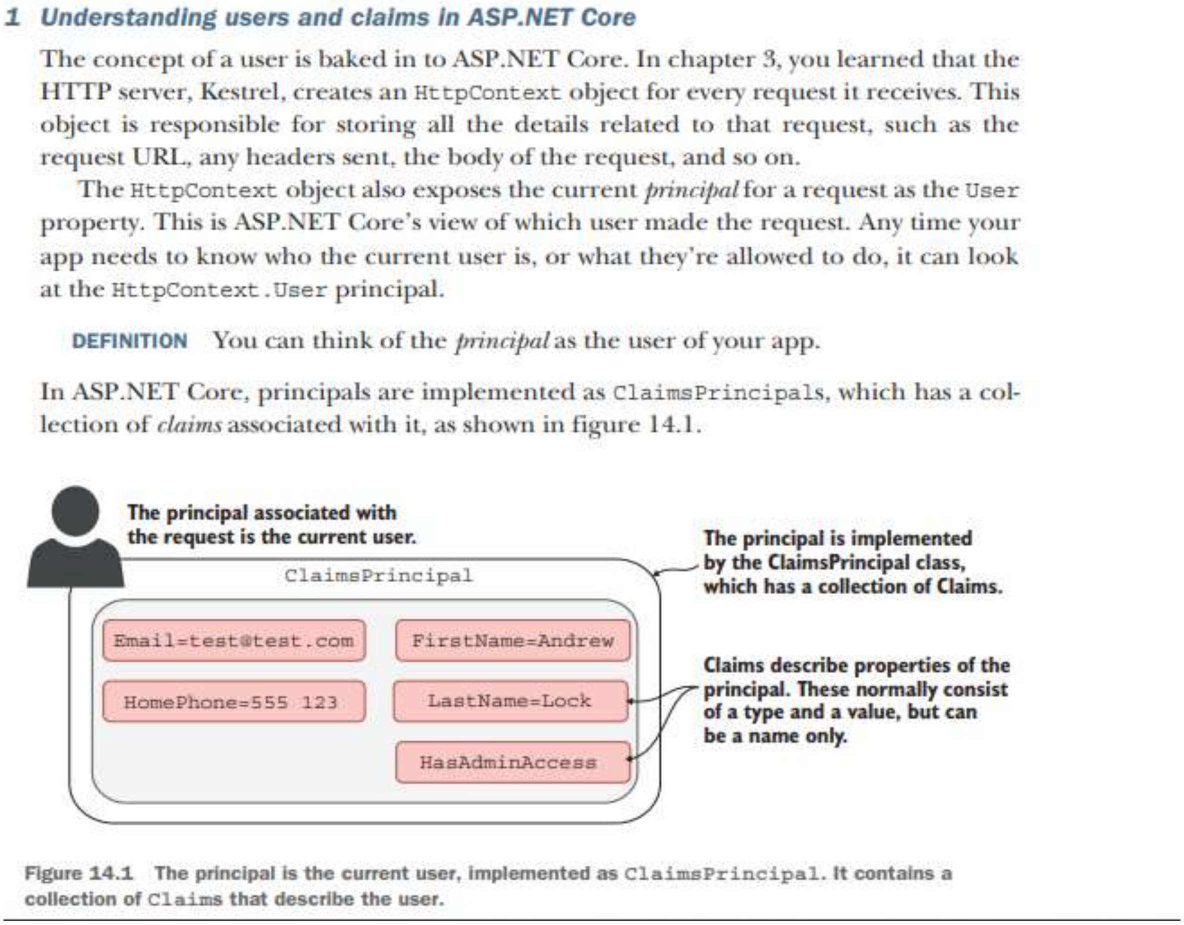
Web Security

When you think about adding users to your application, you typically have two aspects to consider:

1. **Authentication**—The process of creating users and letting them log in to your app
   The process of determining who you are.
2. **Authorization**—Customizing the experience and controlling what users can do, based on the current logged-in user.
   The process of determining what you're allowed to do.

