CHAPTER 5

Configuring Identity

In this chapter, I explain how to configure Identity, including how to support third-party services from Google, Facebook, and Twitter. Some of these configuration options are part of the ASP.NET Core platform, but since they are so closely related to ASP.NET Core Identity, I have included them anyway. Table 5-1 puts the configuration options in context.

Table 5-1. *Putting Identity Configuration Options in Context*

Question	Answer
What are they?	The Identity configuration options are a set of properties whose values are used by the classes that implement the Identity API, which can be used directly or consumed through the Identity UI package.
Why are they useful?	These configuration options let you change the way that Identity behaves, which can make your application easier to use or allow you to meet the type of security standard that is commonly found in large corporations.
How are they used?	Identity is configured using the standard ASP.NET Core options pattern. The configuration for external authentication services is done using extension methods provided in the package that Microsoft provides for each provider.
Are there any pitfalls or limitations?	It is important to ensure that configuration changes do not cause problems for existing user accounts by enforcing a restriction that prevents the user from signing in.
Are there any alternatives?	The configuration options are used by the classes that provide the Identity API, which means the only way to avoid them is to create custom implementations, which I explain in Part 2.

A. Freeman, *Pro ASP.NET Core Identity*, https://doi.org/10.1007/978-1-4842-6858-2_5

Table 5-2 summarizes the chapter.

Table 5-2. *Chapter Summary*

Problem	Solution	Listing
Specify policies for usernames, email addresses, passwords, account confirmations, and lockouts	Set the properties defined by the `IdentityOptions` class.	1–4
Configure Facebook authentication	Install the package Microsoft provides for Facebook and use the `AddFacebook` method to configure the application ID and secret.	5–7
Configure Google authentication	Install the package Microsoft provides for Google and use the `AddGoogle` method to configure the application ID and secret.	8–10
Configure Twitter authentication	Install the package Microsoft provides for Twitter and use the `AddTwitter` method to configure the application ID and secret.	11–13

Preparing for This Chapter

This chapter uses the IdentityApp project from Chapter 4. No changes are required to prepare for this chapter. Open a PowerShell command prompt, navigate to the `IdentityApp` folder, and run the commands shown in Listing 5-1 to delete and then re-create the application and Identity databases.

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

Listing 5-1. Resetting the Databases

```
dotnet ef database drop --force --context ProductDbContext
dotnet ef database drop --force --context IdentityDbContext
dotnet ef database update --context ProductDbContext
dotnet ef database update --context IdentityDbContext
```

Use the PowerShell prompt to run the command shown in Listing 5-2 in the `IdentityApp` folder to start the application.

Listing 5-2. Running the Example Application

```
dotnet run
```

Open a web browser and request `https://localhost:44350`, which will show the output from the `Home` controller, and `https://localhost:44350/pages`, which will show the output from the `Landing` Razor Page, as shown in Figure 5-1.

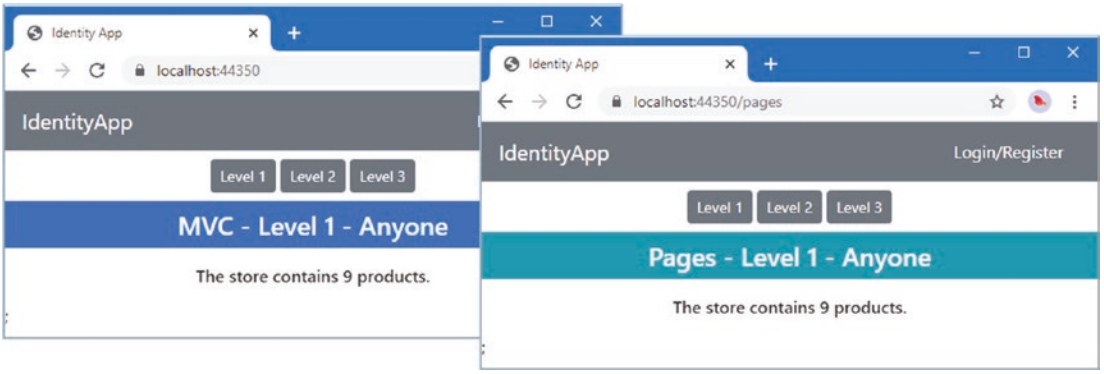

Figure 5-1. *Running the example application*

Click the Login/Register link, click Register as a New User, and create a new account using the values shown in Table 5-3.

Table 5-3. *Values for Creating a New Account*

Field	Value
Email	`alice@example.com`
Password	`MySecret1$`

Click the Register button; the new account will be created, and you will be signed into the application, as shown in Figure 5-2.

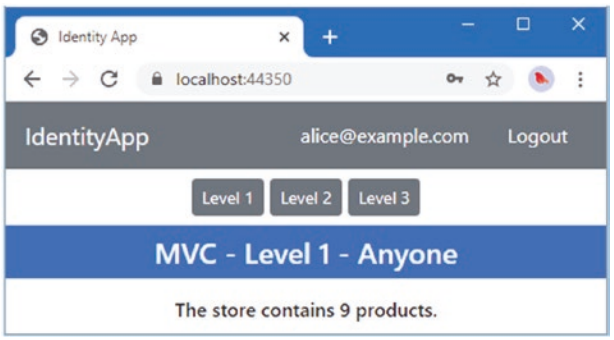

Figure 5-2. *Creating a new account*

Configuring Identity

Identity is configured using the standard ASP.NET Core options pattern, using the settings defined by the `IdentityOptions` class defined in the `Microsoft.AspNetCore.Identity` namespace. Table 5-4 describes the most useful properties defined by the `IdentityOptions` class, each of which leads to its own set of options, described in the sections that follow.

Table 5-4. *Useful IdentityOptions Properties*

Name	Description
`User`	This property is used to configure the username and email options for user accounts using the `UserOptions` class, as described in the "Configuring User Options" section.
`Password`	This property is used to define the password policy using the `PasswordOptions` class, as described in the "Configuring Password Options" section.
`SignIn`	This property is used to specify the confirmation requirements for accounts using the `SignInOptions` class, as described in the "Configuring Sign IN Confirmation Requirements" section.
`Lockout`	This property uses the `LockoutOptions` class to define the policy for locking out accounts after a number of failed attempts to sign in, as described in the "Configuring Lockout Options" section.

Configuring User Options

The `IdentityOptions.User` property is assigned a `UserOptions` object, which is used to configure the properties described in Table 5-5.

Table 5-5. *The UserOptions Properties*

Name	Description
`AllowedUserNameCharacters`	This property specifies the characters allowed in usernames. The default value is the set of upper and lowercase A–Z characters, the digits 0–9, and the symbols `-._@+` (hyphen, period, underscore, at character, and plus symbol).
`RequireUniqueEmail`	This property determines whether email addresses must be unique. The default value is `false`.

The Identity UI package isn't affected by either property because it uses email addresses as usernames. One consequence of this decision is that email addresses are effectively unique because Identity requires usernames to be unique.

Request `https://localhost:44350/Identity/Account/Register` and try to create an account using the email address `alice@example.com` with the password `MySecret1$`. Even though the default value of the `UserOptions.RequireUniqueEmail` property is `false`, you will receive an error message, as shown in Figure 5-3, because the Identity UI package uses the email address as the username when creating an account.

