CHAPTER 22

External Authentication, Part 1

In this chapter, I explain how external authentication works. This is a complex process, even by the high standard set by other ASP.NET Core Identity features, and there is a lot of detail to take in. The examples in this chapter use a simulated third-party service, which allows me to explain the interactions in a controlled way. In Chapter 23, I add support for working with real services from Google and Facebook. Table 22-1 puts external authentication in context.

Table 22-1. *Putting External Authentication in Context*

Question	Answer
What is it?	External authentication delegates the process of identifying users to a third-party service, typically provided by a major technology company or social media platform.
Why is it useful?	External authentication allows users to sign in to applications without having to create and remember an additional password. It can also provide access to stronger sign-in security than is supported directly by ASP.NET Core Identity.
How is it used?	The process is complicated, but the user's browser is redirected to the external service, which authenticates the user and communicates the result to the ASP.NET Core application. The result is verified by ASP.NET Core directly with the external service, and the user is signed in.
Are there any pitfalls or limitations?	The main issue is complexity, although this is not as much of an issue if you use the external authentication packages provided by Microsoft, which are described in Part 1.
Are there any alternatives?	External authentication is an optional feature, and projects do not have to use it.

Preparing for This Chapter

This chapter uses the ExampleApp project from Chapter 21. No changes are required for this chapter. Open a new command prompt, navigate to the `ExampleApp` folder, and run the command shown in Listing 22-1 to start ASP.NET Core.

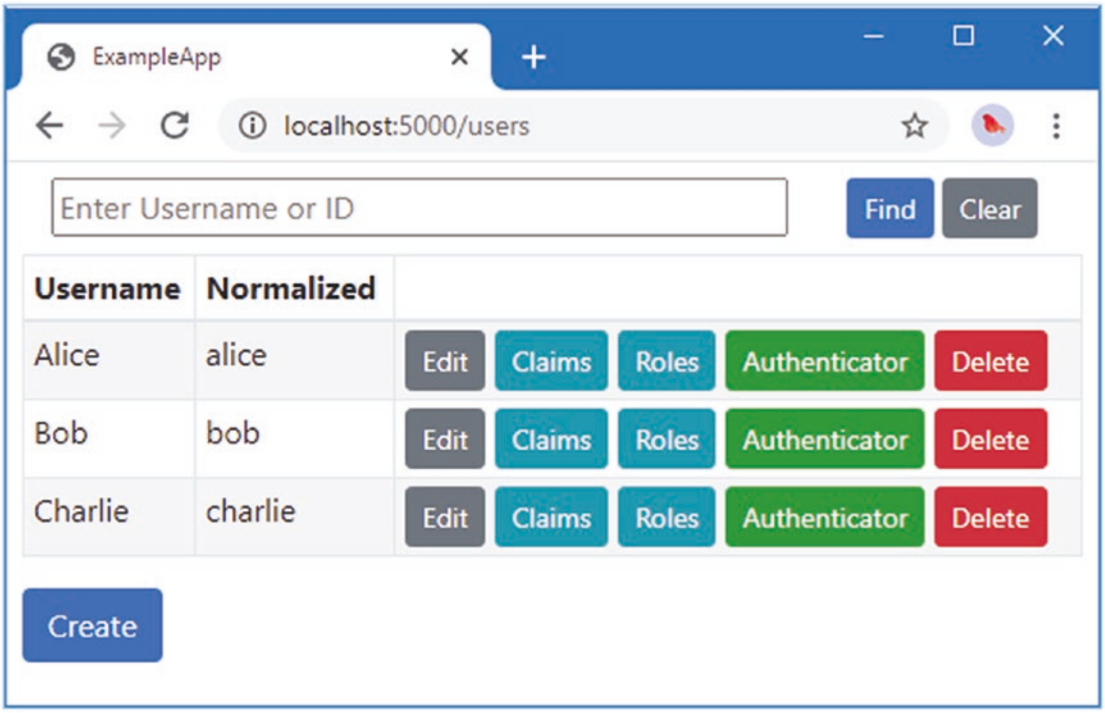
Tip You can download the example project for this chapter—and for all the other chapters in this book—from `https://github.com/Apress/pro-asp.net-core-identity`. See Chapter 1 for how to get help if you have problems running the examples.

A. Freeman, *Pro ASP.NET Core Identity*, https://doi.org/10.1007/978-1-4842-6858-2_22

Listing 22-1. Running the Example Application

```
dotnet run
```

Open a new browser window and request `http://localhost:5000/users`. You will be presented with the user data shown in Figure 22-1. The data is stored only in memory and changes will be lost when ASP.NET Core is stopped.



Username	Normalized
Alice	alice
Bob	bob
Charlie	charlie

Figure 22-1. *Running the example application*

Preparing for External Authentication

When external authentication is used, the process of obtaining and validating a user's credentials is handled outside of the ASP.NET Core application (hence the term *external*). For corporate applications, this typically means that authentication is delegated to a service that is shared between multiple applications, allowing users to sign into several applications with a single set of credentials. In customer-facing applications, authentication is often handled by social media platforms, such as Google or Facebook, where users are likely to already have accounts set up.

Identity doesn't apply any constraints on how external authentication is performed and relies on an authentication handler to take care of the details. Authentication handlers for external services implement the same `IAuthenticationHandler` interface I described in Chapter 14. This approach is consistent with the basic Identity features and means that an application can support multiple external services by having an authentication handler for each one. External authentication has three phases, *preparation*, *authentication*, and *correlation*, which I describe in the following text.

Authentication handlers are part of the ASP.NET Core platform and are not specific to Identity. To help keep the focus on Identity features, I am going to create an authentication handler that returns canned results, without actually performing authentication. In later examples, I replace the canned results, initially to access a test authentication service and then to perform authentication with real social media platforms. Add a class file named `ExternalAuthHandler.cs` to the `ExampleApp/Custom` folder and use it to define the class shown in Listing 22-2.

Listing 22-2. The Contents of the ExternalAuthHandler.cs File in the Custom Folder

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

namespace ExampleApp.Custom {
    public class ExternalAuthHandler : IAuthenticationHandler {

        public AuthenticationScheme Scheme { get; set; }
        public HttpContext Context { get; set; }

        public Task InitializeAsync(AuthenticationScheme scheme,
                HttpContext context) {
            Scheme = scheme;
            Context = context;
            return Task.CompletedTask;
        }

        public Task<AuthenticateResult> AuthenticateAsync() {
            return Task.FromResult(AuthenticateResult.NoResult());
        }

        public Task ChallengeAsync(AuthenticationProperties properties) {
            return Task.CompletedTask;
        }

        public Task ForbidAsync(AuthenticationProperties properties) {
            return Task.CompletedTask;
        }
    }
}
```

This is a minimal implementation of the `IAuthenticationHandler` interface that contains just enough code to compile. I add features to the class as I explain how the external authentication process works in the sections that follow.

Implementing the Selection Phase

The external authentication process presents the user with the option to authenticate using an external service. The `SignInManager<T>` class provides access to the authentication schemes available for external service through the method described in Table 22-2.

Table 22-2. *The SignInManager<T> Method for External Authentication Schemes*

Name	Description
`GetExternalAuthenticationSchemesAsync()`	This asynchronous method returns a sequence of `AuthenticationScheme` objects, each of which can be used for external authentication.

There is an oddity in the `GetExternalAuthenticationSchemesAsync` method, which is that only authentication schemes that have been registered with a `DisplayName` property are returned. In Listing 22-3, I have created a new scheme, with a `DisplayName`, that uses the handler created in Listing 22-2.

■ **Tip** The second argument to the `AddScheme<T>` method is optional and can be omitted if you need to create a scheme that should not be selected by the `GetExternalAuthenticationSchemesAsync` method.

Listing 22-3. Defining an Authentication Scheme in the Startup.cs File in the ExampleApp Folder

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

In Listing 22-4, I have added content to the view section of the `SignIn` Razor Page that presents the user with a button for each of the external authentication schemes.

Listing 22-4. Offering External Authentication in 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-6 border p-2 h-100">
            <h4 class="text-center">Local Authentication</h4>
            <form method="post">
                <div class="form-group">
                    <label>User</label>
                    <select class="form-control"
                            asp-for="Username" asp-items="@Model.Users">
                    </select>
                </div>
                <div class="form-group">
                    <label>Password</label>
                    <input class="form-control" type="password"
                         name="password" value="MySecret1$" />
                </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>
        </div>
        <div class="col-6 text-center">
            <div class="border p-2 h-100">
                <form method="post">
                    <h4>External Authentication</h4>
                    <div class="mt-4 w-75">
                        @foreach (var scheme in
                                await Model.SignInManager
                                    .GetExternalAuthenticationSchemesAsync()) {
                            <div class="mt-2 text-center">
                                <button class="btn btn-block btn-secondary
                                            m-1 mx-5" type="submit"
                                        asp-page="/externalsignin"
                                        asp-route-returnUrl=
                                            "@Request.Query["returnUrl"]"
                                        asp-route-providername="@scheme.Name">
                                    @scheme.DisplayName
                                </button>
                            </div>
                        }
                    </div>
                </form>
            </div>
        </div>
    </div>
</div>
```

The new layout separates the local authentication options from the buttons that will lead to external authentication, which you can see if you restart ASP.NET Core and request `http://localhost:5000/signin`, as shown in Figure 22-2. Clicking the Demo Service button targets a page that I will create in the next step.

