CHAPTER 23

External Authentication, Part 2

In this chapter, I complete my description of the Identity features by showing you how to store authentication tokens received from external services and by adding support for authentication with the real services provided by Google and Facebook.

Preparing for This Chapter

This chapter uses the ExampleApp project from Chapter 22. To prepare for this chapter, comment out the statement in the Startup class that registers the custom user validation class, as shown in Listing 23-1. The validator restricts email addresses to specific domains, which won't work when using creating accounts from real authentication services.

Listing 23-1. Disabling the User Validation Class in the Startup.cs File in the ExampleApp Folder

```
...
services.AddIdentityCore<AppUser>(opts => {
    opts.Tokens.EmailConfirmationTokenProvider = "SimpleEmail";
    opts.Tokens.ChangeEmailTokenProvider = "SimpleEmail";
    opts.Tokens.PasswordResetTokenProvider =
        TokenOptions.DefaultPhoneProvider;

    opts.Password.RequireNonAlphanumeric = false;
    opts.Password.RequireLowercase = false;
    opts.Password.RequireUppercase = false;
    opts.Password.RequireDigit = false;
    opts.Password.RequiredLength = 8;
    opts.Lockout.MaxFailedAccessAttempts = 3;
    opts.SignIn.RequireConfirmedAccount = true;
})
.AddTokenProvider<EmailConfirmationTokenGenerator>("SimpleEmail")
.AddTokenProvider<PhoneConfirmationTokenGenerator>
    (TokenOptions.DefaultPhoneProvider)
.AddTokenProvider<TwoFactorSignInTokenGenerator>
    (IdentityConstants.TwoFactorUserIdScheme)
.AddTokenProvider<AuthenticatorTokenProvider<AppUser>>
    (TokenOptions.DefaultAuthenticatorProvider)
.AddSignInManager()
.AddRoles<AppRole>();
```


A. Freeman, *Pro ASP.NET Core Identity*, https://doi.org/10.1007/978-1-4842-6858-2_23

```
//services.AddSingleton<IUserValidator<AppUser>, EmailValidator>();
services.AddSingleton<IPasswordValidator<AppUser>, PasswordValidator>();
services.AddScoped<IUserClaimsPrincipalFactory<AppUser>,
    AppUserClaimsPrincipalFactory>();
services.AddSingleton<IRoleValidator<AppRole>, RoleValidator>();
...
```

Open a new command prompt, navigate to the ExampleApp folder, and run the command shown in Listing 23-2 to start ASP.NET Core.

■ **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.

Listing 23-2. Running the Example Application

```
dotnet run
```

Open a new browser window, request http://localhost:5000/signout, and click the Sign Out button to remove any authentication cookies created in previous chapters. Next, request http://localhost:5000/secret, which will produce the response shown in Figure 23-1, which allows sign-in using local credentials or with an external service.

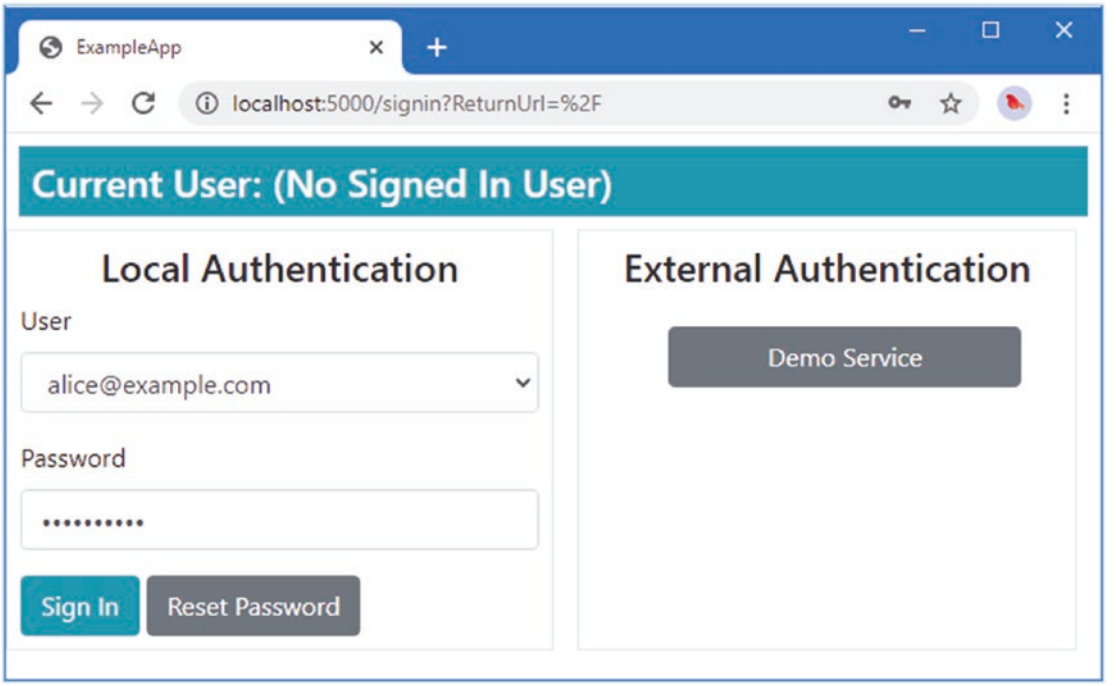

Figure 23-1. *Running the example application*

Storing Authentication Tokens

Some external authentication services provide tokens that can be used to access additional APIs. As a general rule, the set of required APIs is included in the scope requested by the application so that the user can be prompted to grant appropriate access when they are authenticated. The access token that is used to get user data during the authentication process can then be used to access the other APIs.

There is a clash of terminology between the *access tokens* produced by the OAuth authentication process and the *authentication tokens* supported by Identity. This has arisen because Identity is providing a general feature that can be used to store tokens produced by any authentication process, even though OAuth has emerged as the de facto standard.

UNDERSTANDING THE COST OF ADDITIONAL APIS

The major authentication services produce tokens that can be used widely. Google, in particular, has a wide range of APIs that provide access to just about every service it offers, including access to email, calendar, and search data.

But the cost of using these APIs can be high. Not only do some providers charge for each access to the APIs, but there is often a paid-for validation service during which the service provider assesses the application to ensure that user data is handled appropriately. This can be expensive—at the time of writing, the Google validation process can cost between $15,000 and $75,000 and requires a significant amount of work.

For this reason, I create a simulated API in this chapter to demonstrate how tokens are stored and used. I do not demonstrate the use of Google or Facebook APIs, even though I create authentication handlers for these services.

Creating the Simulated External API Controller

To simulate an API that uses access tokens for validation, add a class file named `DemoExternalApiController.cs` to the `Controllers` folder and use it to define the controller shown in Listing 23-3.

Listing 23-3. The Contents of the DemoExternalApiController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;

namespace ExampleApp.Controllers {

    [ApiController]
    [Route("api/[controller]")]
    public class DemoExternalApiController: Controller {

        private Dictionary<string, string> data
            = new Dictionary<string, string> {
                { "token1", "This is Alice's external data" },
                { "token2", "This is Dora's external data" },
            };
```

```
        [HttpGet]
        public IActionResult GetData([FromHeader] string authorization) {
            if (!string.IsNullOrEmpty(authorization)) {
                string token = authorization?[7..];
                if (!string.IsNullOrEmpty(token) && data.ContainsKey(token)) {
                    return Json(new { data = data[token] });
                }
            }
            return NotFound();
        }
    }
}
```

The controller defines a single action that provides a JSON object based on the token included in the `Authorization` request header. This is a simple example, but it provides enough functionality to demonstrate the use of an access token.

Extending the User Class

To prepare for storing access tokens, add the property to the `AppUser` class, as shown in Listing 23-4.

Listing 23-4. Adding a Property in the AppUser.cs File in the Identity Folder