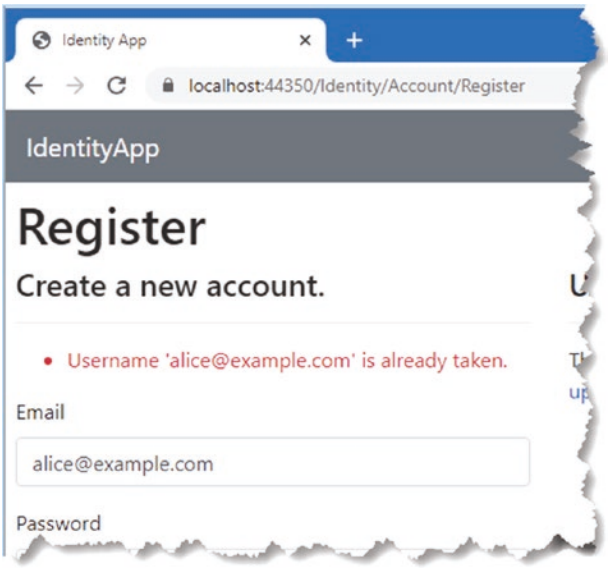

Figure 5-3. *Creating an account with an existing email address*

Configuring Password Options

The `IdentityOptions.Password` property is assigned a `PasswordOptions` object, which is used to configure the properties described in Table 5-6.

Table 5-6. *The PasswordOptions Properties*

Name	Description
`RequiredLength`	This property specifies a minimum number of characters for passwords. The default value is 6.
`RequiredUniqueChars`	This property specifies the minimum number of unique characters that a password must contain. The default value is 1.
`RequireNonAlphanumeric`	This property specifies whether passwords must contain nonalphanumeric characters, such as punctuation characters. The default value is `true`.
`RequireLowercase`	This property specifies whether passwords must contain lowercase characters. The default value is `true`.
`RequireUppercase`	This property specifies whether passwords must contain uppercase characters. The default value is `true`.
`RequireDigit`	This property specifies whether passwords must contain number characters. The default value is `true`.

The `IdentityUI` package only uses email addresses to identify users, to which the `UserOptions.AllowedUserNameCharacters` does not apply. Listing 5-3 uses the other user property and changes the password properties to change the Identity configuration.

Listing 5-3. Configuring Password Settings in the Startup.cs File in the IdentityApp Folder

```
...
services.AddDefaultIdentity<IdentityUser>(opts => {
    opts.Password.RequiredLength = 8;
    opts.Password.RequireDigit = false;
    opts.Password.RequireLowercase = false;
    opts.Password.RequireUppercase = false;
    opts.Password.RequireNonAlphanumeric = false;
}).AddEntityFrameworkStores<IdentityDbContext>();
...
```

Restrictive password policies are falling out of use, with simple minimum length requirements combined with the use of two-factor authentication. The settings in Listing 5-3 increase the length requirement to eight characters and disable the other restrictions.

Restart ASP.NET Core, make sure you are signed in as `alice@example.com` with password `MySecret1$`, click the email address in the header, and click the Password link. Use `MySecret1$` as the current password, which complies with the default password policy, and use `secret` as the new password, which does not meet the length requirement of the new policy. The new password will be rejected, as shown in Figure 5-4. Change the password again using `mysecret` as the new password, which does meet the length requirement specified in Listing 5-3. As shown in Figure 5-4, the longer password is accepted.

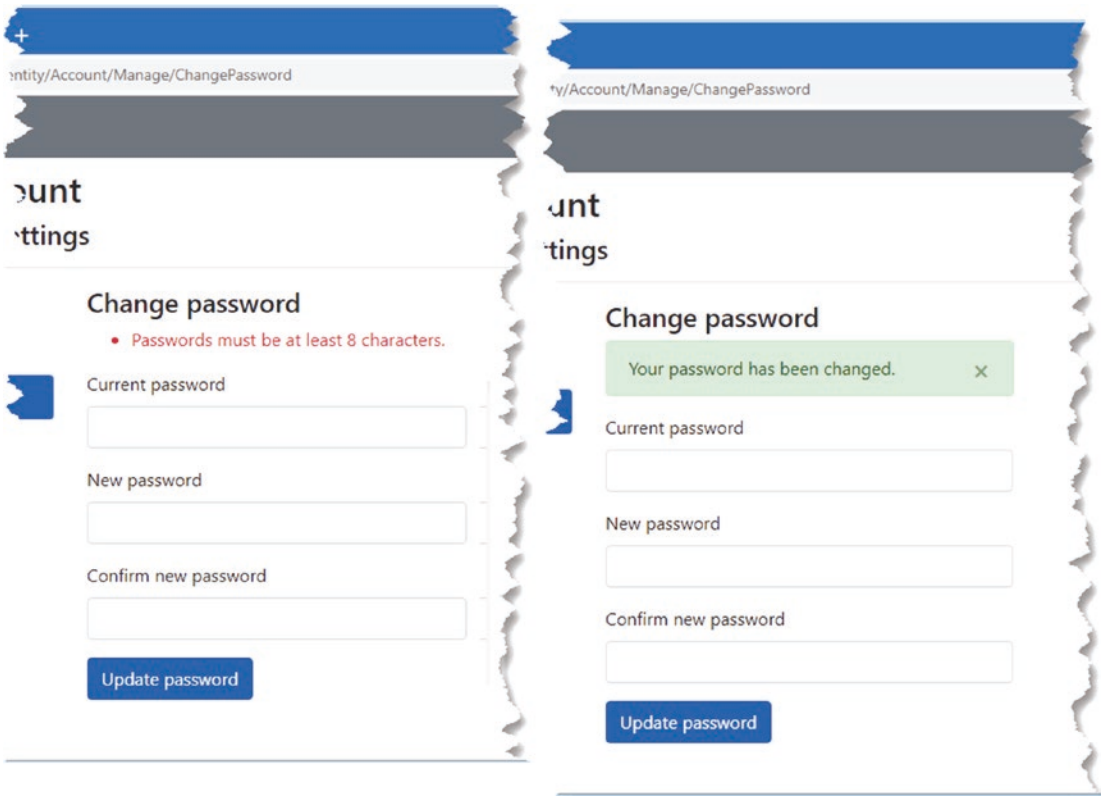

Figure 5-4. *Specifying password options*

Configuring Sign-in Confirmation Requirements

The `IdentityOptions.SignIn` property is assigned a `SignInOptions` object, which is used to configure the confirmation requirements for accounts using the properties described in Table 5-7.

Table 5-7. *The SignInOptions Properties*

Name	Description
`RequireConfirmedEmail`	When this property is set to `true`, only accounts with confirmed email addresses can sign in. The default value is `false`.
`RequireConfirmedPhoneNumber`	When this property is set to true, only accounts with confirmed phone numbers can sign in. The default value is false.
`RequireConfirmedAccount`	When set to `true`, only accounts that pass verification by the `IUserConfirmation<T>` interface can sign in. I describe this interface in detail in Chapter 9, and the default implementation checks that the email address has been confirmed. This default value for this property is `false`.

The Identity UI package doesn't support phone number confirmations, so the `RequireConfirmedPhoneNumber` property must not be set to `true` because it will lock all users out of the application.

It is a good idea to set the `RequireConfirmedAccount` property to `true`, as shown in Listing 5-4, if the application uses email for tasks such as password recovery.

Listing 5-4. Requiring Email Confirmations in the Startup.cs File in the IdentityApp Folder

```
...
services.AddDefaultIdentity<IdentityUser>(opts => {
    opts.Password.RequiredLength = 8;
    opts.Password.RequireDigit = false;
    opts.Password.RequireLowercase = false;
    opts.Password.RequireUppercase = false;
    opts.Password.RequireNonAlphanumeric = false;
    opts.SignIn.RequireConfirmedAccount = true;

}).AddEntityFrameworkStores<IdentityDbContext>();
...
```

The Identity UI features that send emails to users, such as password recovery, will silently fail if the user hasn't confirmed their email address. Enabling the `RequireConfirmedAccount` setting displays the `Account/RegisterConfirmation` page at the end of the sign-in process, which instructs the user to check for the confirmation email. To see this behavior, restart ASP.NET Core and navigate to `https://localhost:44350/Identity/Account/Register`. Create a new account using the values shown in Table 5-8.