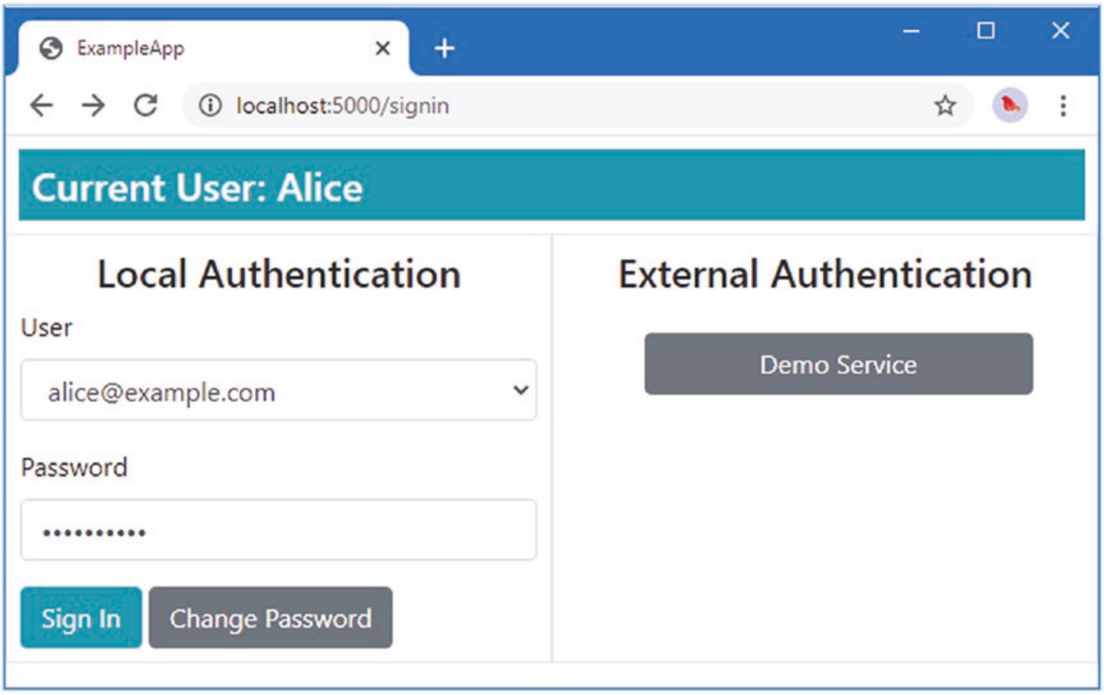


Figure 22-2. *Presenting the user with external authentication options*

Understanding the Preparation Phase

In the preparation phase, the application configures external authentication and starts the authentication process. To configure authentication, an `AuthenticationProperties` object is created, which is used to store state data that will be needed later. Table 22-3 describes the most important properties defined by the `AuthenticationProperties` class.

Table 22-3. *The Most Important AuthenticationProperties Properties*

Name	Description
`Items`	This property is used to store authentication state data using an `IDictionary<string, string?>` object.
`RedirectUri`	This property specifies the URL to which the user's browser should be redirected at the end of the authentication phase.

The Items collection property is used by the SignInManager<T> class to store data that it needs once the authentication phase is complete. The RedirectUri specifies the URL to which the authentication handler should redirect the user's browser once they are authenticated and that will be responsible for performing the correlation phase.

The SignInManager<T> class provides the ConfigureExternalAuthenticationProperties method for creating the AuthenticationProperties object, as described in Table 22-4.

Table 22-4. *The SignInManager<T> Method for the Preparation Phase*

Name	Description
`ConfigureExternalAuthenticationProperties(provider, redirectUrl, userId)`	This method creates an AuthenticationProperties object. The provider value is added to the Items collection, and the redirectUrl value is assigned to the RedirectUri property. The optional userId parameter is added to the Items collection as an anti-cross-site request forgery measure and is used when a user is already signed in.

The Razor Page or controller returns a ChallengeResult, which selects the authentication handler and the AuthenticationProperties object created by the ConfigureExternalAuthenticationProperties method. This results in the authentication handler's ChallengeAsync method being called using the AuthenticationProperties object as an argument, ending the preparation phase.

Add a Razor Page named ExternalSignIn.cshtml to the Pages folder with the content shown in Listing 22-5.

Listing 22-5. The Contents of the ExternalSignIn.cshtml File in the Pages Folder

```
@page
@model ExampleApp.Pages.ExternalSignInModel

<h4 class="bg-info text-white text-center p-2">External Authentication</h4>
```

This is just placeholder content for the moment because the view part of the page isn't required until later. For the moment, the focus is on the page model class. Add the code shown in Listing 22-6 to the ExternalSignIn.cshtml.cs file in the Pages folder. (You will have to create this file if you are using Visual Studio Code.)

Listing 22-6. The Contents of 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;

namespace ExampleApp.Pages {

    public class ExternalSignInModel : PageModel {

        public ExternalSignInModel(SignInManager<AppUser> signInManager) {
            SignInManager = signInManager;
        }
        public SignInManager<AppUser> SignInManager { get; set; }

        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);
        }
    }
}
```

There two levels of redirection to be handled when preparing for external authentication. The standard sign-in process captures the URL that the user requested that led to the challenge response. This value is added as a query string parameter of the URL to which the authentication handler should redirect the browser after the user has been authenticated, like this:

```
...
string redirectUrl = Url.Page("./ExternalSignIn", pageHandler: "Correlate",
    values: new { returnUrl });
...
```

The pageHandler value includes the name of the hander method that will receive the redirected request. I have named the method Correlate in this example, and I will define this method shortly.

Understanding the Authentication Phase

In the authentication phase, the authentication handler is responsible for guiding the user through the process required by the external service. The details of this process are not visible to ASP.NET Core Identity, but most external authentication is done using the OAuth protocol, which I describe later in this chapter. For now, however, I am going to authenticate the user without requiring any credentials to demonstrate the overall process. Add the code shown in Listing 22-7 to the ExternalAuthHandler class.

Listing 22-7. Performing Authentication in the ExternalAuthHandler.cs File in the Custom Folder

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

namespace ExampleApp.Custom {
    public class ExternalAuthHandler : IAuthenticationHandler {

        public AuthenticationScheme Scheme { get; set; }
        public HttpContext Context { get; set; }
        public Task InitializeAsync(AuthenticationScheme scheme,
                HttpContext context) {
            Scheme = scheme;
            Context = context;
            return Task.CompletedTask;
        }

        public Task<AuthenticateResult> AuthenticateAsync() {
            return Task.FromResult(AuthenticateResult.NoResult());
        }

        public async Task ChallengeAsync(AuthenticationProperties properties) {
            ClaimsIdentity identity = new ClaimsIdentity(Scheme.Name);
            identity.AddClaims(new[] {
                new Claim(ClaimTypes.NameIdentifier, "SomeUniqueID"),
                new Claim(ClaimTypes.Email, "alice@example.com"),
                new Claim(ClaimTypes.Name, "Alice")
            });
            ClaimsPrincipal principal = new ClaimsPrincipal(identity);
            await Context.SignInAsync(IdentityConstants.ExternalScheme,
                principal, properties);
            Context.Response.Redirect(properties.RedirectUri);
        }

        public Task ForbidAsync(AuthenticationProperties properties) {
            return Task.CompletedTask;
        }
    }
}
```

It is the outcome from the ChallengeAsync method that is important, even though the handler isn't really performing authentication. A ClaimsPrincipal object is created, with a ClaimsIdentity that describes the authenticated user from the perspective of the external authentication service. For this test handler, this means there are claims to provide a unique ID, an email address, and a name.

Once the handler has created the ClaimsPrincipal, the HttpContext.SignInAsync method is used to sign in the external user with a special scheme, like this:

```
...
await Context.SignInAsync(IdentityConstants.ExternalScheme,
    principal, properties);
...
```

The IdentityConstants.ExternalScheme is used to sign in the external user to prepare for the next phase in the process. The other arguments to the SignInAsync method are the ClaimsPrincipal object and the AuthenticationProperties object, which ensures that the state data received by the handler is preserved. Once the external user has been signed in, the handler issues a redirection to the URL specified by the AuthenticationProperties parameter's RedirectUri method.

```
...
Context.Response.Redirect(properties.RedirectUri);
...
```

To support the IdentityConstants.ExternalScheme scheme, add the statement shown in Listing 22-8 to the Startup class.

Listing 22-8. Adding the Cookie Handler in the Startup.cs File in the ExampleApp Folder

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

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

All external authentication results in a sign-in using IdentityConstants.ExternalScheme, regardless of which external service is used. This allows SignInManager<T> to get the external user details, as described in the next section.

Understanding the Correlation Phase

The correlation phase determines which Identity user account is associated with the external authentication. So, for example, if Alice authenticates using Google, the correlation process uses the claims that the handler associated with the ClaimsPrincipal object to determine that this login is related to the local Alice account.

The way external logins are correlated to local accounts can be adapted to each project and each external authentication service. For the example app, I am going to use a three-stage approach.

1. Check the user store to see if the external login has already been associated with a local account. If it has been, sign the local user into the application.
2. If not, look for a local account with a matching email address. If there is a match, associate the login in the store and sign the user into the application.
3. If there is no matching email address, prompt the user to create a new account.

The external authentication handler defined in Listing 22-8 authenticates only a single user, so I will leave the third part of the process until later in the chapter, where I introduce a wider range of authentication features.

The approach I have outlined relies on the ability to keep track of user logins, which is done by storing UserLoginInfo objects in the user store. The UserLoginInfo class defines the properties shown in Table 22-5.

***Table 22-5.** The UserLoginInfo Properties*

Name	Description
`LoginProvider`	This is the name of the authentication handler for the external service.
`ProviderKey`	This is the unique identifier by which the external service recognizes the user.
`ProviderDisplayName`	This is the name of the external service that will be displayed to the user.

Identity only needs to keep track of the unique ID by which the user is known to the external service and details of the authentication service (which are generally the same as the authentication handler details).

Extending the User Store

User stores that can store external logins implement the `IUserLoginStore<T>` interface, which defines the methods described in Table 22-6. (Like the other store interfaces, these methods define a `CancellationToken` parameter named `token`.)

***Table 22-6.** The IUserLoginStore<T> Methods*

Name	Description
`GetLoginsAsync(user, token)`	This method returns an `IList<UserLoginInfo>` containing the external logins for the specified user.
`AddLoginAsync(user, login, token)`	This method stores a `UserLoginInfo` for the specified user.
`RemoveLoginAsync(user, loginProvider, providerKey)`	This method removes the `UserLoginInfo` with the specified provider and key.
`FindByLoginAsync(loginProvider, providerKey)`	This method locates the user who has a `UserLoginInfo` with the specified provider and key.

To add support for storing login information, add the property shown in Listing 22-9 to the `AppUser` class.