Generally, you need to know who the user is before you can determine what they're allowed to do, so authentication always comes first, followed by authorization.



1 Understanding users and claims in ASP.NET Core

The concept of a user is baked in to ASP.NET Core. In chapter 3, you learned that the HTTP server, Kestrel, creates an `HttpContext` object for every request it receives. This object is responsible for storing all the details related to that request, such as the request URL, any headers sent, the body of the request, and so on.

The `HttpContext` object also exposes the current *principal* for a request as the `User` property. This is ASP.NET Core's view of which user made the request. Any time your app needs to know who the current user is, or what they're allowed to do, it can look at the `HttpContext.User` principal.

DEFINITION You can think of the *principal* as the user of your app.

In ASP.NET Core, principals are implemented as `ClaimsPrincipals`, which has a collection of *claims* associated with it, as shown in figure 14.1.

The principal associated with the request is the current user.

ClaimsPrincipal

Email=test@test.com

FirstName=Andrew

HomePhone=555 123

LastName=Lock

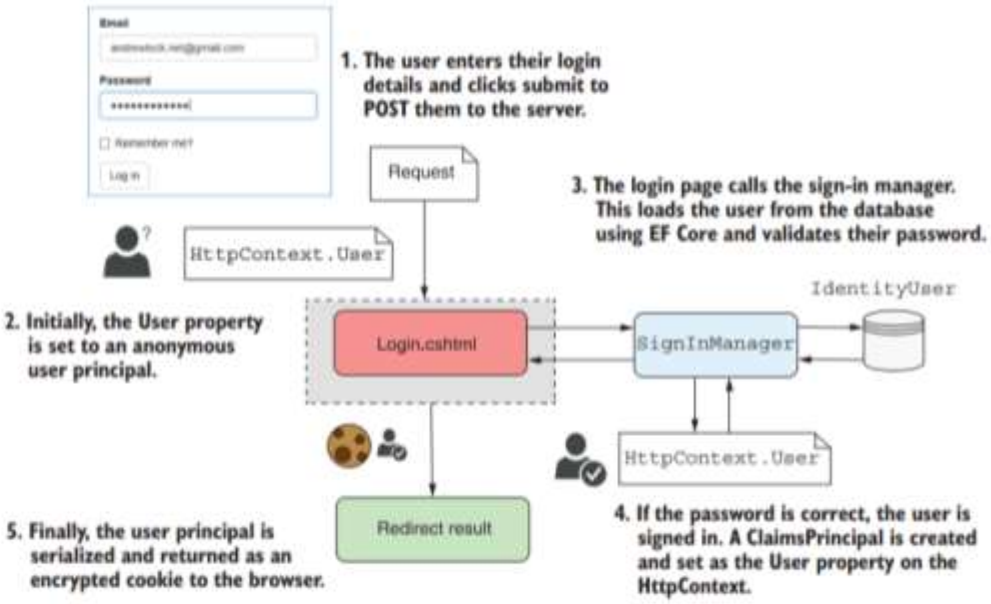
HasAdminAccess

The principal is implemented by the ClaimsPrincipal class, which has a collection of Claims.

Claims describe properties of the principal. These normally consist of a type and a value, but can be a name only.

Figure 14.1 The principal is the current user, implemented as `ClaimsPrincipal`. It contains a collection of `Claims` that describe the user.

SIGNING IN TO AN ASP.NET CORE APPLICATION



1. The user enters their login details and clicks submit to POST them to the server.

2. Initially, the User property is set to an anonymous user principal.

3. The login page calls the sign-in manager. This loads the user from the database using EF Core and validates their password.

4. If the password is correct, the user is signed in. A ClaimsPrincipal is created and set as the User property on the HttpContext.