■ **Tip** I show you how to provide the user with more helpful feedback by customizing the Identity UI package in Chapter 6.

Table 5-8. *Values for Creating a New Account*

Field	Value
Email	`bob@example.com`
Password	`mysecret`

Click the Register button, and you will be presented with the confirmation page, as shown in Figure 5-5.

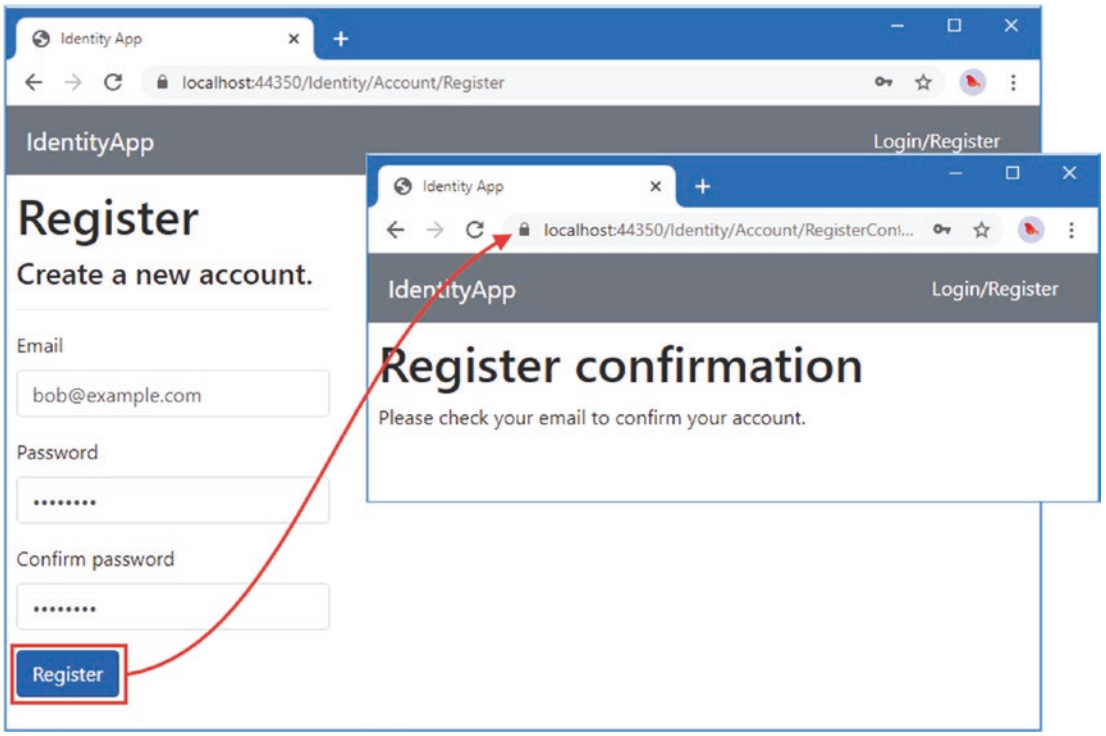

Figure 5-5. *The Identity UI confirmation behavior*

If you attempt to sign in using the new account without using the confirmation link, then you will be presented with a generic Invalid Login Attempt error.

Click the Resend Email Confirmation link displayed by the `Login` page to generate a new confirmation email, and you will see the confirmation link displayed in the console output from ASP.NET Core. Copy the URL into a browser, and you will be able to sign into the application, as shown in Figure 5-6.

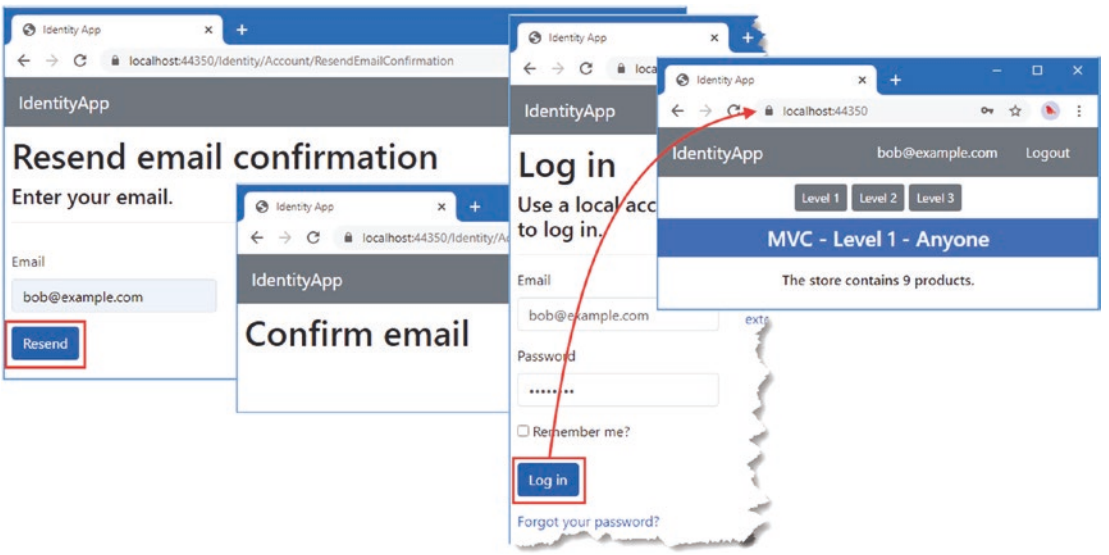

Figure 5-6. *Confirming an email address and signing into the application*

Configuring Lockout Options

The `IdentityOptions.Lockout` property is assigned a `LockoutOptions` object, which is used to configure lockouts that prevent sign-ins, even if the correct password is used, after a number of failed attempts. Table 5-9 describes the properties defined by the `LockoutOptions` class. The Identity UI package signs users into the application in such a way that the lockout feature is not triggered. I demonstrate how to change this behavior in Chapter 9.

Table 5-9. *The LockoutOptions Properties*

Name	Description
`MaxFailedAccessAttempts`	This property specifies the number of failed attempts allowed before an account is locked out. The default value is 5.
`DefaultLockoutTimeSpan`	This property specifies the duration for lockouts. The default value is 5 minutes.
`AllowedForNewUsers`	This property determines whether the lockout feature is enabled for new accounts. The default value is `true`.

Configuring External Authentication

External authentication delegate the process of authenticating users to a third-party service. In a corporate environment, the third-party service can be a company-wide user directory, which allows multiple applications to authenticate users from the same set of accounts.

For most Internet-facing applications, the external services are provided by the big technology/social media companies, such as Google, Facebook, and Twitter, taking advantage of the huge userbases these companies enjoy and the wider range of two-factor authentication schemes these services support and that are not available in Identity.

External authentication generally uses the OAuth protocol, which I describe in detail in Chapter 22. A registration process is required for each external service, during which the application is described and the level of access to user data, known as the *scope*, is declared.

During registration, you will usually have to specify a *redirection URL*. During the authentication process, the external service will send the user's browser an HTTP redirection to this URL, which triggers a request to ASP.NET Core, providing the application with data required to complete the sign-in. During development, this URL will be to `localhost`, such as `https://localhost:44350/signin-google`, for example, so that you can easily test external authentication on your development machine.