```
using System;
using System.Collections.Generic;
using System.Security.Claims;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Authentication;

namespace ExampleApp.Identity {
    public class AppUser {

        public string Id { get; set; } = Guid.NewGuid().ToString();

        public string UserName { get; set; }

        public string NormalizedUserName { get; set; }

        public string EmailAddress { get; set; }
        public string NormalizedEmailAddress { get; set; }
        public bool EmailAddressConfirmed { get; set; }

        public string PhoneNumber { get; set; }
        public bool PhoneNumberConfirmed { get; set; }

        public string FavoriteFood { get; set; }
        public string Hobby { get; set; }

        public IList<Claim> Claims { get; set; }
```

```
        public string SecurityStamp { get; set; }
        public string PasswordHash { get; set; }

        public bool CanUserBeLockedout { get; set; } = true;
        public int FailedSignInCount { get; set; }
        public DateTimeOffset? LockoutEnd { get; set; }

        public bool TwoFactorEnabled { get; set; }
        public bool AuthenticatorEnabled { get; set; }
        public string AuthenticatorKey { get; set; }

        public IList<UserLoginInfo> UserLogins { get; set; }
        public IList<(string provider, AuthenticationToken token)>
            AuthTokens { get; set; }
    }
}
```

Identity provides the `AuthenticationToken` class, which defines `Name` and `Value` properties. To store tokens, I need to be able to keep track of the source of each token, so I have used a list of `(string, AuthenticationToken)` tuples for simplicity.

Extending the User Store

The `IUserAuthenticationTokenStore<T>` interface is implemented by user stores that can manage access tokens, where `T` is the user class. The interface defines the methods shown in Table 23-1. These methods define a `CancellationToken` parameter named `canceltoken` that is used to receive a notification when an asynchronous task is canceled.

Table 23-1. *The IUserAuthenticationTokenStore<T> Methods*

Name	Description
`GetTokenAsync(user, provider, name, cancelToken)`	This method returns the token granted to a user with the specified name generated by the specified provider.
`SetTokenAsync(user, provider, name, cancelToken)`	This method stores a token for the user, with the specified provider and name.
`RemoveTokenAsync(user, provider, name, cancelToken)`	This method removes the token with the specified provider and name for the specified user.

Add a class file named `UserStoreAuthenticationTokens.cs` to the `Identity/Store` folder and use it to define the partial class shown in Listing 23-5.

Listing 23-5. The Contents of the UserStoreAuthenticationTokens.cs File in the Identity/Store Folder

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Identity;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Threading;
using System.Threading.Tasks;

namespace ExampleApp.Identity.Store {
    public partial class UserStore : IUserAuthenticationTokenStore<AppUser> {

        public Task<string> GetTokenAsync(AppUser user, string loginProvider,
                string name, CancellationToken cancelToken) {
            return Task.FromResult(user.AuthTokens?
                .FirstOrDefault(t => t.provider == loginProvider
                    && t.token.Name == name).token.Value);
        }

        public Task RemoveTokenAsync(AppUser user, string loginProvider,
                string name, CancellationToken cancelToken) {
            if (user.AuthTokens!= null) {
                user.AuthTokens= user.AuthTokens.Where(t =>
                   t.provider != loginProvider
                       && t.token.Name != name).ToList();
            }
            return Task.CompletedTask;
        }

        public Task SetTokenAsync(AppUser user, string loginProvider,
             string name, string value, CancellationToken cancelToken) {
            if (user.AuthTokens== null) {
                user.AuthTokens= new List<(string, AuthenticationToken)>();
            }
            user.AuthTokens.Add((loginProvider, new AuthenticationToken {
                Name = name, Value = value }));
            return Task.CompletedTask;
        }
    }
}
```

The implementation of the interface uses the property defined in Listing 23-4 to store access tokens as tuples.

Managing Authentication Tokens

The `UserManager<T>` class provides the members described in Table 23-2 for managing authentication tokens in the user store.

Table 23-2. *The UserManager<T> Members for Managing Authentication Tokens*

Name	Description
`SupportsUserAuthenticationTokens`	This property returns true if the user store implements the `IUserAuthenticationTokenStore<T>` interface.
`GetAuthenticationTokenAsync (user, provider, name)`	This method retrieves a token by calling the user store's `GetTokenAsync` method.
`SetAuthenticationTokenAsync (user, provider, name)`	This method stores a token by calling the user store's `SetTokenAsync` method, after which the update sequence is performed.
`RemoveAuthenticationTokenAsync (user, provider, name)`	This method removes a token by calling the user store's `RemoveTokenAsync` method, after which the update sequence is applied.

The `SignInManager<T>` class also defines a method that is useful for managing authentication tokens, as described in Table 23-3.

Table 23-3. *The SignInManager<T> Method for Managing Authentication Tokens*

Name	Description
`UpdateExternalAuthentication TokensAsync(login)`	This method stores the access tokens in the specific `ExternalLoginInfo` object and stores them using the user manager's `SetAuthenticationTokenAsync` method.

Storing External Authentication Access Tokens

The `UpdateExternalAuthenticationTokensAsync` method provided by the `SignInManager<T>` class populates the user store with the authentication tokens found in the `ExternalLoginInfo.AuthenticationTokens` property. In Listing 23-6, I have updated the external authentication handler to store the tokens it receives.

Listing 23-6. Storing Tokens in the ExternalAuthHandler.cs File in the Custom Folder

```
...
public virtual async Task<bool> HandleRequestAsync() {
    if (Context.Request.Path.Equals(Options.RedirectPath)) {
        string authCode = await GetAuthenticationCode();
        (string token, string state) = await GetAccessToken(authCode);
        if (!string.IsNullOrEmpty(token)) {
            IEnumerable<Claim> claims = await GetUserData(token);
            if (claims != null) {
                ClaimsIdentity identity = new ClaimsIdentity(Scheme.Name);
                identity.AddClaims(claims);
                ClaimsPrincipal claimsPrincipal
                    = new ClaimsPrincipal(identity);
                AuthenticationProperties props =
                    PropertiesFormatter.Unprotect(state);
```

```
                props.StoreTokens(new[] { new AuthenticationToken {
                    Name = "access_token", Value = token } });
                await Context.SignInAsync(IdentityConstants.ExternalScheme,
                    claimsPrincipal, props);
                Context.Response.Redirect(props.RedirectUri);
                return true;
            }
        }
        Context.Response.Redirect(string.Format(Options.ErrorUrlTemplate,
            ErrorMessage));
        return true;
    }
    return false;
}
...
```

The new statement in Listing 23-6 adds the access token obtained from the OAuth authentication process to the `AuthenticationProperties` object using the `StoreTokens` extension method. This is one of the extension methods available for managing tokens, as described in Table 23-4.

Table 23-4. *Useful AuthenticationProperties Token Extension Methods*

Name	Description
`GetTokens()`	This method returns the authentication tokens that have been stored in the `AuthenticationProperties` object.
`GetTokenValue(name)`	This method returns the value of the token with the specified name, or `null` if there is no such token.
`StoreTokens(tokens)`	This method stores tokens expressed as an `IEnumerable<AuthenticationToken>` sequence.
`UpdateTokenValue(name, value)`	This method updates the value for a specific token.

The authentication handler adds the tokens to the `AuthenticationProperties` object, which is then available to the `SignInManager<T>` service when a user has been signed in. In Listing 23-7, I use the `SignInManager<T>.UpdateExternalAuthenticationTokensAsync` method to retrieve the tokens and add them to the user store.

Listing 23-7. Storing Authentication Tokens in the ExternalSignIn.cshtml.cs File in the Pages Folder

```
...
public async Task<IActionResult> OnGetCorrelate(string returnUrl) {
    ExternalLoginInfo info = await SignInManager.GetExternalLoginInfoAsync();

    AppUser user = await UserManager.FindByLoginAsync(info.LoginProvider,
        info.ProviderKey);
    if (user == null) {
        string externalEmail =
            info.Principal.FindFirst(ClaimTypes.Email)?.Value ?? string.Empty;
        user = await UserManager.FindByEmailAsync(externalEmail);
```

```
        if (user == null) {
            return RedirectToPage("/ExternalAccountConfirm",
                new { returnUrl });
        } else {
            UserLoginInfo firstLogin = user?.UserLogins?.FirstOrDefault();
            if (firstLogin != null && firstLogin.LoginProvider
                    != info.LoginProvider) {
                return RedirectToPage(new {
                    error =
                    $"{firstLogin.ProviderDisplayName} Authentication Expected"
                });
            } else {
                await UserManager.AddLoginAsync(user, info);
            }
        }
    }
    SignInResult result = await SignInManager.ExternalLoginSignInAsync(
            info.LoginProvider, info.ProviderKey, false, false);
    await SignInManager.UpdateExternalAuthenticationTokensAsync(info);
    if (result.Succeeded) {
        return RedirectToPage("ExternalSignIn", "Confirm",
            new { info.ProviderDisplayName, returnUrl });
    } else if (result.RequiresTwoFactor) {
        string postSignInUrl = this.Url.Page("/ExternalSignIn", "Confirm",
            new { info.ProviderDisplayName, returnUrl });
        return RedirectToPage("/SignInTwoFactor",
            new { returnUrl = postSignInUrl });
    }
    return RedirectToPage(new { error = true, returnUrl });
}
...
```

I also need to perform the same task when creating an account following an external login, as shown in Listing 23-8.