Listing 22-9. 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;

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; }
    }
}
```

To implement the interface in the example store, add a class file named `UserStoreLogins.cs` to the `ExampleApp/Identity/Store` folder and use it to define the partial class shown in Listing 22-10.

Listing 22-10. The Contents of the UserStoreLogins.cs File in the Identity/Store Folder

```
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 : IUserLoginStore<AppUser> {

        public Task<IList<UserLoginInfo>> GetLoginsAsync(AppUser user,
                CancellationToken token)
            => Task.FromResult(user.UserLogins ?? new List<UserLoginInfo>());

        public Task AddLoginAsync(AppUser user, UserLoginInfo login,
                CancellationToken token) {
            if (user.UserLogins == null) {
                user.UserLogins = new List<UserLoginInfo>();
            }
            user.UserLogins.Add(login);
            return Task.CompletedTask;
        }

        public async Task RemoveLoginAsync(AppUser user, string loginProvider,
                string providerKey, CancellationToken token)
            => user.UserLogins = (await GetLoginsAsync(user, token)).Where(login
                => !login.LoginProvider.Equals(loginProvider)
                       && !login.ProviderKey.Equals(providerKey)).ToList();

        public Task<AppUser> FindByLoginAsync(string loginProvider,
                string providerKey, CancellationToken token) =>
            Task.FromResult(Users.FirstOrDefault(u => u.UserLogins != null &&
                u.UserLogins.Any(login => login.LoginProvider.Equals(loginProvider)
                    && login.ProviderKey.Equals(providerKey))));
    }
}
```

Since the user store in the example application is memory-based, I can keep track of the `UserLoginInfo` objects using a dictionary, with `AppUser` objects as keys. The `UserManager<T>` class defines the methods shown in Table 22-7 for managing the external logins in the user store.

Table 22-7. *The UserManager<T> Methods for Managing External Logins*

Name	Description
`FindByLoginAsync(provider, key)`	This method calls the user store's `FindByLoginAsync` method to locate the local user associated with the specified provider and key.
`GetLoginsAsync(user)`	This method calls the user store's `GetLoginsAsync` method to return an `IList<UserLoginInfo>` containing the external logins associated with the specified local user.
`AddLoginAsync(user, login)`	This method calls the user store's `AddLoginAsync` method to associate the specified external login with the local user, after which the user manager's update sequence is performed. An exception is thrown if it already contains the specified login (which is determined by calling the `FindByLoginAsync` method).
`RemoveLoginAsync(user, provider, key)`	This method calls the store's `RemoveLoginAsync` method to remove the external login, after which the user's security stamp is updated and the user manager's update sequence is performed.

Correlating and Storing Logins

The `SignInManager<T>` class defines the methods shown in Table 22-8 for obtaining the `ClaimsPrincipal` created by the external authentication handler and signing in local users.

Table 22-8. *The SignInManager<T> Methods for Correlating External Logins*

Name	Description
`GetExternalLoginInfoAsync()`	This method returns an `ExternalLoginInfo` object for the external login that can be added to the user store. It does this by retrieving the `ClaimsPrincipal` object created by the external authentication handler. The value of the `NameIdentifier` claims is used as a unique ID. The provider name is obtained from a property named `LoginProvider` in the `Items` collection of the `AuthentionProperties` object associated with the login. The display name is obtained from the results of the `GetExternalAuthenticationSchemesAsync` method.
`ExternalLoginSignInAsync (provider, key, isPersistent, bypassTwoFactor)`	This method uses the `FindByLoginAsync` method to locate the user with a login that matches the specified provider and key and signs them into the application. The `isPersisent` argument controls whether the sign-in cookie persists after the browser is closed. The `bypassTwoFactor` argument determines if the two-factor feature will be bypassed.

The login correlation is performed when the authentication handler redirects the browser following the completion of the authentication phase. The `GetExternalLoginInfoAsync` method returns an instance of the `ExternalLoginInfo` class, which is derived from `UserLoginInfo` and defines the additional properties described in Table 22-9.

Table 22-9. *The ExternalLoginInfo Properties*

Name	Description
`Principal`	This property returns the `ClaimsPrincipal` object created by the external authentication handler.
`AuthenticationTokens`	This property returns a sequence of authentication tokens, which I describe in Chapter 23.
`AuthenticationProperties`	This property returns the `AuthenticationProperties` object associated with the external login.

Add the code shown in Listing 22-11 to define the handler methods that deal with the correlation in the example application.

Listing 22-11. Processing the External Login 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 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);
        }

        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 {
                    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 });
        }

        public async Task OnGetConfirmAsync() {
            string provider = User.FindFirstValue(ClaimTypes.AuthenticationMethod);
            ProviderDisplayName =
                (await SignInManager.GetExternalAuthenticationSchemesAsync())
                   .FirstOrDefault(s => s.Name == provider)?.DisplayName ?? provider;
        }
    }
}
```

When the OnGetCorrelate method is called, I use the GetExternalLoginInfoAsync method to get the ExternalLoginInfo object that describes the external login. Using the details provided by the ExternalLoginInfo, I query the user store for an existing external login to get the AppUser object that represents the user.

If there is no matching login, I locate an email claim from the external ClaimsPrincipal and use it to query the user store. If there is a matching user, I store the external login for future use.

If I have found a user, either using the login or by email address, I sign them into the application using the SignInManager<T>.ExternalLoginSignInAsync method, like this:

```
...
SignInResult result = await SignInManager.ExternalLoginSignInAsync(
    info.LoginProvider, info.ProviderKey, false, false);
...
```

There is an option to bypass two-factor security when signing in with an external login, but I have chosen to leave two-factor enabled to demonstrate how it works. If the user is signed in without two-factor, I redirect them to the Confirm handler, which uses the view part of the page to display a summary of the external login.

```
...
string postSignInUrl = this.Url.Page("/ExternalSignIn", "Confirm",
    new { info.ProviderDisplayName, returnUrl });
return RedirectToPage("/SignInTwoFactor", new { returnUrl = postSignInUrl });
...
```

To display a summary of the external authentication process, add the content shown in Listing 22-12 to the ExternalSignIn.cshtml file.

Listing 22-12. Displaying a Summary in the ExternalSignIn.cshtml File in the Pages Folder

```
@page
@model ExampleApp.Pages.ExternalSignInModel

<h4 class="bg-info text-white text-center p-2">External Authentication</h4>

@{
    string returnUrl = Request.Query["returnUrl"].Count == 0 ?
        "/" : Request.Query["returnUrl"];
}
```

```
@if (Request.Query["error"].Count() > 0) {
    <h5 class="bg-danger text-white text-center m-2 p-2">
        Something went wrong. You could not be signed into the application.
    </h5>
    <h5 class="text-center m-2 p-2">@Request.Query["error"]</h5>
    <div class="text-center">
        <a class="btn btn-info text-center" href="@returnUrl">OK</a>
    </div>
} else {
    <h5 class="text-center">
        @User.Identity.Name has been authenticated by @Model.ProviderDisplayName
    </h5>

    <div class="text-center">
        <a class="btn btn-info text-center" href="@returnUrl">Continue</a>
    </div>
}
```

I added some basic error handling to the `ExternalSignIn` page, which I will use once I have added a broader range of external authentication features.

Restart ASP.NET Core, request `http://localhost:5000/signout`, select the Forget Me option, and click the Sign Out button to sign out of the application. Request `http://localhost:5000/secret` to trigger the challenge response. Click the Demo Service button to authenticate using the external handler. At present, the handler will immediately authenticate the request for the Alice user and redirect the browser to the `ExternalSignIn` Razor Page. (I will expand the features of the external handler so that it prompts for a password.) The application uses the email address it receives from the external authentication handler to correlate the sign-in with the local Alice user account. This account is configured for two-factor authentication with an authenticator, and you will be prompted to enter the current authenticator code. Enter the authenticator code (or use a recovery code), and a summary of the external login will be displayed. Click the OK button, and the browser will be redirected to the protected content, as shown in Figure 22-3.

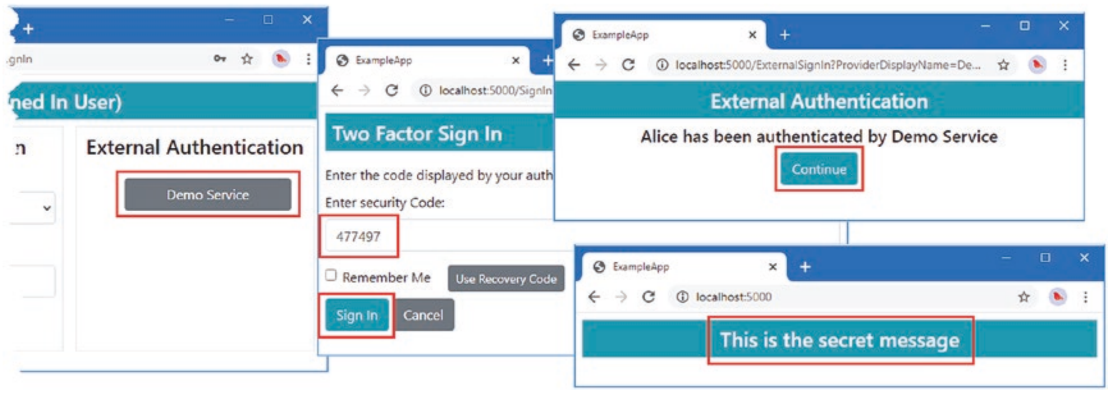


***Figure 22-3.** The external authentication workflow*

Understanding the OAuth Authentication Process

Identity places no restrictions on how external authentication is performed, but most services use the OAuth protocol. In the sections that follow, I explain each step in the OAuth authentication process and implement. In the sections that follow, I explain the authentication sequence and the data that is exchanged at each stage. Each authentication service is slightly different, but you can expect to encounter most of the details I describe in any service that uses OAuth, although you should also expect some variations for each provider, as I demonstrate in Chapter 23 when I add support for real services.

■ **Tip** You don't need to know the OAuth specification to follow the examples in this chapter, but you can dig into the details at `https://oauth.net` if you are interested.

Preparing for External Authentication

Before deployment, the application is registered with the authentication service. Registration requirements differ, but the result is two pieces of data used in the authentication process: the client ID and the client secret.

I demonstrate how these data items are used in detail in the sections that follow, but the client ID can be shared publicly and is just used to identify which application authentication requests are for. The client secret, as the name suggests, should not be shared publicly and is sent only in requests to the authentication service. Table 22-10 describes these data items for quick reference.

Table 22-10. *The Data Items Created During Application Registration*

Name	Description
Client ID	The client ID is included in requests to let the authentication service know which application wants to authenticate a user. This allows the authentication service to present the user with details about the application so they can make an informed choice about whether to provide access to their data.
Client secret	The client secret is included in requests to prove to the authentication service that they originate from the application. This relies on the secret being kept confidential, which means it should not be shared with users and should not be included in public source code repositories.

Preparing the Simulated External Authentication Controller

For simplicity, I am going to use a controller in the existing ExampleApp project to represent the external authentication service. This will allow me to demonstrate the use of HTTP requests in the external authentication process without needing to create and run a separate server. Add a class file named `DemoExternalAuthController.cs` to the `ExampleApp/Controllers` folder and add the code shown in Listing 22-13.