■ **Note** When you are ready to deploy your application, you will need to update your application's registration with each external service to use a publicly accessible URL that contains a hostname that appears in the DNS.

The registration process produces two data items: the client ID and the client secret. The client ID identifies the application to the external authentication service and can be shared publicly. The client secret is secret, as the name suggests, and should be protected.

GETTING HELP WITH EXTERNAL AUTHENTICATION

The setup procedures I describe in the following sections are correct at the time of writing but may change by the time you read this chapter. Microsoft publishes instructions for the most popular external authentication services, which you can find at `https://docs.microsoft.com/en-us/aspnet/core/security/authentication/social`. Each authentication service also provides documentation, for which there are links in the sections that follow.

Please do not email me to ask for help setting up external authentication. I try to help readers with most problems, but figuring out external authentication issues would require signing into a reader's Google/Facebook/Twitter accounts, which is something that I will not do.

Configuring Facebook Authentication

To register the application with Facebook, go to `https://developers.facebook.com/apps` and sign in with your Facebook account. Click the Create App button, select Build Connected Experiences from the list, and click the Continue button. Enter `IdentityApp` into the App Display Name field and click the Create App button. This sequence is shown in Figure 5-7.

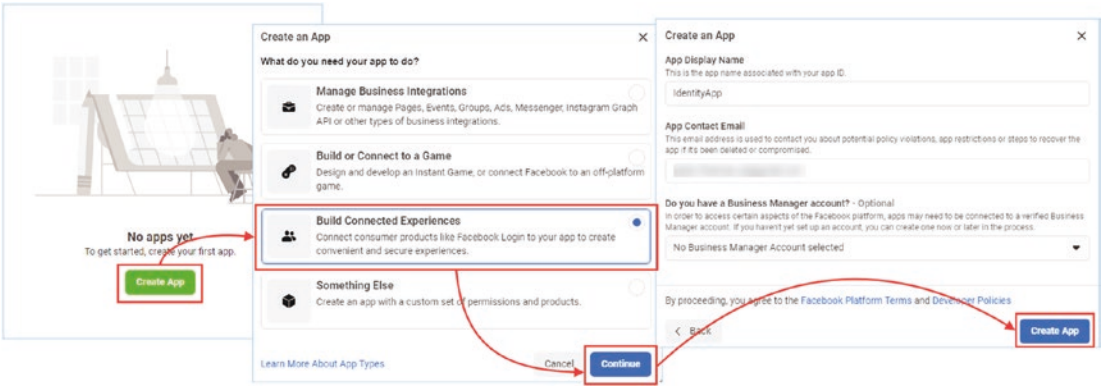

Figure 5-7. *Creating a new application*

Once you have created a Facebook application, you will be returned to the developer dashboard and presented with a list of optional products to use. Locate Facebook Login and click the Setup button. You will see a set of quick-start options, but these can be ignored because the important configuration options are shown under the Facebook Login ➤ Settings section that appears on the left side of the dashboard display, as shown in Figure 5-8.

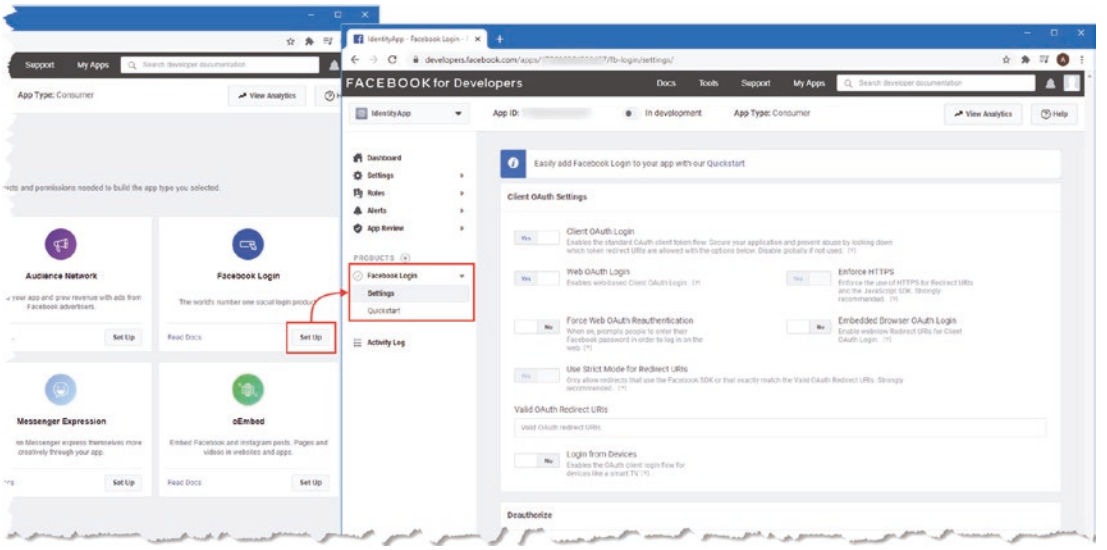

Figure 5-8. *The Facebook Login settings*

You don't need to specify a redirection URL because Facebook allows redirection to `localhost` URLs during development. When you are ready to deploy the application, you will need to return to this page and finalize your configuration, including providing the public-facing redirection URL. Details of the configuration options are included in the Facebook Login documentation, which can be found at `https://developers.facebook.com/docs/facebook-login`.

Navigate to the Basic section in the Settings area to get the App ID and App Secret, as shown in Figure 5-9, which are the terms that Facebook uses for the client ID and secret. (The App Secret is hidden until you click the Show button.) Make a note of these values, which will be required to configure the application in the next section.

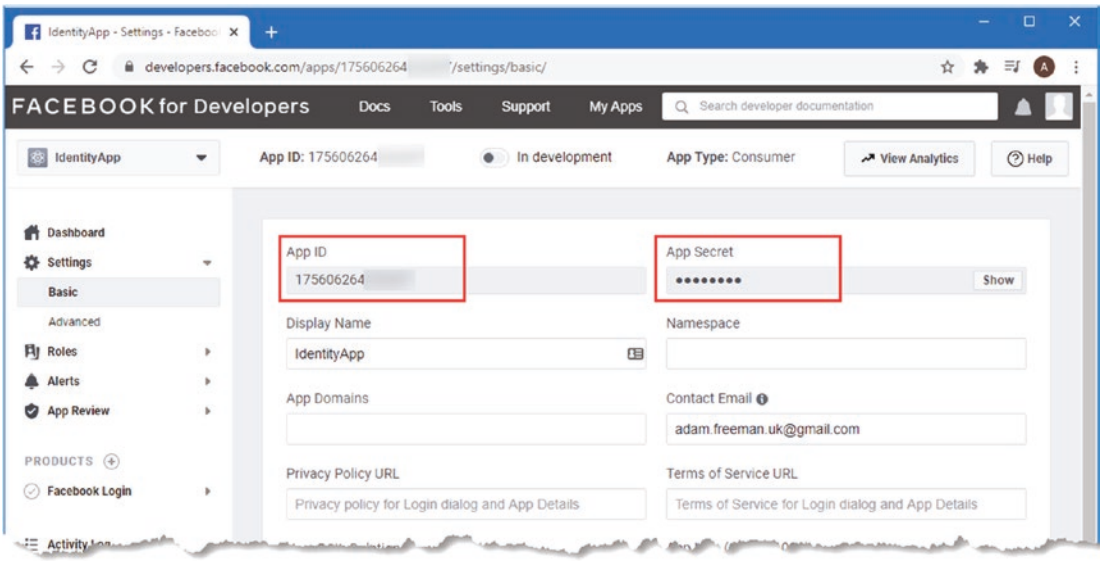

Figure 5-9. *The application credentials for external authentication*

Configuring ASP.NET Core for Facebook Authentication

Run the commands shown in Listing 5-5 to store the App ID and secret using the .NET secrets feature, which ensures that these values won't be included when the source code is committed into a repository.

