CHAPTER 2

Your First Identity Application

The best way to appreciate a software development framework is to jump right in and use it. In this chapter, I explain how to prepare for ASP.NET Core development and how to create and run an ASP.NET Core application that uses ASP.NET Core Identity.

UPDATES TO THIS BOOK

Microsoft has an active development schedule for .NET and ASP.NET Core, which means that there may be new releases available by the time you read this book. It doesn't seem fair to expect readers to buy a new book every few months, especially since most changes are relatively minor. Instead, I will post free updates to the GitHub repository for this book (`https://github.com/Apress/pro-asp.net-core-identity`) for breaking changes.

This kind of update is an ongoing experiment for me (and for Apress), and it continues to evolve—not least because I don't know what the future major releases of ASP.NET Core will contain—but the goal is to extend the life of this book by supplementing the examples it contains.

I am not making any promises about what the updates will be like, what form they will take, or how long I will produce them before folding them into a new edition of this book. Please keep an open mind and check the repository for this book when new ASP.NET Core versions are released. If you have ideas about how the updates could be improved, then email me at `adam@adam-freeman.com` and let me know.

Setting Up the Development Environment

Identity is used in ASP.NET Core projects, so you should already have everything you need to follow the examples, although you must install the specific version of the .NET SDK I used in this book and the Node.js package that is used in Chapter 12. In this section, I recap the basic setup process that will prepare a development environment suitable for following the examples in this book.

A. Freeman, *Pro ASP.NET Core Identity*, https://doi.org/10.1007/978-1-4842-6858-2_2

■ **Note** This book describes ASP.NET Core Identity development for Windows. It is possible to develop and run ASP.NET Core applications on Linux and macOS, but most readers use Windows, and that is what I have chosen to focus on. Many of the examples in this book rely on LocalDB, which is a Windows-only feature provided by SQL Server that is not available on other platforms. If you want to follow this book on another platform, then you can contact me using the email address in Chapter 1, and I will try to help you get started.

Installing the .NET SDK

You must install the same version of the SDK that I used to follow the examples. You are free to use any SDK version for your projects, but to get the expected results, version 5.0.100 is required. Go to `https://dotnet.microsoft.com/download/dotnet/5.0` and download version 5.0.100, which is the SDK version for .NET 5.0.0. Run the installer; once the installation is complete, open a new PowerShell command prompt from the Windows Start menu and run the command shown in Listing 2-1, which displays a list of the installed .NET Core SDKs.

Listing 2-1. Listing the Installed SDKs

```
dotnet --list-sdks
```

Here is the output from a fresh installation on a Windows machine that has not previously been used for .NET development:

```
5.0.100 [C:\Program Files\dotnet\sdk]
```

Installing Node.js

Node.js is a server-side JavaScript runtime that I use in Chapter 12 to explain how to provide authentication to API clients. You must download the same version of Node.js that I use in this book. Although Node.js is relatively stable, there are still breaking API changes from time to time.

The version I have used is 14.15.4, which is the Long-Term Support release at the time of writing. There may be a later version available by the time you read this, but you should stick to the 14.15.4 release for the examples in this book. A complete set of 14.15.4 installers for Windows is available at `https://nodejs.org/dist/v14.15.4`. Run the installer and ensure that the "npm package manager" option and the two Add to PATH options are selected, as shown in Figure 2-1.

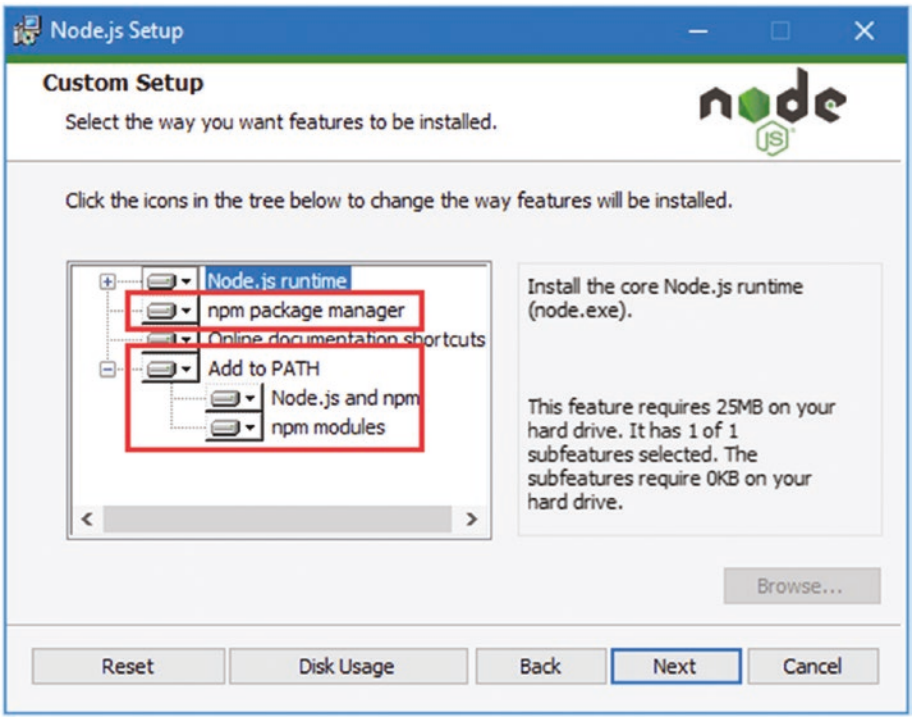

Figure 2-1. *Configuring the Node installation*

When the installation is complete, run the command shown in Listing 2-2.