Listing 22-13. The Contents of the DemoExternalAuthController.cs File in the Controllers Folder

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

namespace ExampleApp.Controllers {
    class UserRecord {
        public string Id { get; set; }
        public string Name { get; set; }
        public string EmailAddress { get; set; }
        public string Password { get; set; }
        public string Code { get; set; }
        public string Token { get; set; }
    }

    public class DemoExternalAuthController : Controller {
        private static string expectedID = "MyClientID";
        private static string expectedSecret = "MyClientSecret";
        private static List<UserRecord> users  = new List<UserRecord> {
            new UserRecord() {
                Id = "1", Name = "Alice", EmailAddress = "alice@example.com",
                Password = "myexternalpassword"
            },
             new UserRecord {
                Id = "2", Name = "Dora", EmailAddress = "dora@example.com",
                Password = "myexternalpassword"
             }
        };
    }
}
```

The controller defines fields that specify the expected values for the client ID and client secret. In a real external service, each application that has been registered will have an ID and secret, but I need only one set of values to demonstrate the authentication sequence in the example application. The controller in Listing 22-13 also defines some basic user data that will be used in the authentication process.

Preparing the Authentication Handler

Make the change shown in Listing 22-14 to the `ExternalAuthHandler` class to prepare for authentication using the controller created in the previous section.

Listing 22-14. Preparing the Handler in the ExternalAuthHandler.cs File in the Custom Folder

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

namespace ExampleApp.Custom {

    public class ExternalAuthOptions {
        public string ClientId { get; set; } = "MyClientID";
        public string ClientSecret { get; set; } = "MyClientSecret";
    }
    public class ExternalAuthHandler : IAuthenticationHandler {

        public ExternalAuthHandler(IOptions<ExternalAuthOptions> options) {
            Options = options.Value;
        }

        public AuthenticationScheme Scheme { get; set; }
        public HttpContext Context { get; set; }

        public ExternalAuthOptions Options { get; set; }

        public Task InitializeAsync(AuthenticationScheme scheme,
                HttpContext context) {
            Scheme = scheme;
            Context = context;
            return Task.CompletedTask;
        }

        public Task<AuthenticateResult> AuthenticateAsync() {
            return Task.FromResult(AuthenticateResult.NoResult());
        }

        public async Task ChallengeAsync(AuthenticationProperties properties) {

            // TODO - authentication implementation
        }

        public Task ForbidAsync(AuthenticationProperties properties) {
            return Task.CompletedTask;
        }
    }
}
```

I have removed the test code I used earlier in the chapter and added support for receiving configuration settings using the options pattern on the ExternalAuthOptions class. Add the statement shown in Listing 22-15 to the Startup class to apply the options pattern, which allows easy configuration changes.

Listing 22-15. Applying the Options Pattern 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.AddIdentityCore<AppUser>(opts => {
        opts.Tokens.EmailConfirmationTokenProvider = "SimpleEmail";
        opts.Tokens.ChangeEmailTokenProvider = "SimpleEmail";
        opts.Tokens.PasswordResetTokenProvider =
            TokenOptions.DefaultPhoneProvider;
...
```

Step 1: Redirecting to the Authentication Service URL

The process starts when the user clicks the button to start the authentication process. The request causes ASP.NET Core to ask the authentication handler class to produce a challenge response. The handler responds by redirecting the user's browser to a URL provided by the authentication service, as shown in Figure 22-4.

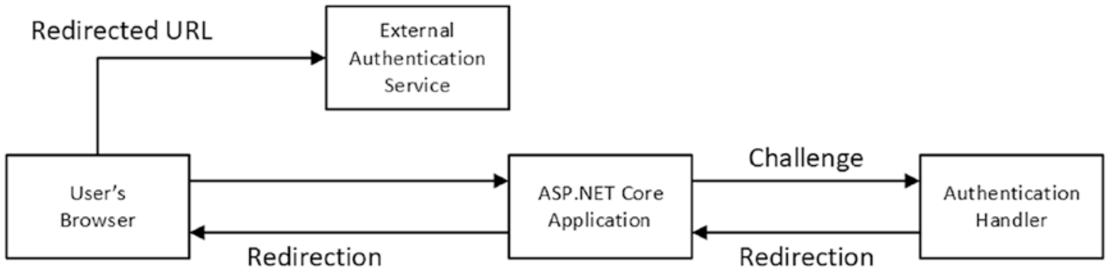


Figure 22-4. *Redirecting the user's browser to the external authentication service*

The handler uses the query string of the redirection URL to convey information to the authentication service via the user's browser: the client ID, the return URL, the scope, and an optional piece of state data. As an example, the query string of the URL to which the browser is redirected has this structure, where real values are substituted for the placeholder that I have denoted with the < and > characters:

```
?client_id=<Client ID>&redirect_uri=<Return URL>&scope=<Scope>&state=<State Data>
```

As explained in the previous section, the client ID is created when the application is registered with the authentication service and identifies the requests that are from a specific application. This value is sent using the `client_id` parameter.

The return URL tells the authentication service where to redirect the client once the user has been authenticated, which I describe in the next section. The return URL is sent using the `redirect_uri` parameter.

The scope describes the data that the application requires for the user. The most popular authentication services are provided by companies like Google and Facebook, whose services combine basic authentication and access to more complex APIs so that applications can get user data, such as messages and calendar appointments. Applications include a scope string in the redirection URL to specify the access they require, which allows the authentication service to ask the user if they consent to access. The scope is sent using the `scope` query string parameter. Each authentication service defines scopes, and some services require applications to declare the scope they require during registration.

Finally, state data is included in the URL so that the application can keep track of the authentication process. Not all applications need state data because they can keep track of which user the authentication relates to by adding cookies to the responses sent to the user's browser, but the OAuth specification recommends that a value should still be included in the request as protection against cross-site request forgery (CSRF) attacks. The state data is sent using the `state` query string parameter.

Table 22-11 summarizes the query string parameters the application includes in the redirection URL for quick reference.

Table 22-11. *The Query String Parameters Included in the Redirection URL*

Name	Description
`client_id`	This parameter sends the client ID, which identifies the application to the authentication service, as described in the previous section.
`redirect_uri`	This parameter sends the URL to which the authentication service should redirect the user's browser once authentication is complete.
`scope`	This parameter sends the scope, which specifies the data and services that the application requires.
`state`	This parameter sends a state data value, which is used by the application to correlate related requests and to protect against cross-site forgery attacks.
`response_type`	This parameter specifies the type of response and must be set to `code`.

Updating the External Authentication Controller

In Listing 22-16, I have added an action method that simulates an external service. This action will be the target of the redirection.

Listing 22-16. Adding an Action in the DemoExternalAuthController.cs File in the Controllers Folder

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

namespace ExampleApp.Controllers {

    class UserRecord {
        public string Id { get; set; }
        public string Name { get; set; }
        public string EmailAddress { get; set; }
        public string Password { get; set; }
        public string Code { get; set; }
        public string Token { get; set; }
    }

    public class ExternalAuthInfo {
        public string client_id { get; set; }
        public string client_secret { get; set; }
        public string redirect_uri { get; set; }
        public string scope { get; set; }
        public string state { get; set; }
        public string response_type { get; set; }
        public string grant_type { get; set; }
        public string code { get; set; }
    }
    public class DemoExternalAuthController : Controller {
        private static string expectedID = "MyClientID";
        private static string expectedSecret = "MyClientSecret";
        private static List<UserRecord> users  = new List<UserRecord> {
            new UserRecord() {
                Id = "1", Name = "Alice", EmailAddress = "alice@example.com",
                Password = "myexternalpassword"
            },
             new UserRecord {
                Id = "2", Name = "Dora", EmailAddress = "dora@example.com",
                Password = "myexternalpassword"
             }
        };

        public IActionResult Authenticate([FromQuery] ExternalAuthInfo info)
         => expectedID == info.client_id ? View((info, string.Empty))
                : View((info, "Unknown Client"));
    }
}
```

I have defined a class named ExternalAuthInfo to make it easier to send and receive the data that the authentication process requires. The new action method, named Authenticate, renders a view to prompt the user for their credentials. The same view is used to display an error message if the request doesn't contain the expected client ID. Notice that any errors are this stage are displayed to the user and are not communicated to the application.

■ **Note** Most authentication services require redirect_uri URL to be registered in advance and will display an error if a different value is received in the request.

To define the view used by the Authenticate action, create the Views/DemoExternalAuth folder and add to it a Razor View named Authenticate.cshtml with the content shown in Listing 22-17.

Listing 22-17. The Contents of the Authenticate.cshtml File in the Views/DemoExternalAuth Folder

```
@model (ExampleApp.Controllers.ExternalAuthInfo info, string error)

@{
    IEnumerable<(string, string)> KeyValuePairs =
        typeof(ExampleApp.Controllers.ExternalAuthInfo).GetProperties()
            .Select(pi => (pi.Name, pi.GetValue(Model.info)?.ToString()));
}

<div class="bg-dark text-white p-2">
    <h4 class="text-center">Demo External Authentication Service</h4>
    <div class="bg-light text-dark m-4 p-5 border">

        @if (!string.IsNullOrEmpty(Model.error)) {
            <div class="h3 bg-danger text-white text-center m-2 p-2">
```

```
                <div>Something Went Wrong</div>
                <div class="h5">(@Model.error)</div>
            </div>
        } else {
            <div asp-validation-summary="All" class="text-danger m-2"></div>
            <form method="post" asp-action="Authenticate">
                @foreach (var tuple in KeyValuePairs) {
                    if (!string.IsNullOrEmpty(tuple.Item2)) {
                        <input type="hidden" name="@tuple.Item1"
                            value="@tuple.Item2" />
                    }
                }
                <div class="p-2">
                    <div class="form-group">
                        <label>Email</label>
                        <input name="email" class="form-control" />
                    </div>
                    <div class="form-group">
                        <label>Password</label>
                        <input name="password" type="password"
                            class="form-control" />
                    </div>
                    <button type="submit" class="btn btn-sm btn-dark">
                        Authenticate & Return
                    </button>
                </div>
            </form>
        }
    </div>
</div>
```

This view simulates the response from the external authentication service and contains a form that submits the user's credentials, along with hidden input elements that contain the values provided by the application in the redirection request.

Updating the Authentication Handler

In Listing 22-18, I have revised the authentication handler so that it sends the redirection when asked to issue a challenge response.

■ **Note** I am writing the authentication handler using protected virtual methods so that I can easily create subclasses to work with real authentication services in Chapter 23.