Listing 5-5. Storing the Facebook ID and Secret

```
dotnet user-secrets init
dotnet user-secrets set "Facebook:AppId" "<app-id>"

dotnet user-secrets set "Facebook:AppSecret" "<app-secret>"
```

The first command initializes the secret store. The other commands store the data values so they can be accessed in the application using the ASP.NET Core configuration system. Next, run the command shown in Listing 5-6 in the `IdentityApp` folder to add the package containing Facebook authentication to the project.

Listing 5-6. Adding the Facebook Package

```
dotnet add package Microsoft.AspNetCore.Authentication.Facebook --version 5.0.0
```

To configure the application, add the statements shown in Listing 5-7 to the Startup class.

Listing 5-7. Configuring Facebook Authentication in the Startup.cs File in the IdentityApp Folder

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddControllersWithViews();
    services.AddRazorPages();
    services.AddDbContext<ProductDbContext>(opts => {
        opts.UseSqlServer(
            Configuration["ConnectionStrings:AppDataConnection"]);
    });

    services.AddHttpsRedirection(opts => {
        opts.HttpsPort = 44350;
    });

    services.AddDbContext<IdentityDbContext>(opts => {
        opts.UseSqlServer(
            Configuration["ConnectionStrings:IdentityConnection"],
            opts => opts.MigrationsAssembly("IdentityApp")
        );
    });

    services.AddScoped<IEmailSender, ConsoleEmailSender>();

    services.AddDefaultIdentity<IdentityUser>(opts => {
        opts.Password.RequiredLength = 8;
        opts.Password.RequireDigit = false;
        opts.Password.RequireLowercase = false;
        opts.Password.RequireUppercase = false;
        opts.Password.RequireNonAlphanumeric = false;

        opts.SignIn.RequireConfirmedEmail = true;
    })
    .AddEntityFrameworkStores<IdentityDbContext>();

    services.AddAuthentication()
        .AddFacebook(opts => {
            opts.AppId = Configuration["Facebook:AppId"];
            opts.AppSecret = Configuration["Facebook:AppSecret"];
        });
}
...
```

The AddAuthentication method sets up the ASP.NET Core authentication features, which I describe in detail in Part 2. This method is called automatically by the AddDefaultIdentity method, which is why it has not been needed until now. The AddFacebook method sets up the Facebook authentication support provided by Microsoft, which is configured using the options pattern with the FacebookOptions class. Table 5-10 describes the most important configuration properties.

Table 5-10. *Selected FacebookOptions Properties*

Name	Description
`AppId`	This property is used to configure the App ID, which is the term Facebook uses for the client ID. In the example, I read the value from the secret created in Listing 5-5.
`AppSecret`	This property is used to configure the App Secret, which is the term Facebook uses for the client secret. In the example, I read the value from the secret created in Listing 5-5.
`Fields`	This property specifies the data values that are requested from Facebook during authentication. The default values are `name`, `email`, `first_name`, and `last_name`. See `https://developers.facebook.com/docs/graph-api/reference/user` for a full list of fields, but bear in mind that some fields require applications to go through an additional validation process.

Restart ASP.NET Core and make sure you are signed out of the application by clicking the Logout link in the header and then again in the page that is displayed. Next, click the Level 2 button to trigger a challenge response, and you will see that the Identity UI package has automatically detected the Facebook configuration. Click the Facebook button, and you will be redirected to the Facebook authentication service. Once authenticated, you will be asked to approve the example application's accessing your data and then redirected to the application to complete the registration process, as shown in Figure 5-10.

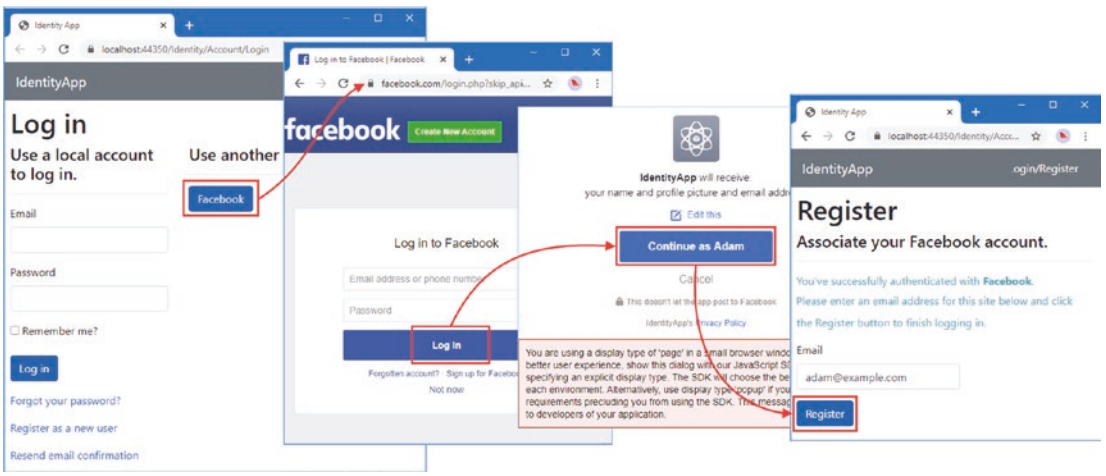

Figure 5-10. *Signing in with Facebook*

The default external authentication registration workflow provided by the Identity UI package allows the user to create an account with an email address that is different from the one associated with the Facebook account. This means that a confirmation message has to be sent to ensure the user has control of the specified email address. If you examine the ASP.NET Core console output, you will see a message similar to this one (I have shortened the confirmation code for brevity):

```
---New Email----
To: adam@example.com
Subject: Confirm your email
Please confirm your account by <a href='https://localhost:44350/Identity/Account/
ConfirmEmail?userId=8917d84d'>
```