Listing 23-8. Storing Tokens in the ExternalAccountConfirm.cshtml.cs File in the Pages Folder

```
...
public async Task<IActionResult> OnPostAsync(string username) {
    ExternalLoginInfo info = await SignInManager.GetExternalLoginInfoAsync();

    if (info != null) {
        ClaimsPrincipal external = info.Principal;
        AppUser.UserName = username;
        AppUser.EmailAddress = external.FindFirstValue(ClaimTypes.Email);
        AppUser.EmailAddressConfirmed = true;
        IdentityResult result = await UserManager.CreateAsync(AppUser);
        if (result.Succeeded) {
            await UserManager.AddClaimAsync(AppUser,
                new Claim(ClaimTypes.Role, "User"));
            await UserManager.AddLoginAsync(AppUser, info);
            await SignInManager.ExternalLoginSignInAsync(info.LoginProvider,
```

```
                info.ProviderKey, false);
            await SignInManager.UpdateExternalAuthenticationTokensAsync(info);
            return Redirect(ReturnUrl);
        } else {
            foreach (IdentityError err in result.Errors) {
                ModelState.AddModelError(string.Empty, err.Description);
            }
        }
    } else {
        ModelState.AddModelError(string.Empty, "No external login found");
    }
    return Page();
}
...
```

The result is that the user store is populated with the authentication tokens provided by the authentication handler.

Using a Stored Authentication Token

The application can access stored tokens through the `UserManager<T>` class and use them to send requests to APIs that will use them to authenticate requests. Create a Razor Page named `ApiData.cshtml` in the `Pages` folder and add the content shown in Listing 23-9.

Listing 23-9. The Contents of the ApiData.cshtml File in the Pages Folder

```
@page
@model ExampleApp.Pages.ApiDataModel
@attribute [Microsoft.AspNetCore.Authorization.Authorize]
<h4 class="bg-info text-center text-white m-2 p-2">Data: @Model.Data</h4>
```

The view part of the page displays the value of a page model property named `Data`. To implement the page model, add the code shown in Listing 23-10 to the `ApiData.cshtml.cs` file in the `Pages` folder. (You will have to create this file if you are using Visual Studio Code.)

Listing 23-10. The Contents of the ApiData.cshtml.cs File in the Pages Folder

```
using ExampleApp.Identity;
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc.RazorPages;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Text.Json;
using System.Threading.Tasks;

namespace ExampleApp.Pages {

    public class ApiDataModel : PageModel {
```

```
        public ApiDataModel(UserManager<AppUser> userManager) {
            UserManager = userManager;
        }

        public UserManager<AppUser> UserManager { get; set; }

        public string Data { get; set; } = "No Data";

        public async Task OnGetAsync () {
            AppUser user = await UserManager.GetUserAsync(HttpContext.User);
            if (user != null) {
                string token = await UserManager.GetAuthenticationTokenAsync
                    (user, "demoAuth", "access_token");
                if (!string.IsNullOrEmpty(token)) {
                    HttpRequestMessage msg = new HttpRequestMessage(
                        HttpMethod.Get,
                        "http://localhost:5000/api/DemoExternalApi");
                    msg.Headers.Authorization = new AuthenticationHeaderValue
                        ("Bearer", token);
                    HttpResponseMessage resp
                        = await new HttpClient().SendAsync(msg);
                    JsonDocument doc = JsonDocument.Parse(await
                        resp.Content.ReadAsStringAsync());
                    Data = doc.RootElement.GetString("data");
                }
            }
        }
    }
}
```

The GET handler method retrieves the authorization token produced by the `demoAuth` scheme and uses it for the `Authorization` header in a request to the demonstration controller. The response from the controller is parsed into JSON and used to set the value of the `Data` property that is displayed by the view part of the page.

Restart ASP.NET Core, request `http://localhost:5000/signout`, and click the Sign Out button to sign out of the application. Request `http://localhost:5000/apidata`, which will trigger a challenge response. Click the Demo Service button to sign in with the external authentication service and sign in using `alice@example.com` as the email address and `myexternalpassword` as the password. Enter the code from your authenticator application, click the Sign In button, and then click the Continue button when the sign-in confirmation page is displayed. Once you have signed into the application, you will be redirected to the `/apidata` URL, which will use the stored token to get data from the demonstration controller. Figure 23-2 shows the entire sequence.

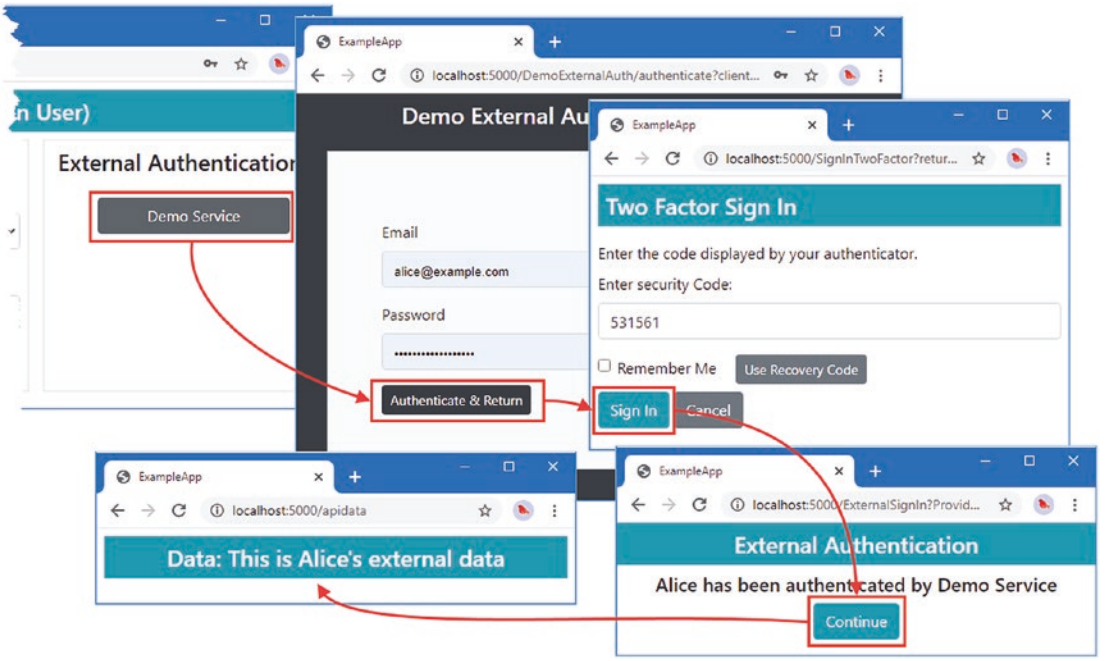

Figure 23-2. *Using a stored authentication token*

Adding Support for Real External Authentication Services

Now that all of the building blocks are in place, I can add support for real external authentication services. In the sections that follow, I extend the external authentication handler created in Chapter 22 to support the OAuth services provided by Google and Facebook, which provide the most popular authentication services.

I have not created an authentication handler for Twitter, which is also widely used for authentication. Twitter does provide an authentication service, but it uses an older version of the OAuth specification, which is more complex than the updated version used by most services. See Chapter 11 for details of how to set up the built-in Twitter authentication handler that Microsoft provides for ASP.NET Core.

Supporting Google Authentication

To register the example application, navigate to `https://console.developers.google.com` and sign in with a Google account. Click the OAuth Consent Screen option and select External for User Type, which will allow any Google account to authenticate for your application.

■ **Tip** You may see a message telling you that no APIs are available to use yet. This is not important when you only need to authenticate users.

Click Create, and you will be presented with a form. Enter **ExampleApp** into the App Name field and enter your email address in the User Support Email and Developer Contact Information sections of the form. The rest of the form can be left empty for the example application.

Click Save and Continue, and you will be presented with the scope selection screen, which is used to specify the scopes that your application requires.

Click the Add or Remove Scopes button, and you will be presented with the list of scopes that your application can request. Select three scopes: `openid`, `auth/userinfo.email`, and `auth/userinfo.profile`. Click the Update button to save your selection.