Listing 22-18. Performing the Redirection in the ExternalAuthHanlder.cs File in the Custom Folder

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;
using System.Security.Claims;
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.Options;
using System.Collections.Generic;
using Microsoft.AspNetCore.DataProtection;

namespace ExampleApp.Custom {

    public class ExternalAuthOptions {
        public string ClientId { get; set; } = "MyClientID";
        public string ClientSecret { get; set; } = "MyClientSecret";

        public virtual string RedirectRoot { get; set; } = "http://localhost:5000";
        public virtual string RedirectPath { get; set; } = "/signin-external";
        public virtual string Scope { get; set; } = "openid email profile";
        public virtual string StateHashSecret { get; set; } = "mysecret";

        public virtual string AuthenticationUrl { get; set; }
            = "http://localhost:5000/DemoExternalAuth/authenticate";
    }

    public class ExternalAuthHandler : IAuthenticationHandler {

        public ExternalAuthHandler(IOptions<ExternalAuthOptions> options,
                IDataProtectionProvider dp) {
            Options = options.Value;
            DataProtectionProvider = dp;
        }

        public AuthenticationScheme Scheme { get; set; }
        public HttpContext Context { get; set; }
        public ExternalAuthOptions Options { get; set; }
        public IDataProtectionProvider DataProtectionProvider { get; set; }
        public PropertiesDataFormat PropertiesFormatter { get; set; }

        public Task InitializeAsync(AuthenticationScheme scheme,
                HttpContext context) {
            Scheme = scheme;
            Context = context;
            PropertiesFormatter = new PropertiesDataFormat(DataProtectionProvider
                .CreateProtector(typeof(ExternalAuthOptions).FullName));
            return Task.CompletedTask;
        }

        public Task<AuthenticateResult> AuthenticateAsync() {
            return Task.FromResult(AuthenticateResult.NoResult());
        }

        public async Task ChallengeAsync(AuthenticationProperties properties) {
            Context.Response.Redirect(await GetAuthenticationUrl(properties));
        }
        protected virtual Task<string>
                GetAuthenticationUrl(AuthenticationProperties properties) {
            Dictionary<string, string> qs = new Dictionary<string, string>();
            qs.Add("client_id", Options.ClientId);
            qs.Add("redirect_uri", Options.RedirectRoot + Options.RedirectPath);
            qs.Add("scope", Options.Scope);
            qs.Add("response_type", "code");
            qs.Add("state", PropertiesFormatter.Protect(properties));
            return Task.FromResult(Options.AuthenticationUrl
                + QueryString.Create(qs));
        }

        public Task ForbidAsync(AuthenticationProperties properties) {
            return Task.CompletedTask;
        }
    }
}
```

The ChallengeAsync method now sends a redirection to the URL that will authenticate the user. To do this, I have defined additional configuration options. The RedirectUri option is used to specify the URL to which the authentication service will redirect the browser after authentication. For the example application, this will be http://localhost:5000/signin-external, and I explain how to handle requests sent to that URL in Step 3.

The Scope option uses a typical scope, although real services differ in the scope they expect, and the values are usually determined when registering an application for authentication. The value I used is the same as the one used by the Google authentication handler that Microsoft provides and that I demonstrated in Chapter 23.

The state value uses the ASP.NET Core data protection feature to securely include a serialized representation of the AuthenticationProperties object in the redirection URL. This data will be returned to the authentication handler later in the authentication process.

You can test the redirection by restart ASP.NET Core, requesting http://localhost:5000/signin, and clicking the Demo Service button. The response will redirect your browser to the (simulated) external authentication service, as shown in Figure 22-5.

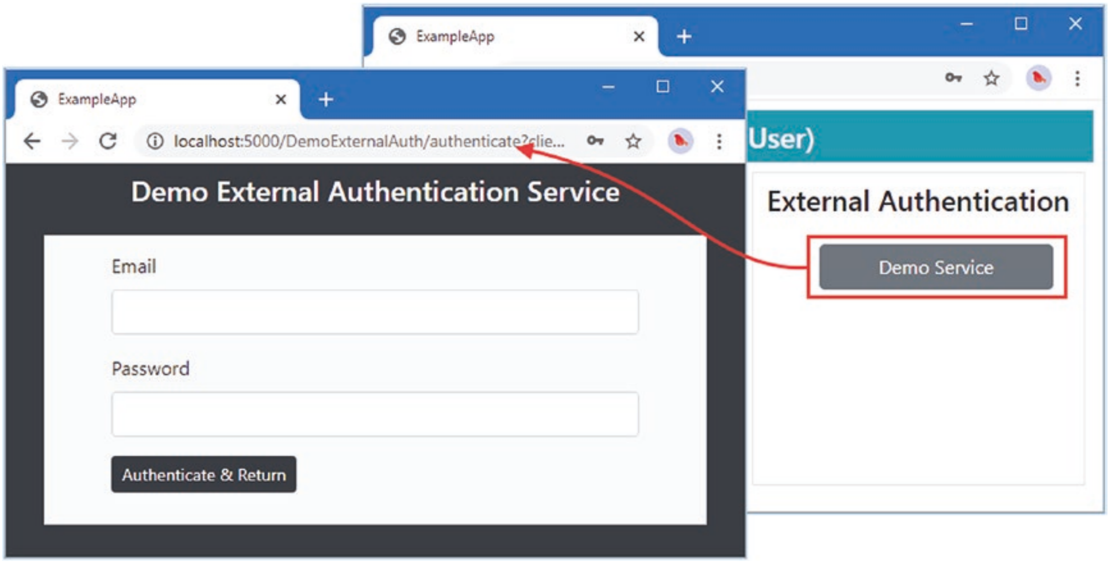

ExampleApp
localhost:5000/DemoExternalAuth/authenticate?clie...
Demo External Authentication Service
Email
Password
Authenticate & Return
User)
External Authentication
Demo Service

Figure 22-5. *Testing the redirection step*

As part of the authentication process, the user will be asked to approve the application's access to their data. This can be done as simply as including a description of the application when the user is prompted for their credentials or as a separate prompt that requires explicit confirmation.

Step 2: Authenticating the User

The ASP.NET Core Identity application doesn't participate in the external authentication process, which is conducted privately between the user and the authentication service, as shown in Figure 22-6.

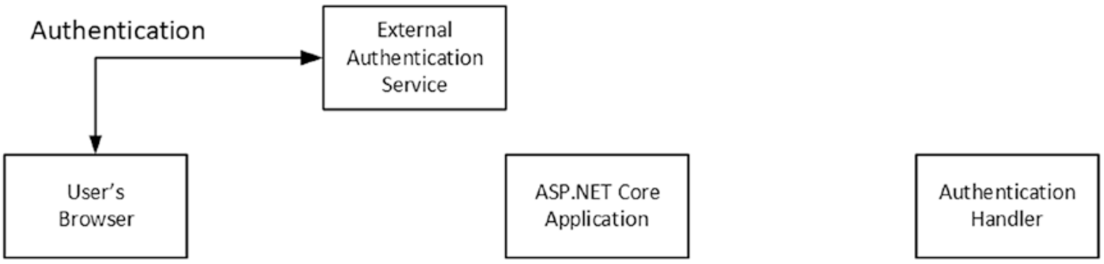


Figure 22-6. *Authenticating with the external service*

In Listing 22-19, I have added an action method to receive the credentials provided by the user and validate them.

Listing 22-19. Authenticating the User in the DemoExternalAuthController.cs File in the Controllers Folder

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

namespace ExampleApp.Controllers {

    class UserRecord {
        public string Id { get; set; }
        public string Name { get; set; }
        public string EmailAddress { get; set; }
        public string Password { get; set; }
        public string Code { get; set; }
        public string Token { get; set; }
    }

    public class ExternalAuthInfo {
        public string client_id { get; set; }
        public string client_secret { get; set; }
        public string redirect_uri { get; set; }
        public string scope { get; set; }
        public string state { get; set; }
        public string response_type { get; set; }
        public string grant_type { get; set; }
        public string code { get; set; }
    }

    public class DemoExternalAuthController : Controller {
        private static string expectedID = "MyClientID";
        private static string expectedSecret = "MyClientSecret";
        private static List<UserRecord> users = new List<UserRecord> {
            new UserRecord() {
                Id = "1", Name = "Alice", EmailAddress = "alice@example.com",
                Password = "myexternalpassword"
            },
             new UserRecord {
                Id = "2", Name = "Dora", EmailAddress = "dora@example.com",
                Password = "myexternalpassword"
             }
        };

        public IActionResult Authenticate([FromQuery] ExternalAuthInfo info)
         => expectedID == info.client_id ? View((info, string.Empty))
                 : View((info, "Unknown Client"));
        [HttpPost]
        public IActionResult Authenticate(ExternalAuthInfo info, string email,
                string password) {
            if (string.IsNullOrEmpty(email) || string.IsNullOrEmpty(password)) {
                ModelState.AddModelError("", "Email and password required");
            } else {
                UserRecord user = users.FirstOrDefault(u =>
                    u.EmailAddress.Equals(email) && u.Password.Equals(password));
                if (user != null) {
                    // user has been successfully authenticated
                } else {
                    ModelState.AddModelError("", "Email or password incorrect");
                }
            }
            return View((info, ""));
        }
    }
}
```

One benefit of using an external authentication service is that it allows applications to take advantage of security factors that are not directly supported by ASP.NET Core Identity. For example, Google and Facebook both offer external authentication services that support FIDO2, which relies on a hardware authenticator device that is accessed through a browser API (and which is described in detail at https://fidoalliance.org/fido2). Identity doesn't support FIDO2 directly but can benefit from its use by using Google or Facebook as an external authentication service.

For this example, the controller that simulates the external service has a static set of email addresses and passwords, which are used to check the credentials received by the action method. The new action method uses the ASP.NET Core model state feature to display an error if the credentials cannot be validated.

Step 3: Receiving the Authorization Code

When the user grants access to the data the application requires, the external authentication service replies with a redirection to the return URL from step 1, as shown in Figure 22-7.

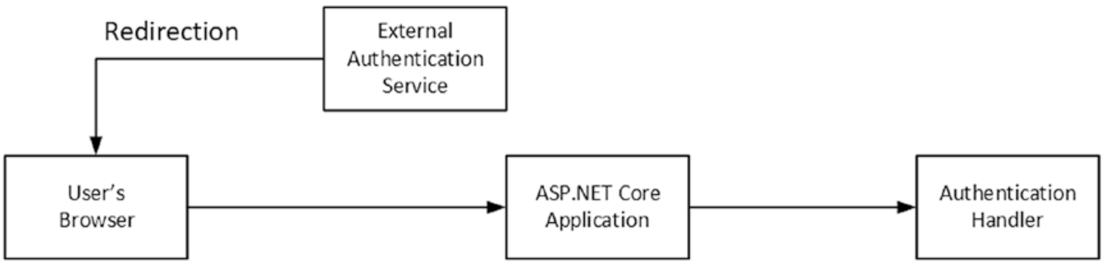


Figure 22-7. Receiving the authorization code

The authentication service uses the query string of the redirect URI to send data to the application via the user's browser: a scope, an authorization code, and a state data value if one was used in step 1. The authorization code tells the application that the user has been authenticated and has granted access to the data specified by the scope.