```
   clicking here
</a>.
-------
```

As a consequence, the user must click the link in the email before they can sign in again. I show you how to customize the Identity UI package to change this behavior in Chapter 6.

Configuring Google Authentication

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

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

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

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

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

Click Save and Continue to return to the OAuth consent screen and then click Back to Dashboard. Figure 5-11 shows the sequence for configuring the consent screen.

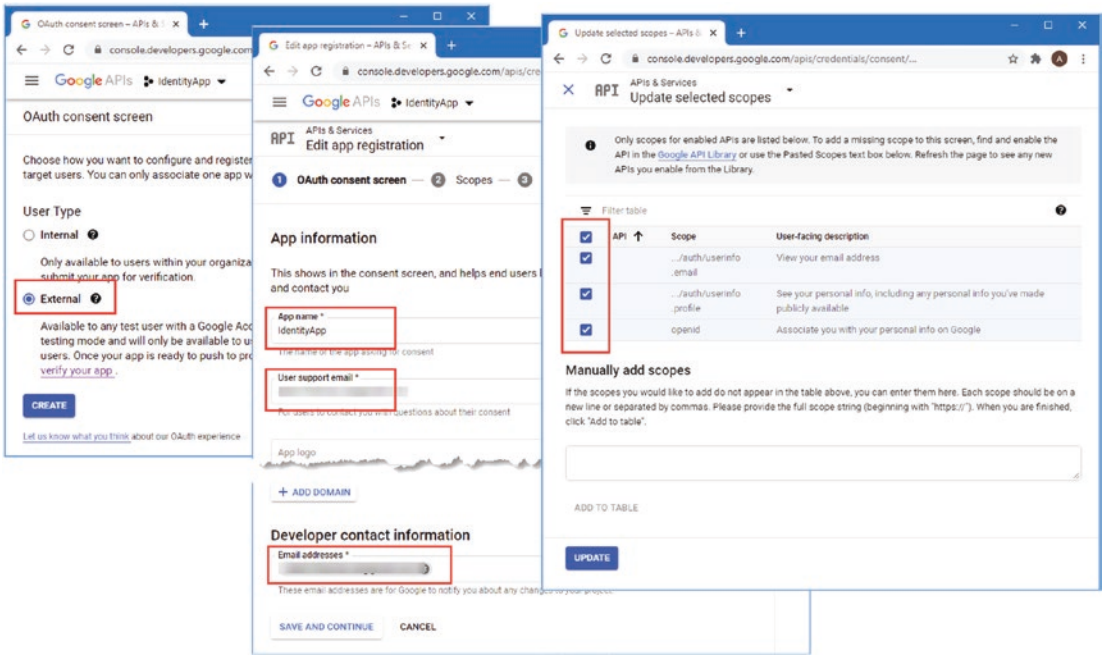

Figure 5-11. *Configuring the Google OAuth consent screen*

Click the Publish App button and click Confirm, as shown in Figure 5-12, which will allow any Google account to be authenticated.

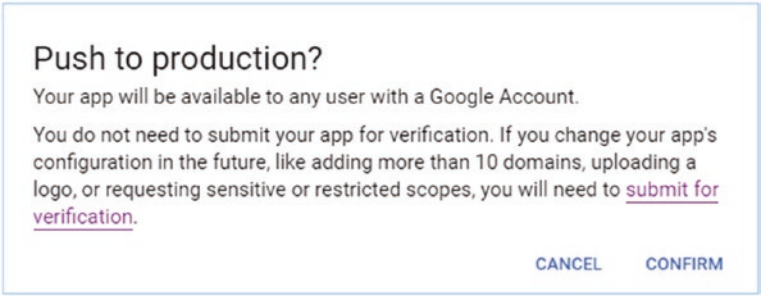

Figure 5-12. *Publishing the application*

Click the Credentials link, click the Create Credentials button at the top of the page, and select OAuth Client ID from the list of options.

Select Web Application from the Application Type list and enter IdentityApp in the Name field. Click Add URI in the Authorized Redirect URIs section and enter `https://localhost:44350/signin-google` into the text field. Click the Create button, and you will be presented with the client ID and client secret for your application, as shown in Figure 5-13 (although I have blurred the details, since these are for my account). Make a note of the ID and secret, which will be used to configure the application.

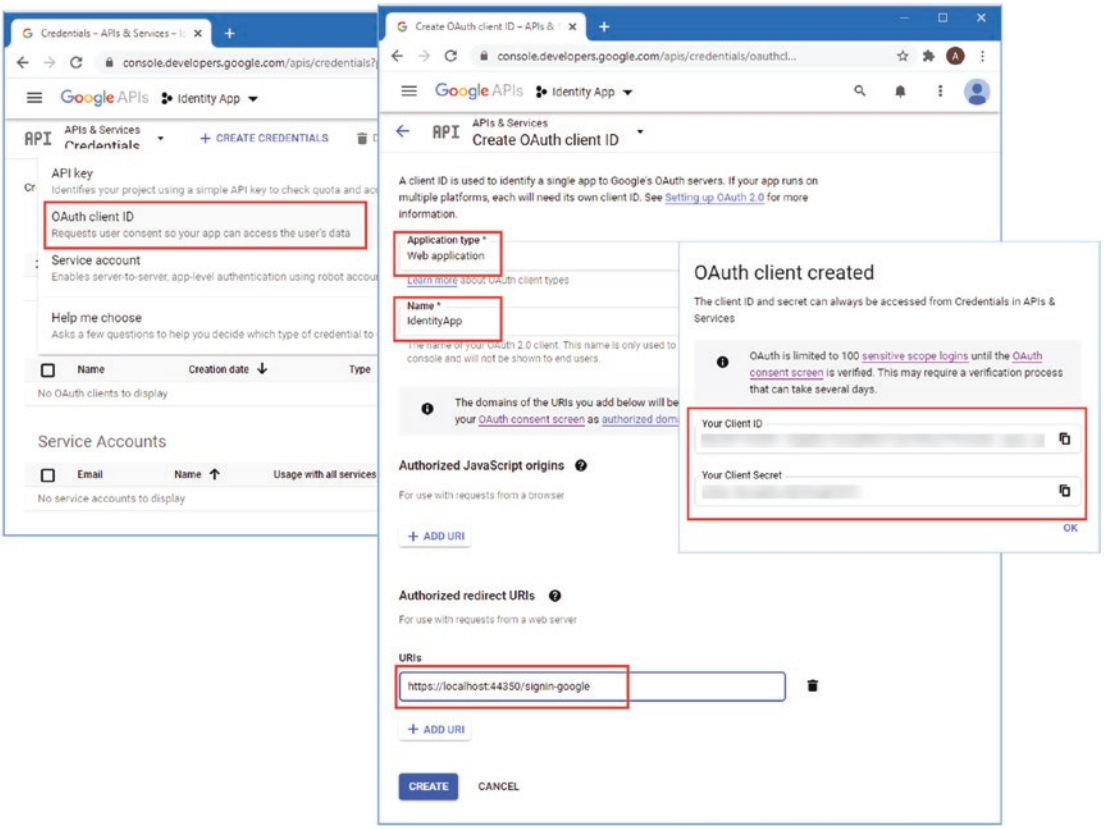

Figure 5-13. *Configuring application credentials*

Configuring ASP.NET Core for Google Authentication

Run the commands shown in Listing 5-8 to store the client ID and secret using the .NET secrets feature, which ensures that these values won't be included when the source code is committed into a repository.

Listing 5-8. Storing the Google Client ID and Secret

```
dotnet user-secrets init
dotnet user-secrets set "Google:ClientId" "<client-id>"

dotnet user-secrets set "Google:ClientSecret" "<client-secret>"
```

The first command initializes the secret store. The other commands store the data values so they can be accessed in the application using the ASP.NET Core configuration system. Next, run the command shown in Listing 5-9 in the `IdentityApp` folder to add the package containing Google authentication to the project.

Listing 5-9. Adding the Google Package

```
dotnet add package Microsoft.AspNetCore.Authentication.Google --version 5.0.0
```

To configure the application, add the statements shown in Listing 5-10 to the `Startup` class.

Listing 5-10. Configuring Google Authentication in the Startup.cs File in the IdentityApp Folder