5. Finally, the user principal is serialized and returned as an encrypted cookie to the browser.

Figure 14.2 Signing in to an ASP.NET Core application. `SignInManager` is responsible for setting `HttpContext.User` to the new principal and serializing the principal to the encrypted cookie.

The meaty work happens inside the SignInManager service. This is responsible for loading a user entity with the provided username from the database and validating that the password they provided is correct.

WARNING Never store passwords in the database directly. They should be hashed using a strong one-way algorithm. The ASP.NET Core Identity system does this for you, but it’s always wise to reiterate this point!

The key to persisting your identity across multiple requests lies in the final step of figure 14.2, where you serialized the principal in a cookie. Browsers will automatically send this cookie with all requests made to your app, so you don’t need to provide your password with every request.

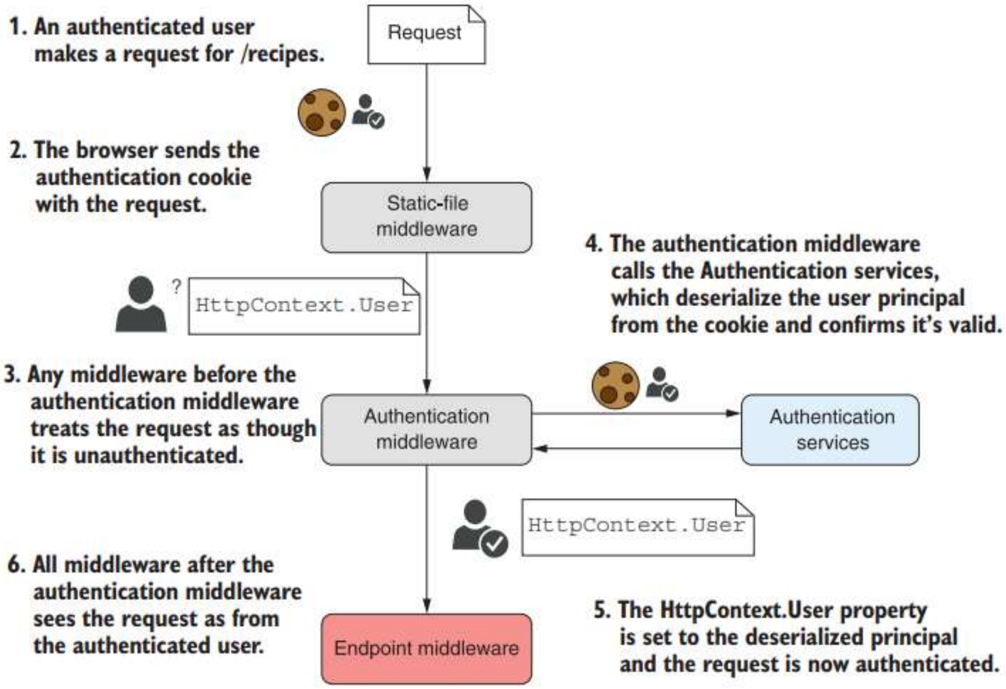

1. An authenticated user makes a request for /recipes.
Request
2. The browser sends the authentication cookie with the request.
Static-file middleware
?
HttpContext.User
4. The authentication middleware calls the Authentication services, which deserialize the user principal from the cookie and confirms it's valid.
3. Any middleware before the authentication middleware treats the request as though it is unauthenticated.
Authentication middleware
Authentication services
HttpContext.User
6. All middleware after the authentication middleware sees the request as from the authenticated user.
Endpoint middleware
5. The HttpContext.User property is set to the deserialized principal and the request is now authenticated.

Figure 14.3 A subsequent request after signing in to an application. The cookie sent with the request contains the user principal, which is validated and used to authenticate the request.

NOTE The AuthenticationMiddleware is only responsible for authenticating incoming requests and setting the ClaimsPrincipal if the request contains an authentication cookie. It is not responsible for redirecting unauthenticated requests to the login page or rejecting unauthorized requests—that is handled by the AuthorizationMiddleware.