Note that the data itself is not included in the request—that happens in a later step. Table 22-12 summarizes the query string parameters the application includes in the redirection URL for quick reference.

Updating the External Authentication Controller

In Listing 22-20, I have added the redirection to the action method that authenticates the user. The query string for the redirection contains the data values described in Table 22-12.

Table 22-12. *The Query String Parameters Included in the Redirection URL*

Name	Description
`code`	This code is exchanged in the next step for an access token.
`state`	This is the same value used in step 1 and can be used by applications to correlate requests to determine which user the redirection relates to and to protect against cross-site request forgery attacks.

Listing 22-20. Redirecting in the DemoExternalAuthController.cs File in the Controllers Folder

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

namespace ExampleApp.Controllers {

    class UserRecord {
        public string Id { get; set; }
        public string Name { get; set; }
        public string EmailAddress { get; set; }
        public string Password { get; set; }
        public string Code { get; set; }
        public string Token { get; set; }
    }

    public class ExternalAuthInfo {
        public string client_id { get; set; }
        public string client_secret { get; set; }
        public string redirect_uri { get; set; }
        public string scope { get; set; }
        public string state { get; set; }
        public string response_type { get; set; }
        public string grant_type { get; set; }
        public string code { get; set; }
    }

    public class DemoExternalAuthController : Controller {
        private static string expectedID = "MyClientID";
        private static string expectedSecret = "MyClientSecret";
        private static List<UserRecord> users = new List<UserRecord> {
        new UserRecord() {
            Id = "1", Name = "Alice", EmailAddress = "alice@example.com",
            Password = "myexternalpassword", Code = "12345"
        },
        new UserRecord {
            Id = "2", Name = "Dora", EmailAddress = "dora@example.com",
            Password = "myexternalpassword", Code = "56789"
        }
    };

        public IActionResult Authenticate([FromQuery] ExternalAuthInfo info)
            => expectedID == info.client_id ? View((info, string.Empty))
                    : View((info, "Unknown Client"));

        [HttpPost]
        public IActionResult Authenticate(ExternalAuthInfo info, string email,
                string password) {
            if (string.IsNullOrEmpty(email) || string.IsNullOrEmpty(password)) {
                ModelState.AddModelError("", "Email and password required");
            } else {
                UserRecord user = users.FirstOrDefault(u =>
                    u.EmailAddress.Equals(email) && u.Password.Equals(password));
                if (user != null) {
                    return Redirect(info.redirect_uri
                        + $"?code={user.Code}&scope={info.scope}"
                        + $"&state={info.state}");
                } else {
                    ModelState.AddModelError("", "Email or password incorrect");
                }
            }
            return View((info, ""));
        }
    }

}
```

The URL to which the browser is redirected is determined using the `redirect_uri`, `scope`, and `state` values provided by the authentication handler in step 1. The query string also includes a code value, which I have defined statically for each user. In a real authentication service, the code values are generated dynamically.

Updating the Authentication Handler

The authentication handler needs to be able to receive the redirected request. ASP.NET Core defines the `IAuthenticationRequestHandler` interface, which is derived from the `IAuthenticationHandler` interface, and defines the additional method described in Table 22-13.

Table 22-13. *The IAuthenticationRequestHandler Method*

Name	Description
`HandleRequestAsync()`	This method is called for every request, allowing the authentication handler to intercept requests. The method returns a `bool` value, which indicates whether the processing of this request should stop. A result of `true` prevents the request from being passed along the request pipeline.

The `HandleRequestAsync` method is called automatically by the ASP.NET Core authentication middleware and allows authentication handlers to intercept requests without the need to create custom middleware classes or endpoints. In Listing 22-21, I have revised the example authentication handler to implement the `IAuthenticationRequestHandler` interface to receive the authorization code from the authentication service.

Listing 22-21. Receiving the Code in the ExternalAuthHandler.cs File in the Custom Folder

```
...
public class ExternalAuthHandler : IAuthenticationRequestHandler {

    public ExternalAuthHandler(IOptions<ExternalAuthOptions> options,
            IDataProtectionProvider dp) {
        Options = options.Value;
        DataProtectionProvider = dp;
    }

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

    public Task ForbidAsync(AuthenticationProperties properties) {
        return Task.CompletedTask;
    }

    public virtual async Task<bool> HandleRequestAsync() {
        if (Context.Request.Path.Equals(Options.RedirectPath)) {
            string authCode = await GetAuthenticationCode();
            return true;
        }
        return false;
    }

    protected virtual Task<string> GetAuthenticationCode() {
        return Task.FromResult(Context.Request.Query["code"].ToString());
    }
}
...
```

The implementation of the `HandleRequestAsync` method checks to see if the URL for the request matches the one specified by the `RedirectUri` configuration option. If it does match, then the authentication code is extracted from the request by the `GetAuthenticationCode` method. If the request doesn't match, it is passed along the pipeline as normal.

■ **Caution** Multiple instances of the authentication handler will be created and used to handle the different steps in the authentication sequence. Do not rely on storing data in instance variables or properties because those values will not be available to the object created to deal with the next step in the sequence.

Step 4: Exchanging the Authorization Code for an Access Token

So far, the application and the authentication service have communicated indirectly, passing data by redirecting the user's browser. In this step, the application contacts the authentication service directly to exchange the authorization code for an access token, which is used in the next step to get the user's data. Figure 22-8 shows the code-for-token exchange.

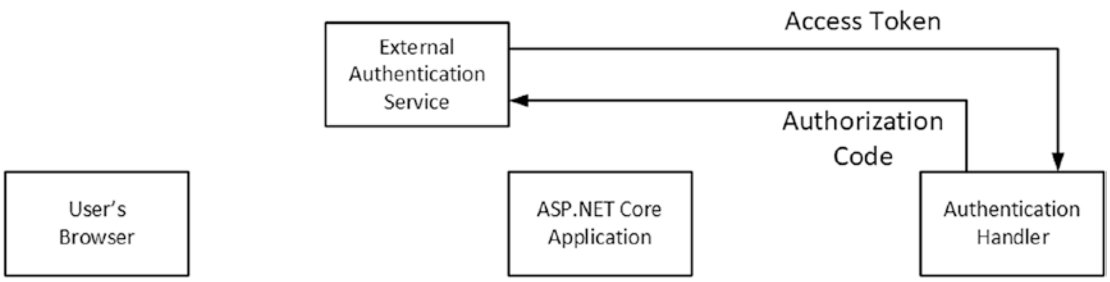


Figure 22-8. *Exchanging an authorization code for an access token*

In general, this step is performed with an HTTP POST request, where the authorization code is included in the request body. Table 22-14 describes the data values commonly included in the request.

Table 22-14. *The Data Values Sent in the Code Exchange Request*

Name	Description
`code`	This property is used to send the authorization code received in the previous step.
`redirect_uri`	This property is used to send the same URL that was used in step 1.
`client_id`	This property is used to send the client ID, created when the application was registered with the authentication service.
`client_secret`	This property is used to send the client secret, created when the application was registered with the authentication service. Some services require the application to authenticate itself differently.
`state`	This parameter sends a state data value, which is used by the application to correlate related requests and to protect against cross-site forgery attacks.
`grant_type`	This property must be set to `authorization_code`.

In the response to the HTTP request, the authentication sends a JSON document that contains an access token, along with additional data. Table 22-15 describes the JSON properties that will be sent.

***Table 22-15.** The JSON Document Properties Received in the Code Exchange*

Name	Description
`access_token`	This property provides the access token, which is used in the next step.
`expires_in`	This property specifies the lifespan of the token in seconds. The token must be used within this period.
`scope`	This property specifies the scope that the token provides access to. This may not be the same scope that was requested in step 1.
`token_type`	This property specifies the token type. The most common value is `Bearer`, which means that the token should be included as a header in requests for data.
`state`	This property contains the `state` value included in the request.

The authentication service can also send the application an error, indicating that something went wrong in the exchange process. Errors are described using a JSON document with the properties described in Table 22-16.

***Table 22-16.** The Error JSON Document Properties*

Name	Description
`error`	All error responses contain this property, which contains an error code describing the problem.
`state`	This property contains the `state` value included in the request if one was specified.
`error_description`	This property provides a human-readable description of the error. This property is optional.
`error_uri`	This property provides a URL for a human-readable web page that contains a description of the error. This property is optional.

■ **Note** There are other points in the process where the specification allows the authentication service to send an error to the application, but these are not always used consistently, and some are optional. The token exchange is the step where errors are most likely to be reported because it is the first time that the application and the authentication service communicate directly.

The important property is `error`, which identifies the error that has occurred. The OAuth specification defines a set of values for the `error` property, such as `invalid_request`, which is used when the request is missing required data, or `access_denied`, when the user doesn't grant the application access to their data.

I am not going to handle all of the errors that are described in the specification because, from the perspective of the ASP.NET Core application, I only care about the outcome of the process. It can be helpful to examine error types to identify the cause of persistent problems and configuration issues, but otherwise, I just need to display a "something went wrong" message when an error occurs.

Updating the External Authentication Controller

In Listing 22-22, I have added tokens to each user record and defined an action method that simulates the code-for-token exchange process.

Listing 22-22. Adding an Action in the DemoExternalAuthController.cs File in the Controllers Folder

```
...
public class DemoExternalAuthController : Controller {
    private static string expectedID = "MyClientID";
    private static string expectedSecret = "MyClientSecret";

    private static List<UserRecord> users  = new List<UserRecord> {
        new UserRecord() {
            Id = "1", Name = "Alice", EmailAddress = "alice@example.com",
            Password = "myexternalpassword", Code = "12345", Token = "token1"
        },
        new UserRecord {
            Id = "2", Name = "Dora", EmailAddress = "dora@example.com",
            Password = "myexternalpassword", Code = "56789", Token = "token2"
        }
    };

    public IActionResult Authenticate([FromQuery] ExternalAuthInfo info)
        => expectedID == info.client_id ? View((info, string.Empty))
                : View((info, "Unknown Client"));

    [HttpPost]
    public IActionResult Authenticate(ExternalAuthInfo info, string email,
            string password) {
        if (string.IsNullOrEmpty(email) || string.IsNullOrEmpty(password)) {
            ModelState.AddModelError("", "Email and password required");
        } else {
            UserRecord user = users.FirstOrDefault(u =>
                u.EmailAddress.Equals(email) && u.Password.Equals(password));
            if (user != null) {
                return Redirect(info.redirect_uri
                    + $"?code={user.Code}&scope={info.scope}"
                    + $"&state={info.state}");
            } else {
                ModelState.AddModelError("", "Email or password incorrect");
            }
        }
        return View((info, ""));
    }