```
...
services.AddAuthentication()
    .AddFacebook(opts => {
        opts.AppId = Configuration["Facebook:AppId"];
        opts.AppSecret = Configuration["Facebook:AppSecret"];
    })
    .AddGoogle(opts => {
        opts.ClientId = Configuration["Google:ClientId"];
        opts.ClientSecret = Configuration["Google:ClientSecret"];
    });
...
```

The `AddGoogle` method sets up the Google authentication handler and is configured using the options pattern with the `GoogleOptions` class. Table 5-11 describes the most important `GoogleOptions` properties.

Table 5-11. *Selected GoogleOptions Properties*

Name	Description
`ClientId`	This property is used to specify the client ID for the application. In the example, I use the value stored in Listing 5-8.
`ClientSecret`	This property is used to specify the application's client secret. In the listing, I use the value stored in Listing 5-8.
`Scope`	This property is used to set the scopes that are requested from the authentication service. The default value requests the scopes specified during the setup process, but additional scopes are available. See `https://developers.google.com/identity/protocols/oauth2/web-server`.

Restart ASP.NET Core and make sure you are signed out of the application by clicking the Logout link in the header and again in the page that is displayed. Next, click the Level 2 button to trigger a challenge response, and you will see that the Identity UI package has automatically detected the Google configuration. Click the Google button, and you will be redirected to the Google authentication service. Once authenticated, you will be redirected to the application to complete the registration process, as shown in Figure 5-14.

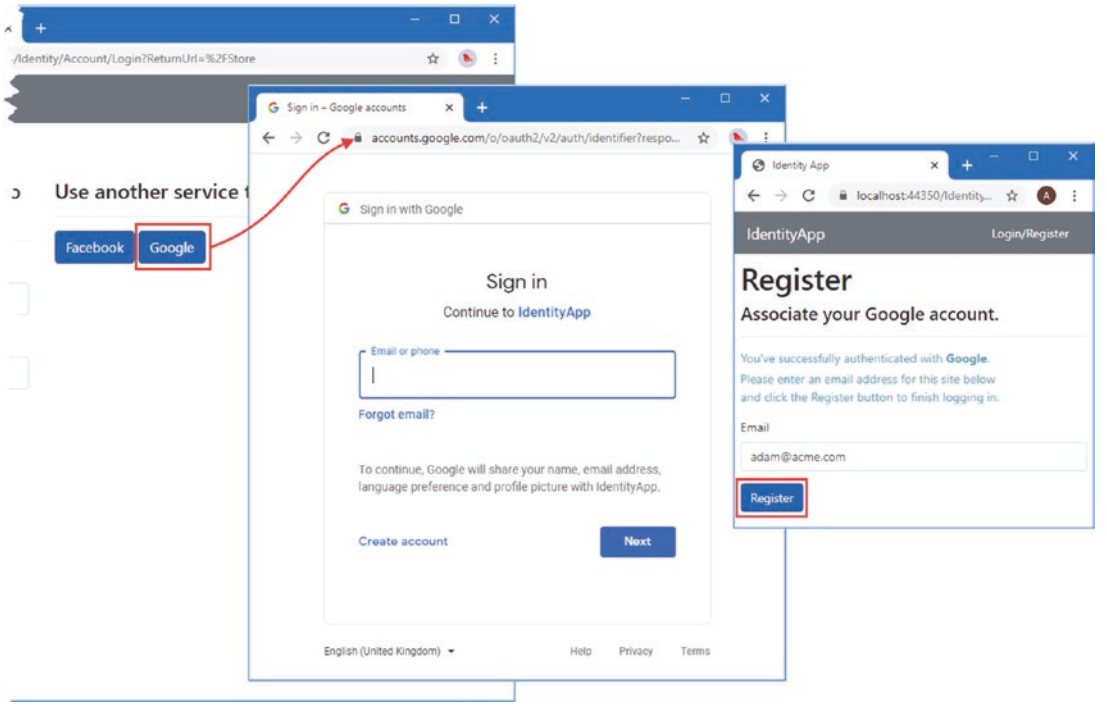

Figure 5-14. *Signing in with Facebook*

As with the previous example, users can enter a different email address from the one associated with the Google account, which means the user has to follow the link in a confirmation email before they can sign into the application again. If you examine the ASP.NET Core console output, you will see a message similar to this one (I have shortened the confirmation code for brevity):

```
---New Email----
To: adam@acme.com
Subject: Confirm your email
Please confirm your account by <a href='https://localhost:44350/Identity/Account/
ConfirmEmail?userId=65e6b14e'>
    clicking here
</a>.
-------
```

Once the email address has been confirmed, the user can sign in again using the Google account.

Configuring Twitter Authentication

To register the application with Twitter, go to `https://developer.twitter.com/en/portal/dashboard` and sign in with a Twitter account. Click the Create Project button, set the project name to Identity Project, and click the Next button. Select a description from the list and click the Next button. Enter a name and click the Complete button to finish the first part of the setup, as shown in Figure 5-15. The name must be unique, so it can take a while to find an available name, but it doesn't matter what it is for this example.

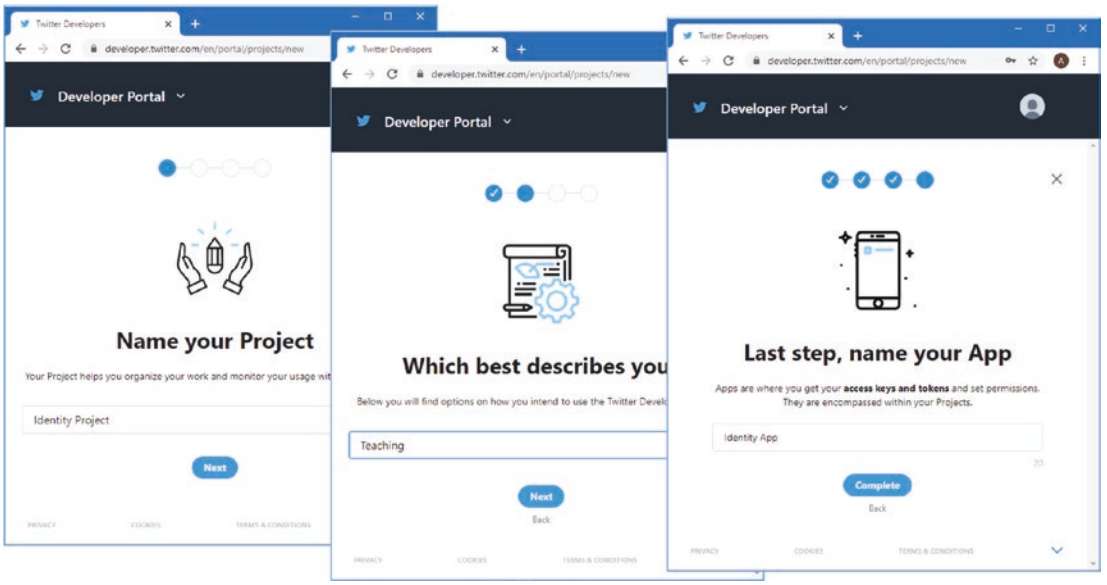

Figure 5-15. *Creating a Twitter application configuration*

When you create the Twitter app, you will be presented with a set of keys, as shown in Figure 5-16. It is important to make a note of the API key and the API key secret (which are how Twitter refers to the client ID and the client secret) because you won't be able to see them again.

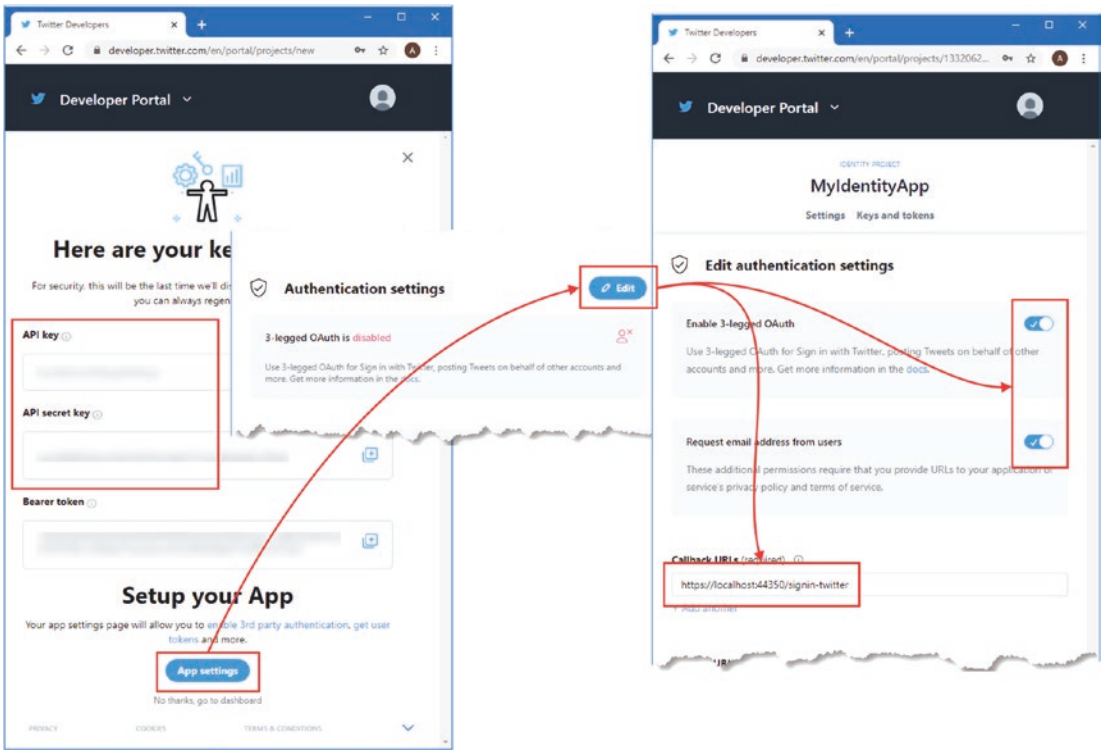

Figure 5-16. *Completing the registration process*

Click the App Settings button and click the Edit button for the Authentication Settings category. Select the Enable 3-Legged OAuth and Request Email Address from Users options and enter `https://localhost:44350/signin-twitter` into the Callback URLs field. You will also have to enter URLs for the website, terms of service, and privacy policy fields. It doesn't matter what URLs you use for this chapter, but you will require URLs with suitable content when you register a real project.

Configuring ASP.NET Core for Twitter Authentication

Run the commands shown in Listing 5-11 to store the client ID and secret using the .NET secrets feature, which ensures that these values won't be included when the source code is committed into a repository.

Listing 5-11. Storing the Twitter Client ID and Secret