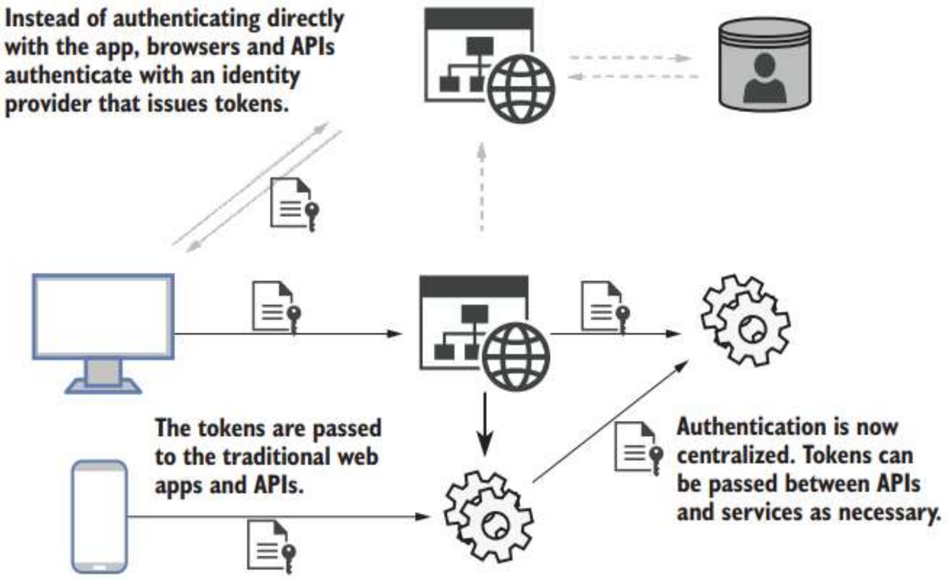

Instead of authenticating directly with the app, browsers and APIs authenticate with an identity provider that issues tokens.
The tokens are passed to the traditional web apps and APIs.
Authentication is now centralized. Tokens can be passed between APIs and services as necessary.