    [HttpPost]
    public IActionResult Exchange([FromBody] ExternalAuthInfo info) {
        UserRecord user = users.FirstOrDefault(user => user.Code.Equals(info.code));
        if (user == null || info.client_id != expectedID
                || info.client_secret != expectedSecret) {
            return Json(new { error = "unauthorized_client" });
        } else {
            return Json(new {
                access_token = user.Token,
                expires_in = 3600,
                scope = "openid+email+profile",
                token_type = "Bearer",
                info.state
            });
        }
    }
}
...
```

The action method locates the user with the specified code and returns the user's token. In a real authentication service, the tokens are generated dynamically, which means you cannot rely on always receiving the same token for a given user.

Updating the Authentication Handler

In Listing 22-23, I have added support for obtaining a token to the authentication handler, targeting the action method defined in the previous section.

Listing 22-23. Obtaining a Token in the ExternalAuthHandler.cs File in the Custom Folder

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

namespace ExampleApp.Custom {

    public class ExternalAuthOptions {
        public string ClientId { get; set; } = "MyClientID";
        public string ClientSecret { get; set; } = "MyClientSecret";

        public virtual string RedirectRoot { get; set; } = "http://localhost:5000";
        public virtual string RedirectPath { get; set; } = "/signin-external";
        public virtual string Scope { get; set; } = "openid email profile";
        public virtual string StateHashSecret { get; set; } = "mysecret";

        public virtual string AuthenticationUrl { get; set; }
            = "http://localhost:5000/DemoExternalAuth/authenticate";
        public virtual string ExchangeUrl { get; set; }
```

```
            = "http://localhost:5000/DemoExternalAuth/exchange";
        public virtual string ErrorUrlTemplate { get; set; }
            = "/externalsignin?error={0}";
}

public class ExternalAuthHandler : IAuthenticationRequestHandler {

    public ExternalAuthHandler(IOptions<ExternalAuthOptions> options,
            IDataProtectionProvider dp, ILogger<ExternalAuthHandler> logger) {
        Options = options.Value;
        DataProtectionProvider = dp;
        Logger = logger;
    }

    public AuthenticationScheme Scheme { get; set; }
    public HttpContext Context { get; set; }
    public ExternalAuthOptions Options { get; set; }
    public IDataProtectionProvider DataProtectionProvider { get; set; }
    public PropertiesDataFormat PropertiesFormatter { get; set; }
    public ILogger<ExternalAuthHandler> Logger { get; set; }
    public string ErrorMessage { get; set; }

    public Task InitializeAsync(AuthenticationScheme scheme,
            HttpContext context) {
        Scheme = scheme;
        Context = context;
        PropertiesFormatter = new PropertiesDataFormat(DataProtectionProvider
            .CreateProtector(typeof(ExternalAuthOptions).FullName));
        return Task.CompletedTask;
    }

    public Task<AuthenticateResult> AuthenticateAsync() {
        return Task.FromResult(AuthenticateResult.NoResult());
    }

    public async Task ChallengeAsync(AuthenticationProperties properties) {
        Context.Response.Redirect(await GetAuthenticationUrl(properties));
    }

    protected virtual Task<string>
            GetAuthenticationUrl(AuthenticationProperties properties) {
        Dictionary<string, string> qs = new Dictionary<string, string>();
        qs.Add("client_id", Options.ClientId);
        qs.Add("redirect_uri", Options.RedirectRoot + Options.RedirectPath);
        qs.Add("scope", Options.Scope);
        qs.Add("response_type", "code");
        qs.Add("state", PropertiesFormatter.Protect(properties));
        return Task.FromResult(Options.AuthenticationUrl
            + QueryString.Create(qs));
    }
```

```
        public Task ForbidAsync(AuthenticationProperties properties) {
            return Task.CompletedTask;
        }

        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)) {
                    // todo - process token
                }
                Context.Response.Redirect(string.Format(Options.ErrorUrlTemplate,
                    ErrorMessage));
                return true;
            }
            return false;
        }

        protected virtual Task<string> GetAuthenticationCode() {
            return Task.FromResult(Context.Request.Query["code"].ToString());
        }

        protected virtual async Task<(string code, string state)>
                GetAccessToken(string code) {
            string state = Context.Request.Query["state"];
            HttpClient httpClient = new HttpClient();
            httpClient.DefaultRequestHeaders.Add("Accept", "application/json");
            HttpResponseMessage response = await httpClient
                .PostAsJsonAsync(Options.ExchangeUrl,
                    new {
                        code,
                        redirect_uri = Options.RedirectRoot + Options.RedirectPath,
                        client_id = Options.ClientId,
                        client_secret = Options.ClientSecret,
                        state,
                        grant_type = "authorization_code",
                    });
            string jsonData = await response.Content.ReadAsStringAsync();
            JsonDocument jsonDoc = JsonDocument.Parse(jsonData);
            string error = jsonDoc.RootElement.GetString("error");
            if (error != null) {
                ErrorMessage = "Access Token Error";
                Logger.LogError(ErrorMessage);
                Logger.LogError(jsonData);
            }
            string token = jsonDoc.RootElement.GetString("access_token");
            string jsonState = jsonDoc.RootElement.GetString("state") ?? state;
            return error == null ? (token, state) : (null, null);
        }

    }
}
```

The new method sends an HTTP POST request with the data values described in Table 22-16 and parses the JSON response using the built-in .NET Core JSON support. For simplicity, I have used a tuple as the result from the method, which allows me to return an error, the access code, and the state data returned by the authentication service. If the server has sent an error, the `HandleRequestAsync` method redirects the browser to the URL specified by the `ErrorUrl` configuration option. I'll create a Razor Page that displays an error message to the user later. I also log the error, including the JSON response. This is useful when adding support for new authentication services, which can be a trial-and-error process.

Step 5: Requesting User Data from the Authentication Service

The next step is to request the user's data from the authentication service. This is typically done as an HTTP GET request with the access token included as an `Authorization` header, as shown in Figure 22-9. No other information needs to be included in the request because the authentication service can use the tokens it issues to determine which user and application a token relates to.

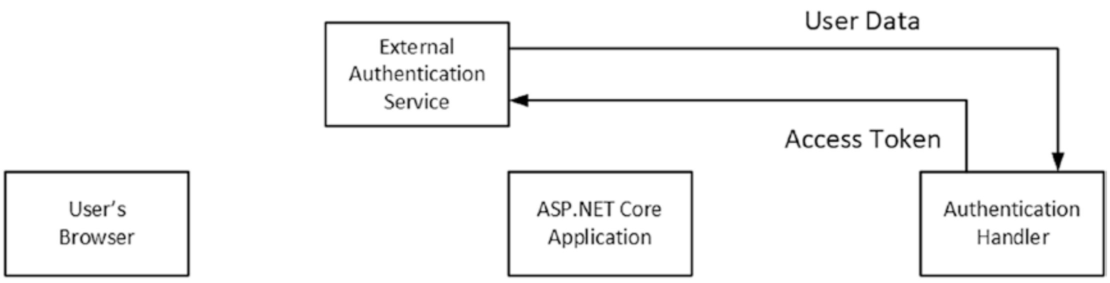


Figure 22-9. *Requesting user data*

The authentication service responds with JSON data that describes the user. As an example, here is a typical data response from the Google authentication service:

```
...
{
  "id": "102888805263382592",
  "email": "adam@adam-freeman.com",
  "verified_email": true,
  "name": "Adam Freeman",
  "given_name": "Adam",
  "family_name": "Freeman",
  "picture": "https://lh6.googleusercontent.com/s96-c/photo.jpg",
  "locale": "en"
}
...
```

The data that the application receives depends on the authentication service and the scope that has been requested. The data that the example controller produces is simpler but will be sufficient for this chapter.

Updating the External Authentication Controller

In Listing 22-24, I have added an action method named data that receives an access token and returns the data for a user.

Listing 22-24. Providing User Data in the DemoExternalAuthController.cs File in the Controllers Folder

```
...
public class DemoExternalAuthController : Controller {
    private static string expectedID = "MyClientID";
    private static string expectedSecret = "MyClientSecret";

    // ...statements and methods omitted for brevity...

    [HttpGet]
    public IActionResult Data([FromHeader] string authorization) {
        string token = authorization?[7..];
        UserRecord user = users.FirstOrDefault(user => user.Token.Equals(token));
        if (user != null) {
            return Json(new { user.Id, user.EmailAddress, user.Name });
        } else {
            return Json(new { error = "invalid_token" });
        }
    }
}
...
```

The access token will be obtained from the `Authorization` header, which is formatted so that it starts with `Bearer`, followed by a space, followed by the access token. The action method extracts the `token` from the header, locates the corresponding user data, and returns a JSON document that has `id`, `emailAddress` and `name` properties. (The case of the property names will be transformed automatically so that `EmailAddress` becomes `emailAddress` in the JSON document, for example.)

Updating the Authentication Handler

In the authentication handler, an HTTP GET request with an `Authorization` header is sent to the user, and a JSON document is returned. A set of `Claim` objects is created to represent the user data provided by the authentication service, and these objects are used to create a `ClaimsPrincipal` object that is signed into the application using the `HttpContext.SignInAsync` method, as shown in Listing 22-25.