Click Save and Continue to return to the OAuth consent screen and then click Back to Dashboard. Figure 23-3 shows the sequence for configuring the consent screen.

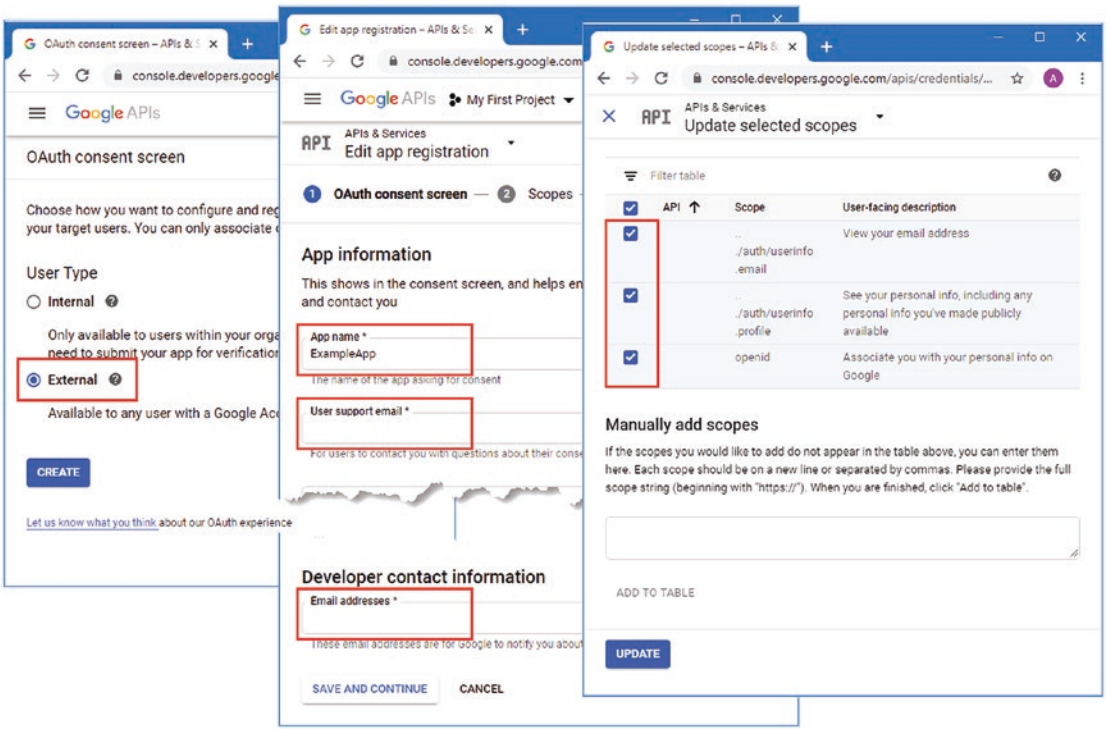

Figure 23-3. *Configuring the Google OAuth consent screen*

■ **Tip** You can find the Google documentation for OAuth at `https://developers.google.com/identity/protocols/oauth2/web-server`.

Configuring Application Credentials

The next step is to create credentials for the application. Click the Credentials link, click the Create Credentials button at the top of the page, and select OAuth Client ID from the list of options.

Select Web Application from the Application Type list and enter **ExampleApp** in the Name field. Click Add URL in the Authorized Redirect URIs section and enter **`http://localhost:5000/signin-google`** into the text field. Click the Create button, and you will be presented with the client ID and client secret for your application, as shown in Figure 23-4 (although I have blurred the details since these are for my account). Make a note of the ID and secret.

UNDERSTANDING THE REDIRECT URL

To configure the example application, I have specified the URL `http://localhost:5000/signin-google`. The redirection URLs are evaluated by the browser, and since the browser and the ASP.NET Core application are running on the same machine, using localhost in the URL means that the redirection performed by the Google service will target the example application.

For real applications, you must use a URL that contains a hostname that can be correctly evaluated by your users' browsers. This can be a corporate hostname for line-of-business applications or a name registered in the global DNS for internet-facing applications. You should not use localhost except during development.

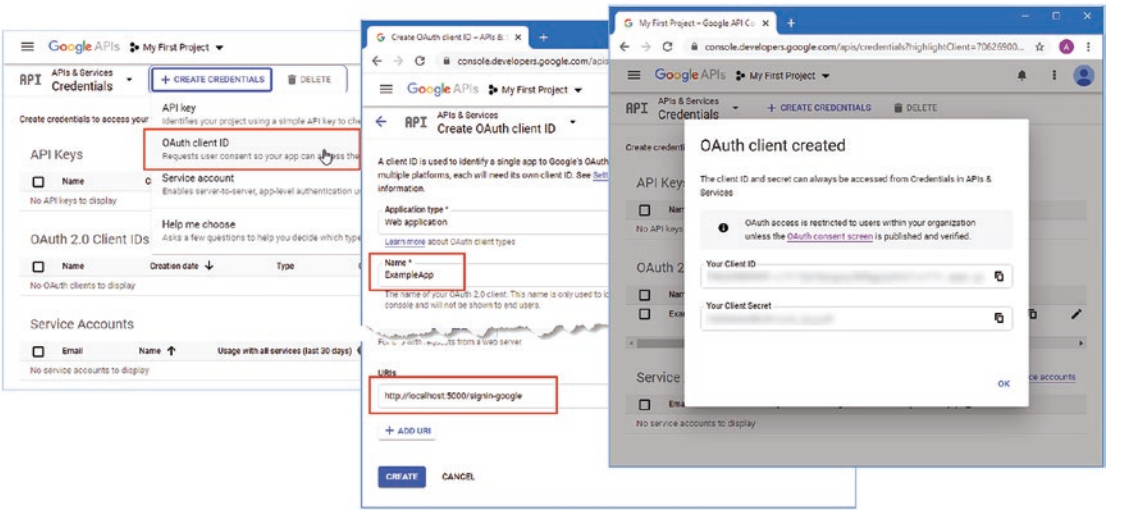

Figure 23-4. *Configuring application credentials*

Creating the Authentication Handler

The next step is to create an authentication handler that will use the Google OAuth service to authenticate users. Add a class file named `GoogleHandler.cs` to the `Custom` folder with the code shown in Listing 23-11.

Listing 23-11. The Contents of the GoogleHandler.cs File in the Custom Folder

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;
using System;
using System.Collections.Generic;
using System.Security.Claims;
using System.Text.Json;
```

```
using System.Threading.Tasks;

namespace ExampleApp.Custom {

    public class GoogleOptions : ExternalAuthOptions {
        public override string RedirectPath { get; set; } = "/signin-google";
        public override string AuthenticationUrl =>
            "https://accounts.google.com/o/oauth2/v2/auth";
        public override string ExchangeUrl =>
            "https://www.googleapis.com/oauth2/v4/token";
        public override string DataUrl =>
             "https://www.googleapis.com/oauth2/v2/userinfo";
    }

    public class GoogleHandler : ExternalAuthHandler {

        public GoogleHandler(IOptions<GoogleOptions> options,
            IDataProtectionProvider dp,
            ILogger<GoogleHandler> logger) : base(options, dp, logger) {}

        protected override IEnumerable<Claim> GetClaims(JsonDocument jsonDoc) {
            List<Claim> claims = new List<Claim>();
            claims.Add(new Claim(ClaimTypes.NameIdentifier,
                jsonDoc.RootElement.GetString("id")));
            claims.Add(new Claim(ClaimTypes.Name,
                jsonDoc.RootElement.GetString("name")?.Replace(" ", "_")));
            claims.Add(new Claim(ClaimTypes.Email,
               jsonDoc.RootElement.GetString("email")));
            return claims;
        }

        protected async override Task<string> GetAuthenticationUrl(
                AuthenticationProperties properties) {
            if (CheckCredentials()) {
                return await base.GetAuthenticationUrl(properties);
            } else {
                return string.Format(Options.ErrorUrlTemplate, ErrorMessage);
            }
        }

        private bool CheckCredentials() {
            string secret = Options.ClientSecret;
            string id = Options.ClientId;
            string defaultVal = "ReplaceMe";
            if (string.IsNullOrEmpty(secret) || string.IsNullOrEmpty(id)
                || defaultVal.Equals(secret) || defaultVal.Equals(secret)) {
                    ErrorMessage = "External Authentication Secret or ID Not Set";
                    Logger.LogError("External Authentication Secret or ID Not Set");
                return false;
            }
```