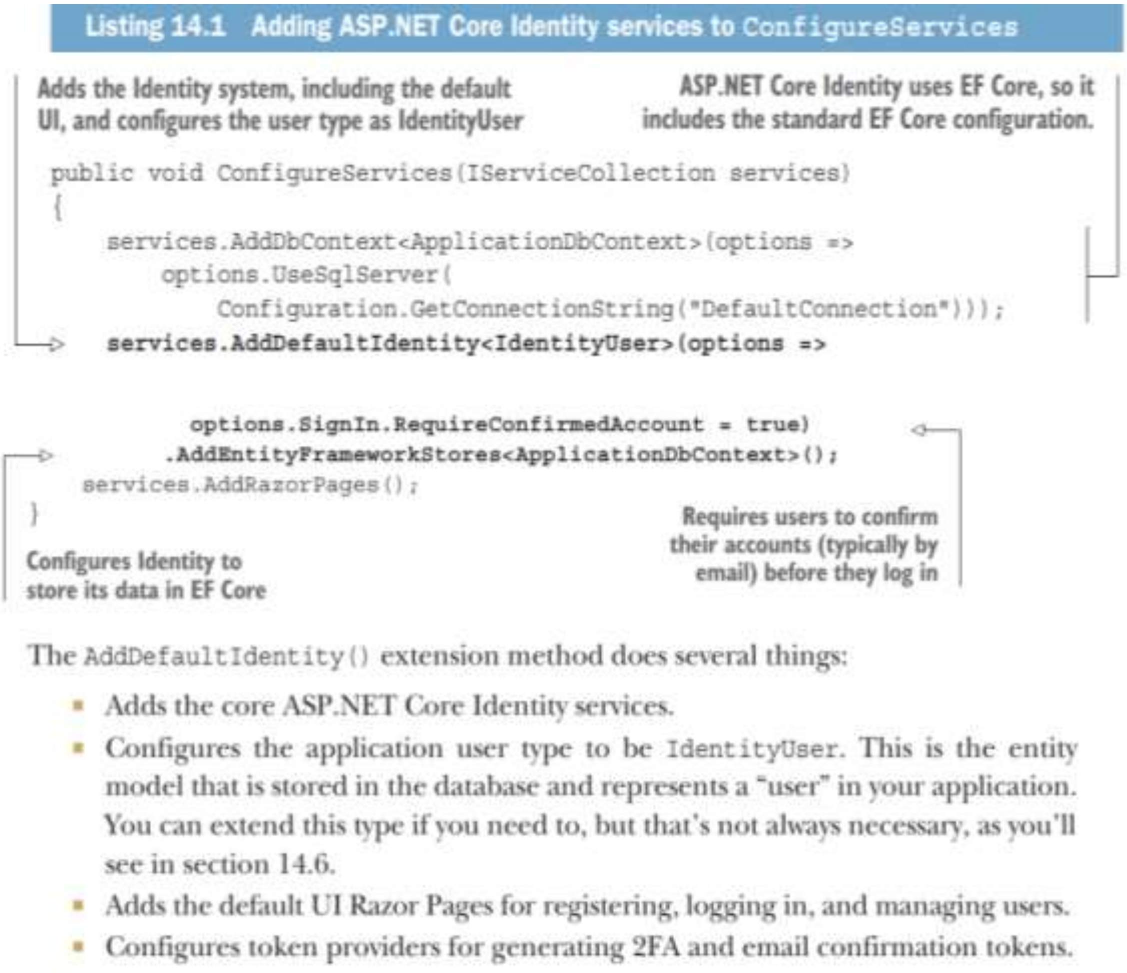
Figure 14.6 An alternative architecture involves using a central identity provider to handle all the authentication and user management for the system. Tokens are passed back and forth between the identity provider, apps, and APIs.

This architecture is clearly more complicated on the face of it, as you've thrown a whole new service—the identity provider—into the mix, but in the long run this has a number of advantages:

- *Users can share their identity between multiple services.* As you're logged in to the central identity provider, you're essentially logged in to *all* apps that use that service. This gives you the single-sign-on experience, where you don't have to keep logging in to multiple services.
- *Reduced duplication.* All of the sign-in logic is encapsulated in the identity provider, so you don't need to add sign-in screens to all your apps.
- *Can easily add new providers.* Whether you use the identity provider approach or the traditional approach, it's possible to use external services to handle the authentication of users. You'll have seen this on apps that allow you to "log in using Facebook" or "log in using Google," for example. If you use a centralized identity provider, adding support for additional providers can be handled in one place, instead of having to configure every app and API explicitly.

TIP Wherever possible, I recommend this approach, as it delegates security responsibilities to someone else. You can't lose your user's details if you never had them!

After creating ASP.net Core Web App with Identity services, following DI is created.



Listing 14.1 Adding ASP.NET Core Identity services to ConfigureServices

Adds the Identity system, including the default UI, and configures the user type as IdentityUser

ASP.NET Core Identity uses EF Core, so it includes the standard EF Core configuration.

