.User.IsInRole("Administrator"))));
        }
    }
}
```

To create the code that will apply the authorization policy, add a class file named AuthorizationPolicyConvention.cs to the ExampleApp/Custom folder with the code shown in Listing 15-28.

Listing 15-28. The Contents of the AuthorizationPolicyConvention.cs File in the Custom Folder

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace ExampleApp.Custom {

    public class AuthorizationPolicyConvention : IActionModelConvention {
        private string controllerName;
        private string actionName;
        private IAuthorizeData attr = new AuthData();

        public AuthorizationPolicyConvention(string controller,
                string action = null, string policy = null,
                string roles = null, string schemes = null) {
            controllerName = controller;
            actionName = action;
            attr.Policy = policy;
            attr.Roles = roles;
            attr.AuthenticationSchemes = schemes;
        }

        public void Apply(ActionModel action) {
            if (controllerName == action.Controller.ControllerName
                    && (actionName == null || actionName == action.ActionName)) {
                foreach (var s in action.Selectors) {
                    s.EndpointMetadata.Add(attr);
                }
            }
        }
    }
    class AuthData : IAuthorizeData {
        public string AuthenticationSchemes { get; set; }
        public string Policy { get; set; }
        public string Roles { get; set; }
    }
}
```

This class implements the IActionModelConvention interface, which is the part of the application model feature that allows action methods to be altered. The constructor accepts a controller name and, optionally, the action to which the policy should be applied. There are optional parameters for the authorization policy, the set of acceptable roles, and the acceptable authentication schemes. If no roles or policies are provided, then the default policy will be used. If no action method is specified, then the policy will be applied to all the actions defined by the controller.

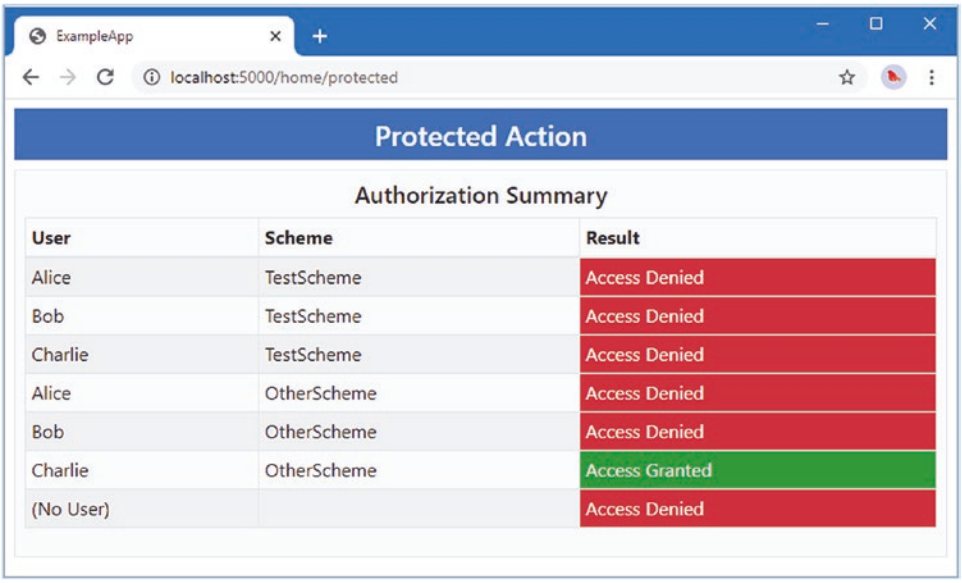
During startup, AuthorizationPolicyConvention will receive details of all the action methods in the application and will add an object that implements the IAuthorizeData interface to the action's endpoint metadata. This is the interface that is implemented by the Authorize attribute and that the authorization middleware (and the custom replacement created earlier) looks for to build an authorization policy.

In Listing 15-29, I have used the new class to add policies to the action methods defined by the Home controller.

Listing 15-29. Applying an Authorization Policy in the Startup.cs File in the ExampleApp Folder

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

    services.AddTransient<IAuthorizationHandler, CustomRequirementHandler>();

    services.AddAuthentication(opts => {
        opts.DefaultScheme
            = CookieAuthenticationDefaults.AuthenticationScheme;
    }).AddCookie(opts => {
        opts.LoginPath = "/signin";
        opts.AccessDeniedPath = "/signin/403";
    });
    services.AddAuthorization(opts => {
        AuthorizationPolicies.AddPolicies(opts);
    });
    services.AddRazorPages(opts => {
        opts.Conventions.AuthorizePage("/Secret", "NotAdmins");
    });
    services.AddControllersWithViews(opts => {
        opts.Conventions.Add(new AuthorizationPolicyConvention("Home",
            policy: "NotAdmins"));
        opts.Conventions.Add(new AuthorizationPolicyConvention("Home",
            action: "Protected", policy: "UsersExceptBob"));
    });
}
...
```

The conventions apply the `NotAdmins` policy to all the `Home` controller's actions and apply the `UsersExceptBob` policy to just the `Protected` action. Restart ASP.NET Core and request `http://localhost:5000/home/protected` to see the effect of the new policies, shown in Figure 15-12, which are applied in combination with the `Authorize` attributes applied directly to the controller class.



Protected Action

Authorization Summary

User	Scheme	Result
Alice	TestScheme	Access Denied
Bob	TestScheme	Access Denied
Charlie	TestScheme	Access Denied
Alice	OtherScheme	Access Denied
Bob	OtherScheme	Access Denied
Charlie	OtherScheme	Access Granted
(No User)		Access Denied

Figure 15-12. *Applying authorization policies using the application model*

Summary

In this chapter, I explained how ASP.NET Core authorizes requests using policies. Authorization policies are a key building block on which Identity relies, even though few applications need to work with them directly. In the next chapter, I introduce Identity to the example project and create a custom user store.