```
dotnet user-secrets init
dotnet user-secrets set "Twitter:ApiKey" "<client-id>"

dotnet user-secrets set "Twitter:ApiSecret" "<client-secret>"
```

The first command initializes the secret store. The other commands store the data values so they can be accessed in the application using the ASP.NET Core configuration system. Next, run the command shown in Listing 5-12 in the `IdentityApp` folder to add the package containing Twitter authentication to the project.

Listing 5-12. Adding the Twitter Package

```
dotnet add package Microsoft.AspNetCore.Authentication.Twitter --version 5.0.0
```

To configure the application, add the statements shown in Listing 5-13 to the `Startup` class.

Listing 5-13. Configuring Twitter Authentication in the Startup.cs File in the IdentityApp Folder

```
...
services.AddAuthentication()
    .AddFacebook(opts => {
        opts.AppId = Configuration["Facebook:AppId"];
        opts.AppSecret = Configuration["Facebook:AppSecret"];
    })
    .AddGoogle(opts => {
        opts.ClientId = Configuration["Google:ClientId"];
        opts.ClientSecret = Configuration["Google:ClientSecret"];
    })
    .AddTwitter(opts => {
        opts.ConsumerKey = Configuration["Twitter:ApiKey"];
        opts.ConsumerSecret = Configuration["Twitter:ApiSecret"];
    });
...
```

The `Twitter` method sets up the Twitter authentication handler and is configured using the options pattern with the `TwitterOptions` class. Table 5-12 describes the most important `TwitterOptions` properties.

Table 5-12. *Selected TwitterOptions Properties*

Name	Description
`ConsumerKey`	This property is used to specify the client ID for the application. In the example, I use the value stored in Listing 5-11.
`ConsumerSecret`	This property is used to specify the application's client secret. In the listing, I use the value stored in Listing 5-11.
`RetrieveUserDetails`	When set to `true`, this property requests user data, including the email address, as part of the authentication process. This property isn't required when using the Identity UI package, which allows users to enter an email address.

Restart ASP.NET Core and make sure you are signed out of the application by clicking the Logout link in the header and again in the page that is displayed. Next, click the Level 2 button to trigger a challenge response, and you will see that the Identity UI package has automatically detected the Twitter configuration. Click the Twitter button, and you will be redirected to the Twitter authentication service. Once authenticated, you will be redirected to the application to complete the registration process, as shown in Figure 5-17.

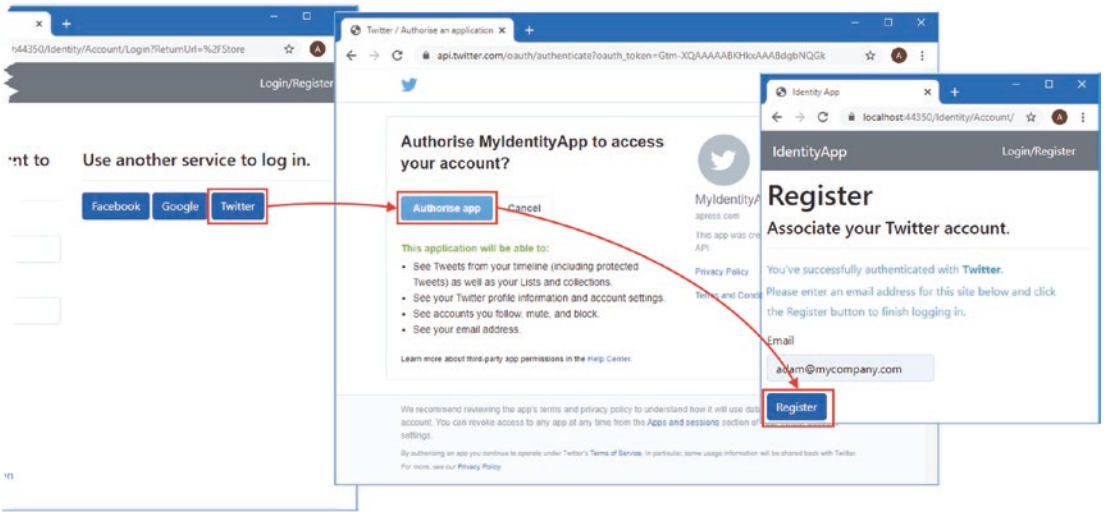

Figure 5-17. *Signing in with Twitter*

As with the previous example, users can enter a different email address from the one associated with the Twitter account, which means the user has to follow the link in a confirmation email before they sign into the application again. If you examine the ASP.NET Core console output, you will see a message similar to this one (I have shortened the confirmation code for brevity):

```
---New Email----
To: adam@mycompany.com
Subject: Confirm your email
Please confirm your account by <a href='https://localhost:44350/Identity/Account/
ConfirmEmail?userId=fa1b86c2'>
    clicking here
</a>.
-------
```

Once the email address has been confirmed, the user can sign in again using the Twitter account.

Summary

In this chapter, I described the Identity configuration options, which determine the validation requirements for accounts, passwords, and control-related features such as lockouts. I also described the process for configuring ASP.NET Core and the Identity UI package to support external authentication services from Google, Facebook, and Twitter. In the next chapter, I show you how to adapt the workflows the Identity UI package provides.