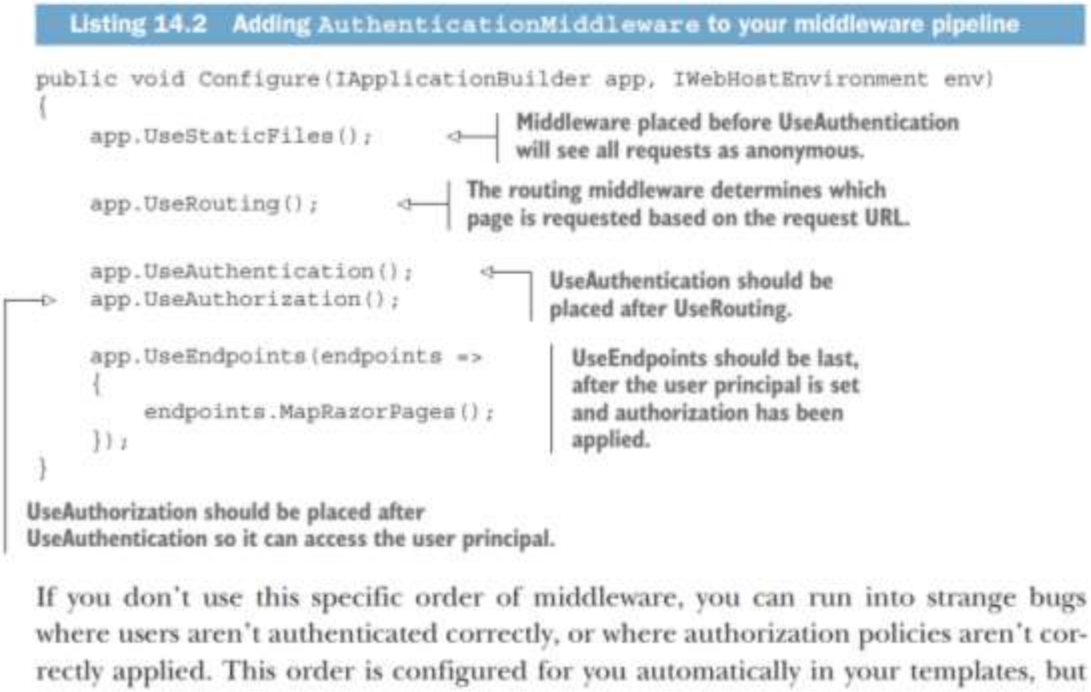
```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>(options =>

          options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationDbContext>();
  services.AddRazorPages();
}
```

Configures Identity to store its data in EF Core

Requires users to confirm their accounts (typically by email) before they log in

The `AddDefaultIdentity()` extension method does several things:

- Adds the core ASP.NET Core Identity services.
- Configures the application user type to be `IdentityUser`. This is the entity model that is stored in the database and represents a "user" in your application. You can extend this type if you need to, but that's not always necessary, as you'll see in section 14.6.
- Adds the default UI Razor Pages for registering, logging in, and managing users.
- Configures token providers for generating 2FA and email confirmation tokens.



Listing 14.2 Adding `AuthenticationMiddleware` to your middleware pipeline

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

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

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

Middleware placed before UseAuthentication will see all requests as anonymous.

The routing middleware determines which page is requested based on the request URL.

UseAuthentication should be placed after UseRouting.

UseEndpoints should be last, after the user principal is set and authorization has been applied.

UseAuthorization should be placed after UseAuthentication so it can access the user principal.

If you don't use this specific order of middleware, you can run into strange bugs where users aren't authenticated correctly, or where authorization policies aren't correctly applied. This order is configured for you automatically in your templates, but

You need to update the database for Identity services using Nuget Package Console: update-database.

This will create all the necessary Database and Tables for storing user information, explore the database using Server Explorer menu.

It's important to understand the difference between the IdentityUser entity (stored in the AspNetUsers table) and the ClaimsPrincipal, which is exposed on Http�Context.User. When a user first logs in, an IdentityUser is loaded from the database. This entity is combined with additional claims for the user from the AspNetUser�Claims table to create a ClaimsPrincipal. It's this ClaimsPrincipal that is used for authentication and is serialized to the authentication cookie, not the IdentityUser.

It's useful to have a mental model of the underlying database schema Identity uses, but in day-to-day work, you shouldn't have to interact with it directly—that's what Identity is for, after all! In the next section, we'll look at the other end of the scale— the UI of the app, and what you get out of the box with the default UI.