Listing 22-25. Getting User Data in the ExternalAuthHandler.cs File in the Custom Folder

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;
using System.Security.Claims;
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.Options;
using System.Collections.Generic;
using Microsoft.AspNetCore.DataProtection;
using System.Net.Http;
using System.Net.Http.Json;
using System.Text.Json;
using Microsoft.Extensions.Logging;
using System.Net.Http.Headers;

namespace ExampleApp.Custom {

    public class ExternalAuthOptions {
        public string ClientId { get; set; } = "MyClientID";
        public string ClientSecret { get; set; } = "MyClientSecret";

        public virtual string RedirectRoot { get; set; } = "http://localhost:5000";
        public virtual string RedirectPath { get; set; } = "/signin-external";
        public virtual string Scope { get; set; } = "openid email profile";
        public virtual string StateHashSecret { get; set; } = "mysecret";

        public virtual string AuthenticationUrl { get; set; }
            = "http://localhost:5000/DemoExternalAuth/authenticate";
        public virtual string ExchangeUrl { get; set; }
            = "http://localhost:5000/DemoExternalAuth/exchange";
        public virtual string ErrorUrlTemplate { get; set; }
            = "/externalsignin?error={0}";
        public virtual string DataUrl { get; set; }
            = "http://localhost:5000/DemoExternalAuth/data";
    }

    public class ExternalAuthHandler : IAuthenticationRequestHandler {

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

        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);
                        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;
        }

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

        protected virtual async Task<IEnumerable<Claim>>
          GetUserData(string accessToken) {
            HttpRequestMessage msg = new HttpRequestMessage(HttpMethod.Get,
                Options.DataUrl);
            msg.Headers.Authorization = new AuthenticationHeaderValue("Bearer",
                 accessToken);
            HttpResponseMessage response = await new HttpClient().SendAsync(msg);
            string jsonData = await response.Content.ReadAsStringAsync();
            JsonDocument jsonDoc = JsonDocument.Parse(jsonData);

            var error = jsonDoc.RootElement.GetString("error");
            if (error != null) {
                ErrorMessage = "User Data Error";
                Logger.LogError(ErrorMessage);
                Logger.LogError(jsonData);
                return null;
            } else {
                return GetClaims(jsonDoc);
            }
        }

        protected virtual 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")));
            claims.Add(new Claim(ClaimTypes.Email,
                jsonDoc.RootElement.GetString("emailAddress")));
            return claims;
        }
    }
}
```

Throughout the authentication process, I have been using the `AuthenticationProperties` object received by the `ChallengeAsync` method as the state data in the redirections and requests to the authentication service. This has allowed me to preserve the data provided by the `SignInManager<T>` class at the start of the process so that I can use it when creating the `ClaimsPrincipal` object. First, I unprotected the serialized data, like this:

```
...
AuthenticationProperties props = PropertiesFormatter.Unprotect(state);
...
```

The Unprotect method re-creates the AuthenticationProperties object, which the authentication server returns without modification. I then use this object when signing in the ClaimsPrincipal object, like this:

```
...
await Context.SignInAsync(IdentityConstants.ExternalScheme, claimsPrincipal, props);
...
```

This is important because the AuthenticationProperties object contains data values that the SignInManager<T>.GetExternalLoginInfoAsync method looks for. The external sign-in won't be detected if you don't preserve and use this data when signing in.

Completing the External Authentication Process

The authentication handler is complete, and all that remains is to add the Razor Page that will create an Identity user when the correlation process fails. I prepared for this step in the "Understanding the Correlation Phase" section earlier in the chapter, but I couldn't easily add the feature because the authentication handler always authenticated requests with the same user ID. Now that I have built a simulated external authentication service, I can go back and finish up. Add a Razor Page named ExternalAccountConfirm.cshtml to the Pages folder with the contents shown in Listing 22-26. The name of this page was specified by the code in Listing 22-11.

Listing 22-26. The Contents of the ExternalAccountConfirm.cshtml File in the Pages Folder

```
@page
@model ExampleApp.Pages.ExternalAccountConfirmModel

<h4 class="bg-info text-white text-center p-2">Create Account</h4>

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

<form method="post" class="m-2">
    <input type="hidden" asp-for="@Model.ReturnUrl" />
    <div class="form-group">
        <label>Name</label>
        <input class="form-control" name="username"
               value="@Model.AppUser.UserName" />
    </div>
    <div class="form-group">
        <label>Email Address</label>
        <input readonly class="form-control"
               asp-for="@Model.AppUser.EmailAddress" />
    </div>
    <div class="form-group">
        <label>Authentication Scheme</label>
        <input readonly class="form-control"
               asp-for="@Model.ProviderDisplayName" />
    </div>
    <button class="btn btn-info" type="submit">
        Create Account
    </button>
    <a href="/signin" class="btn btn-secondary">Cancel</a>
</form>
```

In a real application, this page can be used to gather the data the application requires that is not provided by the authentication service, but, for this example, I present read-only fields for the user's email address and authentication scheme and allow only the username to be edited.

To implement the page model class, add the code shown in Listing 22-27 to the `ExternalAccountConfirm.cshtml.cs` file in the Pages folder. You will have to create this file if you are using Visual Studio Code.

Listing 22-27. The Contents of the ExternalAccountConfirm.cshtml.cs File in the Pages Folder

```
using ExampleApp.Identity;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using System.Security.Claims;
using System.Threading.Tasks;

namespace ExampleApp.Pages {

    public class ExternalAccountConfirmModel : PageModel {

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

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

        public AppUser AppUser { get; set; } = new AppUser();

        public string ProviderDisplayName { get; set; }

        [BindProperty(SupportsGet = true)]
        public string ReturnUrl { get; set; }

        public async Task<IActionResult> OnGetAsync() {
            ExternalLoginInfo info = await SignInManager.GetExternalLoginInfoAsync();
            if (info == null) {
                return Redirect(ReturnUrl);
            } else {
                ClaimsPrincipal external = info.Principal;
                AppUser.EmailAddress = external.FindFirstValue(ClaimTypes.Email);
                AppUser.UserName = external.FindFirstValue(ClaimTypes.Name);
                ProviderDisplayName = info.ProviderDisplayName;
                return Page();
            }
        }
        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);
                    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 GET handler method uses the `SignInManager<T>.GetExternalLoginInfoAsync` method to get the external sign-in created by the authentication handler. The claims created by the handler are used to set values for the `AppUser` properties, which are displayed to the user.

The POST handler method also creates an `AppUser` object from the external sign-in, with the addition of the value provided by the user for the `UserName` property. The `UserManager<T>` class is used to add the `AppUser` object to the store, add a claim for the `User` role, and store the external login details. The user is then signed into the application with the `SignInManager.ExternalLoginSignInAsync` method.

The validation performed by the `UserManager<T>` class is still applied when creating user objects based on external logins. For the example application, this means that the username must be unique, the email address must be in the `example.com` domain, and the email address must be confirmed before the user is signed in. I trust that the email address has been confirmed by the authentication service, so I set the `EmailAddressConfirmed` property to `true` before storing the `AppUser` object. If your application is unable to trust the authentication service's confirmation process, then you will need to extend the process of creating an account to send the user a confirmation code.

To test the login for an existing user, restart ASP.NET Core and request `http://localhost:5000/signout`. Check the Forget Me option and click the Sign Out button to sign out of the application. Next, request `http://localhost:5000/secret`, which will trigger the challenge response and offer the choice of a local or external login. Click Demo Service, and the browser will be redirected to the simulated authentication service. Enter alice@example.com into the Email field, enter myexternalpassword into the Password field, and click the Authenticate & Return button. You will be redirected to the application and prompted for an authenticator code because the user Alice is set up for two-factor authentication. Click the Sign In button, and you will see the external authentication summary. Click the Continue button, and you will be redirected to the `/secret` URL. Figure 22-10 shows the key parts of the sequence.

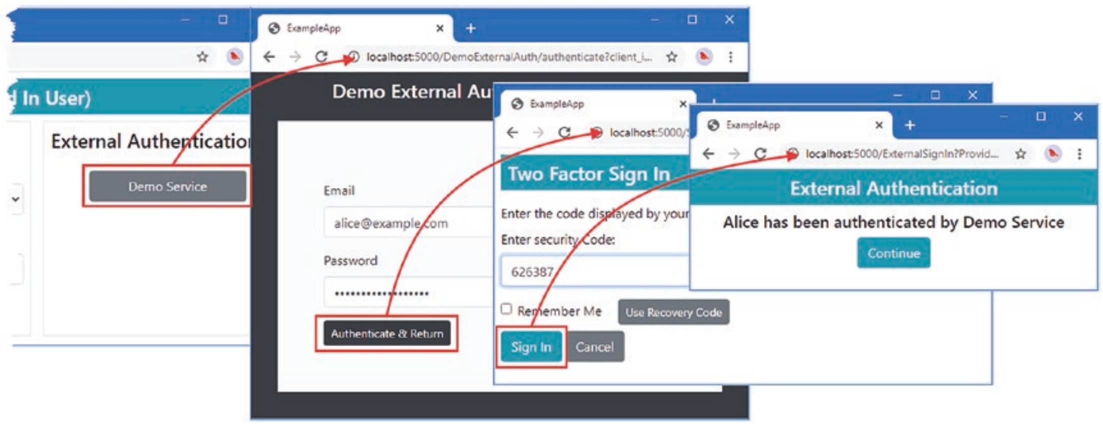


Figure 22-10. *External authentication*

Repeat the process and sign in as `dora@example.com` with the password `myexternalpassword` to see the process when there is no local account, as shown in Figure 22-11.

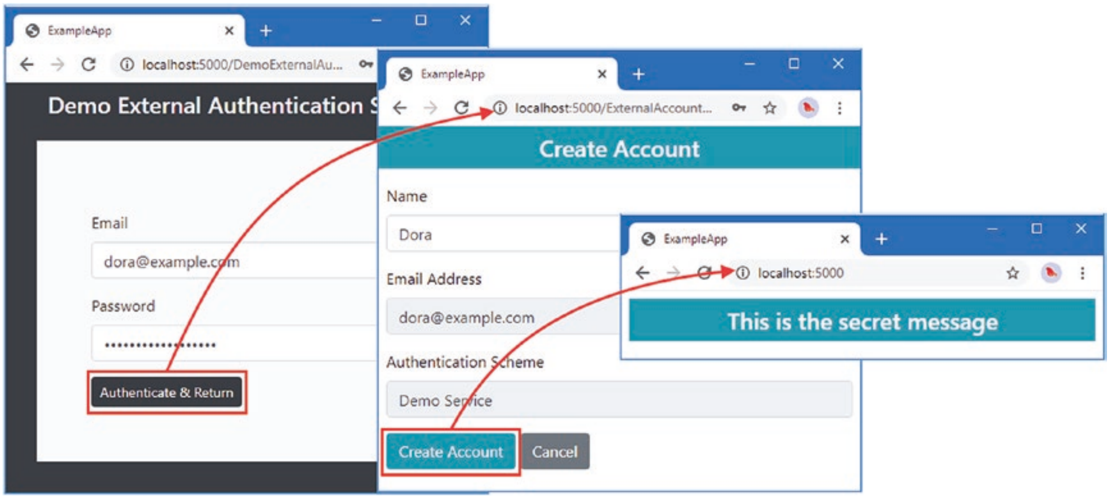


Figure 22-11. *External authentication without a local account*

Summary

In this chapter, I explained the process by which external services can be used to authenticate users on behalf of an ASP.NET Core application, using data stored by ASP.NET Core Identity. The process is complex, but the results can be worthwhile because it allows users to use their accounts on large-scale platforms, often taking advantage of security options that are not directly supported by Identity. In the next chapter, I replace the simulated external service with real ones.