```
            return true;
        }
    }
}
```

The handler is derived from the `ExternalAuthHandler` class created in Chapter 22. To support the Google service, I have defined a new callback URL and specified the URLs for each part of the process, which I have summarized in Table 23-5.

Table 23-5. *The Google OAuth URLs*

Step	URL
Authentication	`https://accounts.google.com/o/oauth2/v2/auth`
Token Exchange	`https://www.googleapis.com/oauth2/v4/token`
User Data	`https://www.googleapis.com/oauth2/v2/userinfo`

I also have to map a different set of JSON properties to claims when signing a user into the application. Each authentication service returns data in a different format, and I find the easiest approach to writing an authentication handler is to print out the JSON response and pick out the properties I require. For this example, I have used the `id`, `name`, and `email` properties.

Notice that I replace any spaces when I create the `Name` claim, like this:

```
...
claims.Add(new Claim(ClaimTypes.Name,
    jsonDoc.RootElement.GetString("name")?.Replace(" ", "_")));
...
```

The Google data will contain a name such as `Adam Freeman`, which won't be accepted as an Identity account name. To avoid a validation error, I replace spaces with underscores (the _ character).

I also defined a method named `CheckCredentials` in Listing 23-11. You must create your own client ID and client secret and use them to configure the application. The `CheckCredentials` method is called in the `GetAuthenticationUrl` method and displays an error if the credentials have not been set.

Configuring the Application

The final step is to register the authentication handler and specify the client ID and secret, as shown in Listing 23-12. Use the ID and secret you created in the previous section instead of the placeholder strings in the listing.

Listing 23-12. Configuring the Application in the Startup.cs File in the ExampleApp Folder

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddSingleton<ILookupNormalizer, Normalizer>();
    services.AddSingleton<IUserStore<AppUser>, UserStore>();
    services.AddSingleton<IEmailSender, ConsoleEmailSender>();
    services.AddSingleton<ISMSSender, ConsoleSMSSender>();
    services.AddSingleton<IPasswordHasher<AppUser>, SimplePasswordHasher>();
    services.AddSingleton<IRoleStore<AppRole>, RoleStore>();
```

```
    services.AddOptions<ExternalAuthOptions>();

    services.Configure<GoogleOptions>(opts => {
        opts.ClientId = "ReplaceMe";
        opts.ClientSecret = "ReplaceMe";
    });

    // ...statements omitted for brevity...

    services.AddAuthentication(opts => {
        opts.DefaultScheme = IdentityConstants.ApplicationScheme;
        opts.AddScheme<ExternalAuthHandler>("demoAuth", "Demo Service");
        opts.AddScheme<GoogleHandler>("google", "Google");
    }).AddCookie(IdentityConstants.ApplicationScheme, opts => {
        opts.LoginPath = "/signin";
        opts.AccessDeniedPath = "/signin/403";
    })
    .AddCookie(IdentityConstants.TwoFactorUserIdScheme)
    .AddCookie(IdentityConstants.TwoFactorRememberMeScheme)
    .AddCookie(IdentityConstants.ExternalScheme);

    services.AddAuthorization(opts => {
        AuthorizationPolicies.AddPolicies(opts);
        opts.AddPolicy("Full2FARequired", builder => {
            builder.RequireClaim("amr", "mfa");
        });
    });
    services.AddRazorPages();
    services.AddControllersWithViews();
}
...
```

The `Configure` method is used to configure the `GoogleOptions` object that will be provided to the authentication handler through dependency injection. The other change in the listing registers the Google authentication handler with the `AddScheme` method.

USING SERVICE-SPECIFIC QUERY STRING PARAMETERS

Most services support optional query strings in the authentication redirection URL that control additional features. One useful option for the Google service is `login_hint`, which can be set to the email address of the user who is to be authenticated. Google will use the email address to simplify the authentication process.

Restart ASP.NET Core and request `http://localhost:5000/secret`. When challenged, click the Google button, and you will be prompted to sign in with a Google account and grant access to the application, as shown in Figure 23-5.

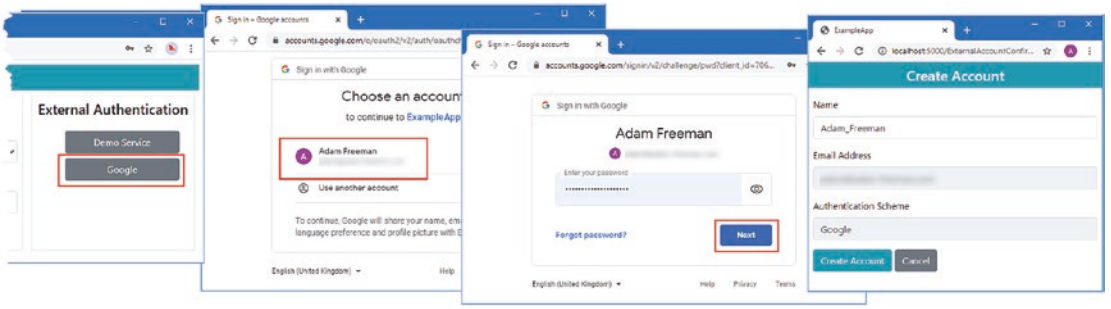

Figure 23-5. *Authenticating users with Google*

Once authenticated, you will be presented with the Create Account screen, which displays the name and email address of the Google account. Click the Create Account button to create a new Identity account and redirection to the protected resource. You can see the account that has been created by requesting `http://localhost:5000/users`, which will display the newly created account alongside those used to seed the user store, as shown in Figure 23-6.

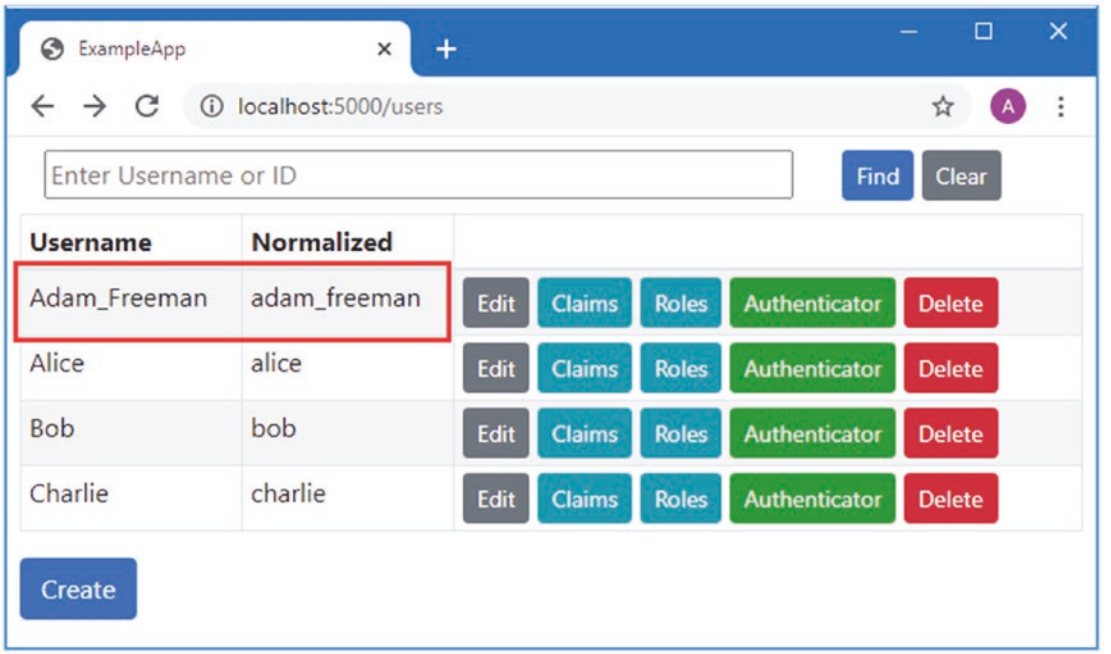

Figure 23-6. *Inspecting the newly created account*

Supporting Facebook Authentication

To register the application with Facebook, go to `https://developers.facebook.com/apps` and sign in with your Facebook account. Click the Create App button, select Build Connected Experiences from the list, and click the Continue button. Enter **ExampleApp** into the App Display Name field and click the Create App button. Figure 23-7 shows this sequence.