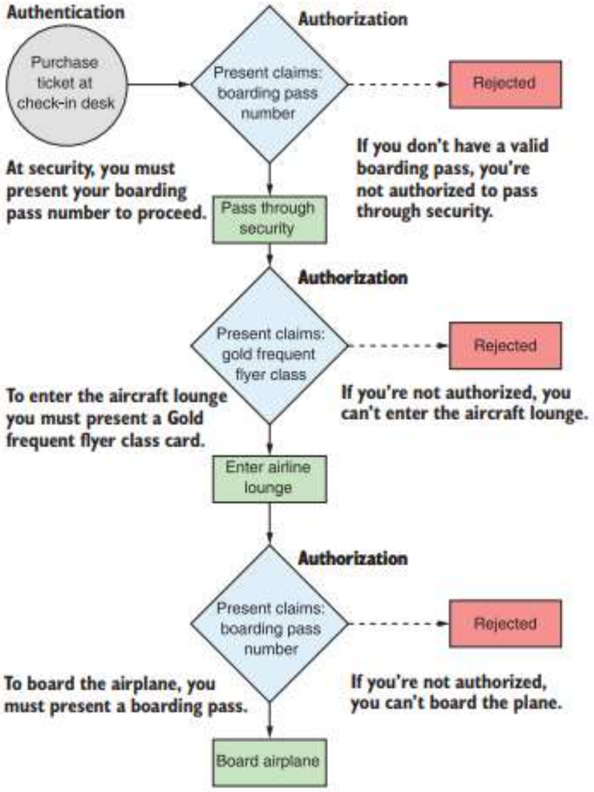
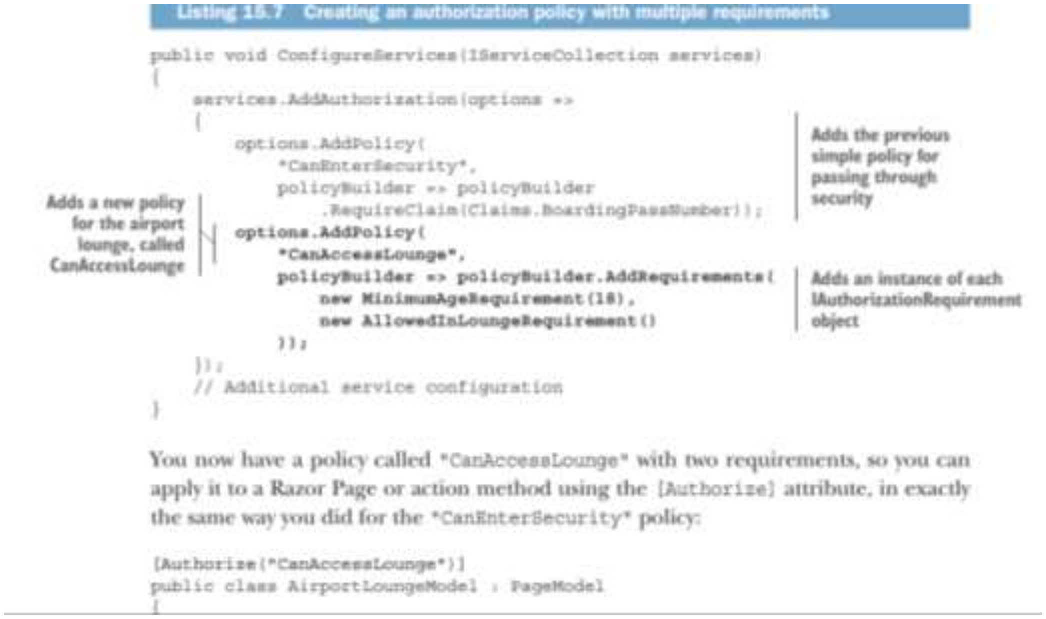

Authentication
Purchase ticket at check-in desk
Authorization
Present claims: boarding pass number
Rejected
If you don't have a valid boarding pass, you're not authorized to pass through security.
At security, you must present your boarding pass number to proceed.
Pass through security
Authorization
Present claims: gold frequent flyer class
Rejected
If you're not authorized, you can't enter the aircraft lounge.
To enter the aircraft lounge you must present a Gold frequent flyer class card.
Enter airline lounge
Authorization
Present claims: boarding pass number
Rejected
To board the airplane, you must present a boarding pass.
If you're not authorized, you can't board the plane.
Board airplane

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

See the Airport Project Code for the implementation of this process.



Listing 15.7 Creating an authorization policy with multiple requirements

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

Adds the previous simple policy for passing through security

Adds a new policy for the airport lounge, called CanAccessLounge

Adds an instance of each IAuthorizationRequirement object

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

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

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

CREATING AUTHORIZATION HANDLERS TO SATISFY YOUR REQUIREMENTS

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

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

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

Listing 15.8 FrequentFlyerHandler for AllowedInLoungeRequirement

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

You must override the abstract HandleRequirementAsync method.

The handler implements AuthorizationHandler<T>.

The context contains details such as the ClaimsPrincipal user object.

The requirement instance to handle

Checks whether the user has the FrequentFlyerClass claim with the Gold value

If the user had the necessary claim, then mark the requirement as satisfied by calling Succeed.

If the requirement wasn't satisfied, do nothing.

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

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

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

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

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

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

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

Listing 15.9 IsAirlineEmployeeHandler

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

The handler implements AuthorizationHandler<T>.

You must override the abstract HandleRequirementAsync method.

If the user has the necessary claim, mark the requirement as satisfied by calling Succeed.

Checks whether the user has the EmployeeNumber claim

If the requirement wasn't satisfied, do nothing.

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

This pattern of authorization handler is common,[1] but in some cases, instead of checking for a *success* condition, you might want to check for a *failure* condition. In the

[1] I'll leave the implementation of MinimumAgeHandler for MinimumAgeRequirement as an exercise. You can find an example in the code samples for the chapter.

airport example, you don't want to authorize someone who was previously banned from the lounge, even if they would otherwise be allowed to enter.

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

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

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

        return Task.CompletedTask;
    }
}
```

The handler implements AuthorizationHandler<T>.

You must override the abstract HandleRequirementAsync method.

Checks whether the user has the IsBanned claim

If the user has the claim, fail the requirement by calling Fail. The whole policy will fail.

If the claim wasn't found, do nothing.

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

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

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

Listing 15.11 Registering the authorization handlers with the DI container

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthorization(options =>
    {
        options.AddPolicy(
            "CanEnterSecurity",
            policyBuilder => policyBuilder
                .RequireClaim(Claims.BoardingPassNumber));
        options.AddPolicy(
            "CanAccessLounge",
            policyBuilder => policyBuilder.AddRequirements(
                new MinimumAgeRequirement(18),
                new AllowedInLoungeRequirement()
            ));
    });
    services.AddSingleton<IAuthorizationHandler, MinimumAgeHandler>();
    services.AddSingleton<IAuthorizationHandler, FrequentFlyerHandler>();
    services
        .AddSingleton<IAuthorizationHandler, BannedFromLoungeHandler>();
    services
        .AddSingleton<IAuthorizationHandler, IsAirlineEmployeeHandler>();
    // Additional service configuration
}
```

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

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

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

As well as applying the [Authorize] attribute explicitly to actions and Razor Pages, you can also configure it globally, so that a policy is applied to every Razor Page or controller in your application. Additionally, for Razor Pages you can apply different authorization policies to different folders. You can read more about applying authorization policies using conventions in Microsoft's "Razor Pages authorization conventions in ASP.NET Core" documentation: http://mng.bz/nMm2.

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

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