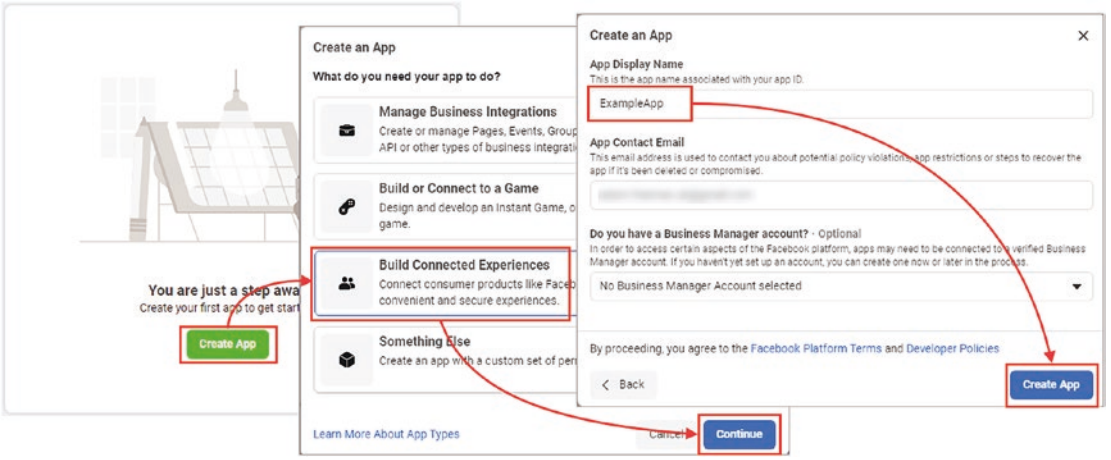

Figure 23-7. *Creating a new application*

Once you have created a Facebook application, you will be returned to the developer dashboard and presented with a list of optional products to use. Locate Facebook Login and click the Setup button. You will see a set of quick-start options, but they can be ignored because the important configuration options are shown under the Facebook Login ➤ Settings section that appears on the left side of the dashboard display, as shown in Figure 23-8.

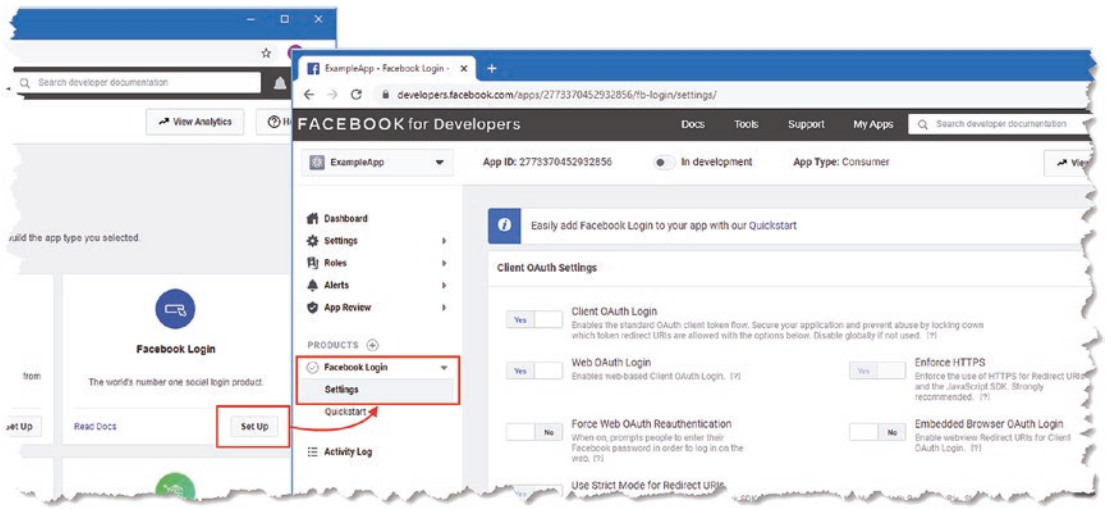

Figure 23-8. *The Facebook Login settings*

No configuration changes are required for the example application because Facebook makes it easy to use OAuth during the development of a project. When you are ready to deploy the application, you will need to return to this page and finalize your configuration, including providing the public-facing redirection URL, which will replace the localhost URL I use in this chapter. Details of the configuration options are included in the Facebook Login documentation, which can be found at https://developers.facebook.com/docs/facebook-login.

Obtaining Application Credentials

Navigate to the Basic section in the Settings area to get the App ID and App Secret values, as shown in Figure 23-9, which are the terms that Facebook uses for the client ID and secret. (The App Secret value is hidden until you click the Show button.) Make a note of these values, which will be required to configure the application.

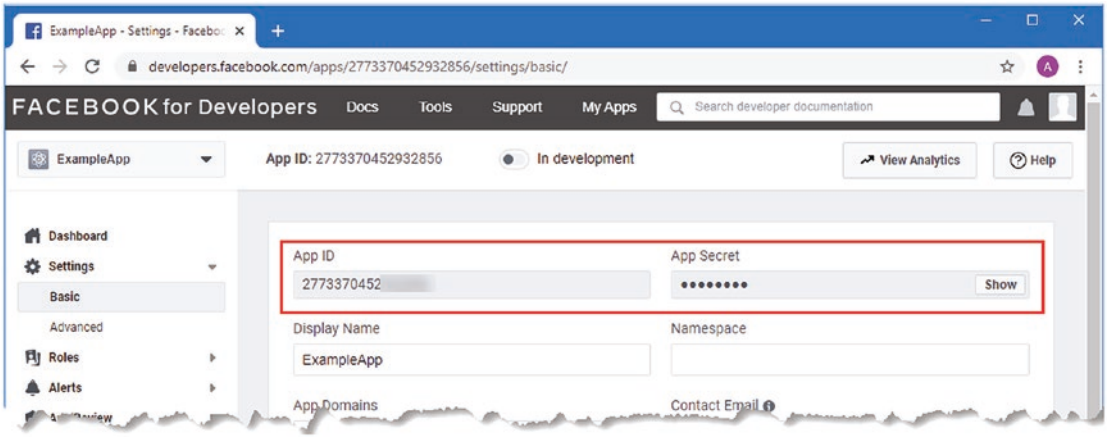

Figure 23-9. *The application credentials for external authentication*

Creating the Authentication Handler

Add a class file named FacebookHandler.cs to the Custom folder and use it to define the class shown in Listing 23-13.

Listing 23-13. The Contents of the FacebookHandler.cs File in the Custom Folder

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;
using System;
using System.Collections.Generic;
using System.Security.Claims;
using System.Text.Json;
```

```
namespace ExampleApp.Custom {

    public class FacebookOptions : ExternalAuthOptions {
        public override string RedirectPath { get; set; } = "/signin-facebook";
        public override string Scope { get; set; } = "email";

        public override string AuthenticationUrl =>
            "https://www.facebook.com/v8.0/dialog/oauth";
        public override string ExchangeUrl =>
            "https://graph.facebook.com/v8.0/oauth/access_token";
        public override string DataUrl =>
            "https://graph.facebook.com/v8.0/me?fields=name,email";
    }

    public class FacebookHandler : ExternalAuthHandler {

        public FacebookHandler(IOptions<FacebookOptions> options,
            IDataProtectionProvider dp,
            ILogger<FacebookHandler> logger) : base(options, dp, logger) {

            string secret = Options.ClientSecret;
            if (string.IsNullOrEmpty(secret) || "MyClientSecret"
                    .Equals(secret, StringComparison.OrdinalIgnoreCase)) {
                logger.LogError("External Authentication Secret Not Set");
            }
        }

        protected override IEnumerable<Claim> GetClaims(JsonDocument jsonDoc) {
            List<Claim> claims = new List<Claim>();

            claims.Add(new Claim(ClaimTypes.NameIdentifier,
                jsonDoc.RootElement.GetString("id")));
            claims.Add(new Claim(ClaimTypes.Name,
                jsonDoc.RootElement.GetString("name")?.Replace(" ", "_")));
            claims.Add(new Claim(ClaimTypes.Email,
                jsonDoc.RootElement.GetString("email")));
            return claims;
        }
    }
}
```

The handler is derived from the `ExternalAuthHandler` class created in Chapter 22. To support the Facebook service, I have defined a new callback URL and specified the URLs for each part of the process, which I have summarized in Table 23-6.

Table 23-6. *The Facebook OAuth URLs*

Step	URL
Authentication	https://www.facebook.com/v8.0/dialog/oauth
Token Exchange	https://graph.facebook.com/v8.0/oauth/access_token
User Data	https://graph.facebook.com/v8.0/me

The Facebook authentication service requires individual data fields to be selected using a `fields` query string parameter, which is why the value for the `DataUrl` configuration options is as follows:

```
...
https://graph.facebook.com/v8.0/me?fields=name,email
...
```

For the example application, I require the `name` and `email` fields. The complete set of fields is described at https://developers.facebook.com/docs/graph-api/reference/user.

Configuring the Application

To configure the application, add the statements shown in Listing 23-14 to the `ConfigureServices` method of the `Startup` class, ensuring that you use the client ID and secret that were displayed in the Facebook developer dashboard when you registered the application.

Listing 23-14. Configuring the Application in the Startup.cs File in the ExampleApp Folder

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddSingleton<ILookupNormalizer, Normalizer>();
    services.AddSingleton<IUserStore<AppUser>, UserStore>();
    services.AddSingleton<IEmailSender, ConsoleEmailSender>();
    services.AddSingleton<ISMSSender, ConsoleSMSSender>();
    services.AddSingleton<IPasswordHasher<AppUser>, SimplePasswordHasher>();
    services.AddSingleton<IRoleStore<AppRole>, RoleStore>();

    services.AddOptions<ExternalAuthOptions>();

    services.Configure<GoogleOptions>(opts => {
        opts.ClientId = "ReplaceMe";
        opts.ClientSecret = "ReplaceMe";
    });

    services.Configure<FacebookOptions>(opts => {
        opts.ClientId = "ReplaceMe";
        opts.ClientSecret = "ReplaceMe";
    });

    // ...statements omitted for brevity...

    services.AddAuthentication(opts => {
        opts.DefaultScheme = IdentityConstants.ApplicationScheme;
        opts.AddScheme<ExternalAuthHandler>("demoAuth", "Demo Service");
```

```
            opts.AddScheme<GoogleHandler>("google", "Google");
            opts.AddScheme<FacebookHandler>("facebook", "Facebook");
        }).AddCookie(IdentityConstants.ApplicationScheme, opts => {
            opts.LoginPath = "/signin";
            opts.AccessDeniedPath = "/signin/403";
        })
        .AddCookie(IdentityConstants.TwoFactorUserIdScheme)
        .AddCookie(IdentityConstants.TwoFactorRememberMeScheme)
        .AddCookie(IdentityConstants.ExternalScheme);

        services.AddAuthorization(opts => {
            AuthorizationPolicies.AddPolicies(opts);
            opts.AddPolicy("Full2FARequired", builder => {
                builder.RequireClaim("amr", "mfa");
            });
        });
        services.AddRazorPages();
        services.AddControllersWithViews();
}
...
```

Restart ASP.NET Core, request `http://localhost:5000/signout`, and click the Sign Out button. This will ensure that the external login performed through the Google service won't be used. Request `http://localhost:5000/secret`, and you will be challenged to sign in. Click the Facebook button, and you will be prompted to sign in with a Facebook account, as shown in Figure 23-10. Once you have signed in, you will be prompted to create an Identity account using the Facebook details, and then you will be redirected to the protected resource.

Figure 23-10. *Authenticating users with Facebook*

Simplifying the Sign-In Process

I introduced different methods of signing into the example application gradually so I could explain how important Identity features work. The result, however, is confusing because users can choose a different authentication option each time they sign in, and, worse, users have different sets of options depending on how their account was created.

If a user has an account that has been created with a password, they can choose to sign in with the password or choose any of the external authentication providers, just as long as the email address provided in the external user data matches the email address in the Identity user store. But if a user account is created following external authentication, the user won't have a password in the store and can only log in with an external provider, although they are free to switch between providers.

The overall effect is confusing, not least because all users are presented with the complete set of authentication options, even though they won't all work for every account. I am going to tidy this up by selecting a single authentication scheme for each user.

Updating the Sign-In Page

My policy will be simple: once a user has signed in with an external service, then that will be their only sign-in method. You don't have to follow this policy, but it has the advantage of being simple, consistent, and easy for the user to understand.

In Listing 23-15, I replaced the contents of the view part of the `SignIn` Razor page to support the new policy.

Listing 23-15. Replacing the Contents of the SignIn.cshtml File in the Pages Folder

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

@if (!string.IsNullOrEmpty(Model.Message)) {
    <h3 class="bg-danger text-white text-center p-2">@Model.Message</h3>
}

<h4 class="bg-info text-white m-2 p-2">Current User: @Model.Username</h4>

<div class="container-fluid">
    <div class="row">
        <div class="col">
            <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>
    </div>
    <div class="row">
        <div class="col text-center p-2">
            <div class="border p-2">
                <h6>Create a New Account</h6>
                <form method="post">
                    @foreach (var scheme in await Model.SignInManager
                            .GetExternalAuthenticationSchemesAsync()) {
                        <button class="btn btn-secondary m-2" type="submit"
```

```
                            asp-page="/externalsignin"
                            asp-route-returnUrl="@Request.Query["returnUrl"]"
                            asp-route-providername="@scheme.Name">
                                @scheme.DisplayName
                        </button>
                    }
                </form>
            </div>
        </div>
    </div>
</div>
```

The new layout removes the option to enter a password and groups the external authentication buttons with a "Create a New Account" message. In Listing 23-16, I have updated the SignIn page model class so that the POST handler method locates the user and performs a redirection for either external authentication or password authentication.

Listing 23-16. Redirecting for Authentication 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;
using Microsoft.AspNetCore.Identity;
using System.Linq;
using ExampleApp.Identity;
using SignInResult = Microsoft.AspNetCore.Identity.SignInResult;
using System;

namespace ExampleApp.Pages {
    public class SignInModel : PageModel {

        public SignInModel(UserManager<AppUser> userManager,
                SignInManager<AppUser> signInManager) {
            UserManager = userManager;
            SignInManager = signInManager;
        }

        public UserManager<AppUser> UserManager { get; set; }
        public SignInManager<AppUser> SignInManager { get; set; }

        public SelectList Users => new SelectList(
            UserManager.Users.OrderBy(u => u.EmailAddress),
                "EmailAddress", "NormalizedEmailAddress");

        public string Username { get; set; }

        public int? Code { get; set; }
```

```
        public string Message { get; set; }

        public void OnGet(int? code) {
            if (code == StatusCodes.Status401Unauthorized) {
                Message = "401 - Challenge Response";
            } else if (code == StatusCodes.Status403Forbidden) {
                Message = "403 - Forbidden Response";
            }
            Username = User.Identity.Name ?? "(No Signed In User)";
        }

        public async Task<IActionResult> OnPost(string username,
                [FromQuery] string returnUrl) {
            AppUser user = await UserManager.FindByEmailAsync(username);
            UserLoginInfo loginInfo = user?.UserLogins?.FirstOrDefault();
            if (loginInfo != null) {
                return RedirectToPage("/ExternalSignIn", new {
                    returnUrl, providerName = loginInfo.LoginProvider
                });
            }
            return RedirectToPage("SignInPassword", new { username, returnUrl });
        }
    }
}
```

If the user has external logins, then the first one in the store is used for authentication with a redirection to the `ExternalSignIn` page. If the user does have an external login—or there is no such user in the store—then a redirection to the `SignInPassword` page is performed, which I create in the next section.

Creating the Password Page

The next step is to create a new Razor Page that will prompt for passwords and validate them. Add a Razor Page named `SignInPassword.cshtml` to the `Pages` folder with the content shown in Listing 23-17.

Listing 23-17. The Contents of the SignInPassword.cshtml File in the Pages Folder

```
@page
@model ExampleApp.Pages.SignInPasswordModel

<div asp-validation-summary="All" class="text-danger m-2"></div>

 <form method="post" class="p-2">
    <input type="hidden" name="returnUrl" value="@Model.ReturnUrl" />
    <div class="form-group">
        <label>User</label>
        <input class="form-control" readonly name="username"
             value="@Model.Username" />
    </div>
    <div class="form-group">
        <label>Password</label>
        <input class="form-control" type="password" name="password" />
    </div>
```

```
    <button class="btn btn-info" type="submit">Sign In</button>
    @if (User.Identity.IsAuthenticated) {
        <a asp-page="/Store/PasswordChange" class="btn btn-secondary"
            asp-route-id="@Model.User?
                    .FindFirst(ClaimTypes.NameIdentifier)?.Value">
                Change Password
        </a>
    } else {
        <a class="btn btn-secondary" href="/password/reset">
            Reset Password
        </a>
    }
</form>
```

To define the page model class, add the code shown in Listing 23-18 to the SignInPassword.cshtml.cs file. (You will have to create this file if you are using Visual Studio Code.)

Listing 23-18. The Contents of the SignInPassword.cshtml File in the Pages Folder

```
using ExampleApp.Identity;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using System;
using System.Threading.Tasks;
using SignInResult = Microsoft.AspNetCore.Identity.SignInResult;

namespace ExampleApp.Pages {

    public class SignInPasswordModel : PageModel {

        public SignInPasswordModel(UserManager<AppUser> userManager,
                SignInManager<AppUser> signInManager) {
            UserManager = userManager;
            SignInManager = signInManager;
        }

        public UserManager<AppUser> UserManager { get; set; }
        public SignInManager<AppUser> SignInManager { get; set; }

        public string Username { get; set; }
        public string ReturnUrl { get; set; }

        public void OnGet(string username, string returnUrl) {
            Username = username;
            ReturnUrl = returnUrl;
        }

        public async Task<ActionResult> OnPost(string username,
                string password, string returnUrl) {
            SignInResult result = SignInResult.Failed;
            AppUser user = await UserManager.FindByEmailAsync(username);
```

```
            if (user != null && !string.IsNullOrEmpty(password)) {
                result = await SignInManager.PasswordSignInAsync(user, password,
                    false, true);
            }
            if (!result.Succeeded) {
                if (result.IsLockedOut) {
                    TimeSpan remaining = (await UserManager
                        .GetLockoutEndDateAsync(user))
                        .GetValueOrDefault().Subtract(DateTimeOffset.Now);
                    ModelState.AddModelError("",
                        $"Locked Out for {remaining.Minutes} mins and"
                            + $" {remaining.Seconds} secs");
                } else if (result.RequiresTwoFactor) {
                    return RedirectToPage("/SignInTwoFactor", new { returnUrl });
                } else if (result.IsNotAllowed) {
                    ModelState.AddModelError("", "Sign In Not Allowed");
                } else {
                    ModelState.AddModelError("", "Access Denied");
                }
                Username = username;
                ReturnUrl = returnUrl;
                return Page();
            }
            return Redirect(returnUrl ?? "/signin");
        }
    }
}
```

The page model class for this page reuses code that was previously part of the `SignIn` page and does not introduce any new features.

Adding a GET Handler Method for External Authentication

In Listing 23-19, I have added a GET handler method to the page model class for the `ExternalSignIn` Razor Page. This allows me to easily start the external authentication process by sending form data in a POST request or with a redirection, which uses a GET request.

Listing 23-19. Adding a Handler Method in the ExternalSignIn.cshtml.cs File in the Pages Folder

```
using ExampleApp.Identity;
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using System.Linq;
using System.Security.Claims;
using System.Threading.Tasks;
using SignInResult = Microsoft.AspNetCore.Identity.SignInResult;
```

```
namespace ExampleApp.Pages {

    public class ExternalSignInModel : PageModel {

        public ExternalSignInModel(SignInManager<AppUser> signInManager,
                UserManager<AppUser> userManager) {
            SignInManager = signInManager;
            UserManager = userManager;
        }

        public SignInManager<AppUser> SignInManager { get; set; }
        public UserManager<AppUser> UserManager { get; set; }

        public string ProviderDisplayName { get; set; }

        public IActionResult OnGet(string error, string providerName,
                string returnUrl)
            => error == null ? OnPost(providerName, returnUrl) : Page();

        public IActionResult OnPost(string providerName,
                string returnUrl = "/") {
            string redirectUrl = Url.Page("./ExternalSignIn",
                 pageHandler: "Correlate", values: new { returnUrl });
            AuthenticationProperties properties = SignInManager
              .ConfigureExternalAuthenticationProperties(providerName,
                  redirectUrl);
            return new ChallengeResult(providerName, properties);
        }

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

Care should be taken when using the handler method for one HTTP verb to call a handler for another because odd results can be produced. In this case, however, the POST handler method produces a challenge result that leads to a redirection, which presents no issues.

Restricting Additional External Authentication

The final step in the process is to change the correlation part of the external authentication process so that only one external authentication scheme can be used to sign in, as shown in Listing 23-20.

Listing 23-20. Preventing Additional Sign-Ins in the ExternalSignIn.cshtml.cs File in the Pages Folder

```
...
public async Task<IActionResult> OnGetCorrelate(string returnUrl) {
    ExternalLoginInfo info = await SignInManager.GetExternalLoginInfoAsync();
    AppUser user = await UserManager.FindByLoginAsync(info.LoginProvider,
        info.ProviderKey);
    if (user == null) {
        string externalEmail =
```

```
            info.Principal.FindFirst(ClaimTypes.Email)?.Value
                ?? string.Empty;
        user = await UserManager.FindByEmailAsync(externalEmail);
        if (user == null) {
            return RedirectToPage("/ExternalAccountConfirm",
                new { returnUrl });
        } else {
            UserLoginInfo firstLogin = user?.UserLogins?.FirstOrDefault();
            if (firstLogin != null
                    && firstLogin.LoginProvider != info.LoginProvider) {
                return RedirectToPage(
                    new {
                      error =
                        $"{firstLogin.ProviderDisplayName} Authentication Expected"
                    });
            } else {
                await UserManager.AddLoginAsync(user, info);
            }
        }
    }
    SignInResult result = await SignInManager.ExternalLoginSignInAsync(
            info.LoginProvider, info.ProviderKey, false, false);
    if (result.Succeeded) {
        return RedirectToPage("ExternalSignIn", "Confirm",
            new { info.ProviderDisplayName, returnUrl });
    } else if (result.RequiresTwoFactor) {
        string postSignInUrl = this.Url.Page("/ExternalSignIn", "Confirm",
            new { info.ProviderDisplayName, returnUrl });
        return RedirectToPage("/SignInTwoFactor",
            new { returnUrl = postSignInUrl });
    }
    return RedirectToPage(new { error = true, returnUrl });
}
...
```

An error message is displayed if the user signs in using the wrong external authentication service. To see the revised sequence, restart ASP.NET Core, request `http://localhost:5000/signout`, and click the Sign Out button to remove existing authentication cookies.

Request `http://localhost:5000/signin` and click the Google button. Go through the process of signing in and creating an Identity account. Request `http://localhost:5000/signin` again, but this time click the Facebook button. Sign in with an account that has the same email address as your Google account, and you will see the error shown in Figure 23-11.

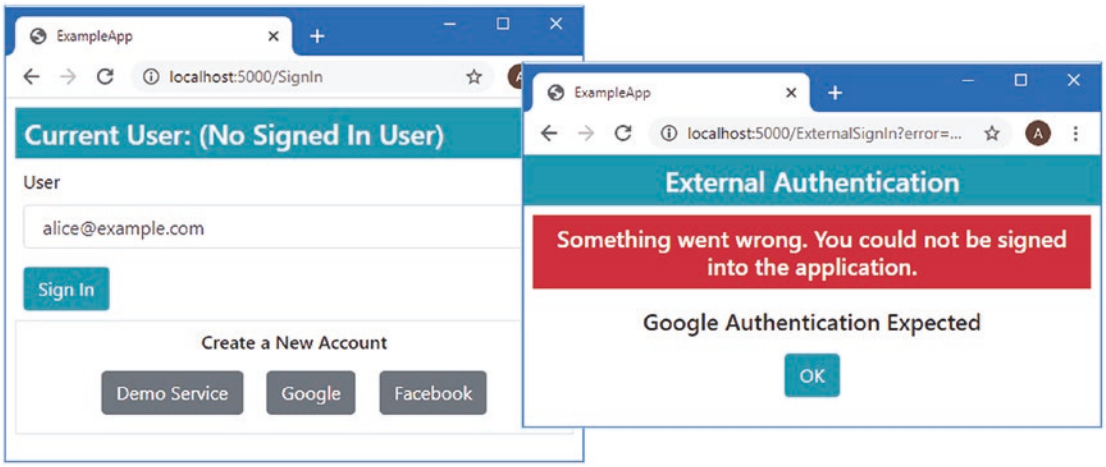

Figure 23-11. *Restricting signing in with external authentication*

Summary

In this chapter, I showed you how to store authentication tokens, which can be used with external services and built on the foundation from Chapter 22 to implement custom authentication support for the services provided by Google and Facebook.

And that's all I have to teach you about ASP.NET Core Identity. I can only hope that you have enjoyed reading this book as much as I enjoyed writing it, and I wish you every success in your ASP.NET Core and ASP.NET Core Identity projects.