Listing 2-2. Running Node.js

```
node -v
```

If the installation has gone as it should, then you will see the following version number displayed:

```
v14.15.4
```

The Node.js installer includes the Node Package Manager (NPM), which is used to manage the packages in a project. Run the command shown in Listing 2-3 to ensure that NPM is working.

Listing 2-3. Running NPM

```
npm -v
```

If everything is working as it should, then you will see the following version number:

```
6.14.10
```

Installing a Code Editor

You can use any code editor to follow the examples for this book. The most popular choices are Visual Studio and Visual Studio Code, for which I provide installation instructions in the sections that follow. Visual Studio has better support for C# development, but Visual Studio Code is lighter and quicker. It doesn't matter which one you pick—or whether you use a different editor entirely—because all the commands for building and running projects are run from the command line.

But, regardless of your editor, you will require the exact version of the .NET SDK, and you must ensure that LocalDB is installed (see the following instructions for details).

MUDDLING THROUGH THE MICROSOFT NAMING SCHEME

Microsoft doesn't seem able to settle on a naming convention. The .NET Framework evolved into .NET Core, which has now become .NET (just .NET, without *Framework* or *Core*). To make things more confusing, ASP.NET became ASP.NET Core but hasn't been renamed to follow the change from .NET Core to .NET.

The name changes have been one part of a years-long period of disruptive change, U-turns, confusion, and a general lack of leadership and direction, all of which have trickled down to developers. I am sure that there is someone at Microsoft who thinks this has all been worthwhile, but I have yet to meet them. This is a shame because I have a wide-ranging and expletive-filled rant on this topic that I have been saving for that occasion.

ASP.NET Core and ASP.NET Core Identity have remained relatively stable through the transition from .NET Core to .NET, and the main impact for this book is that I may have used the wrong name of .NET in places. So, if you see a reference to .NET Core, please just take a pencil and cross out the Core part.

Installing Visual Studio

Before installing Visual Studio, make sure you have installed the .NET SDK as described in the previous section. ASP.NET Core 5 requires Visual Studio 2019. I use the free Visual Studio 2019 Community Edition, which can be downloaded from `www.visualstudio.com`. Run the installer, and you will see the prompt shown in Figure 2-2.

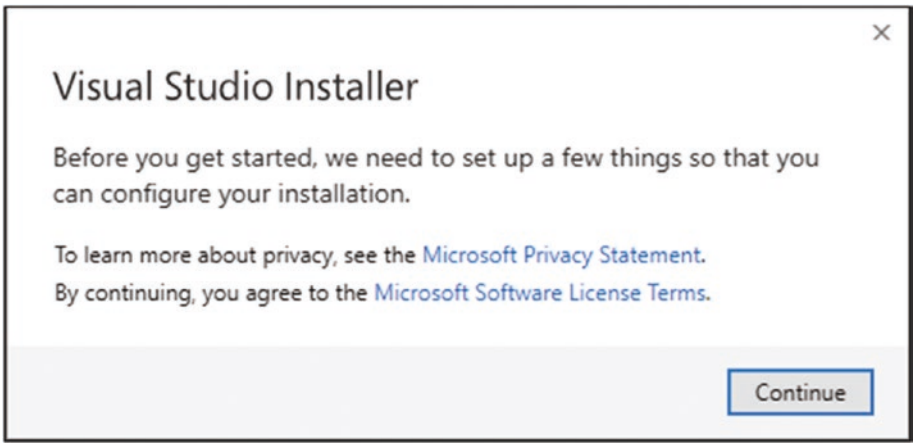

Figure 2-2. *Starting the Visual Studio installer*

Click the Continue button, and the installer will download the installation files, as shown in Figure 2-3.

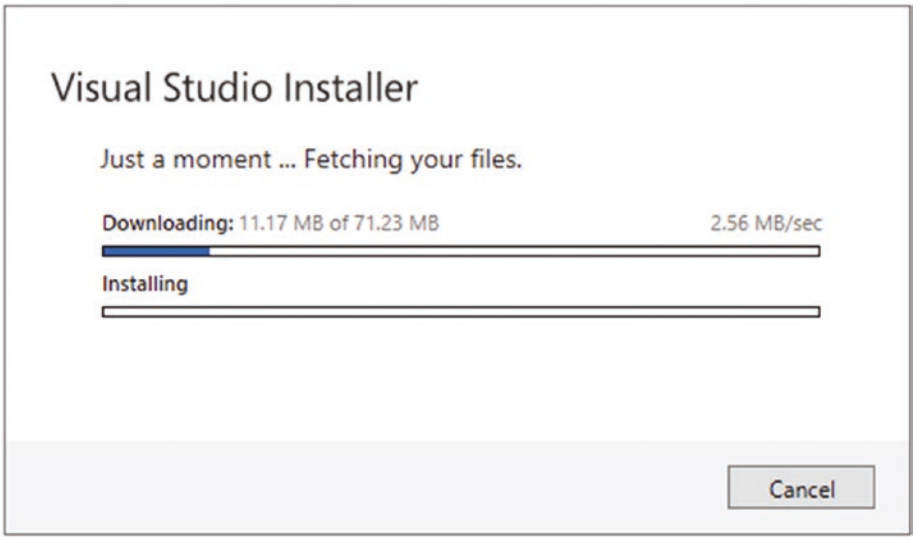

Figure 2-3. *Downloading the Visual Studio installer files*

When the installer files have been downloaded, you will be presented with a set of installation options, grouped into workloads. Ensure that the "ASP.NET and web development" workload is checked, as shown in Figure 2-4.

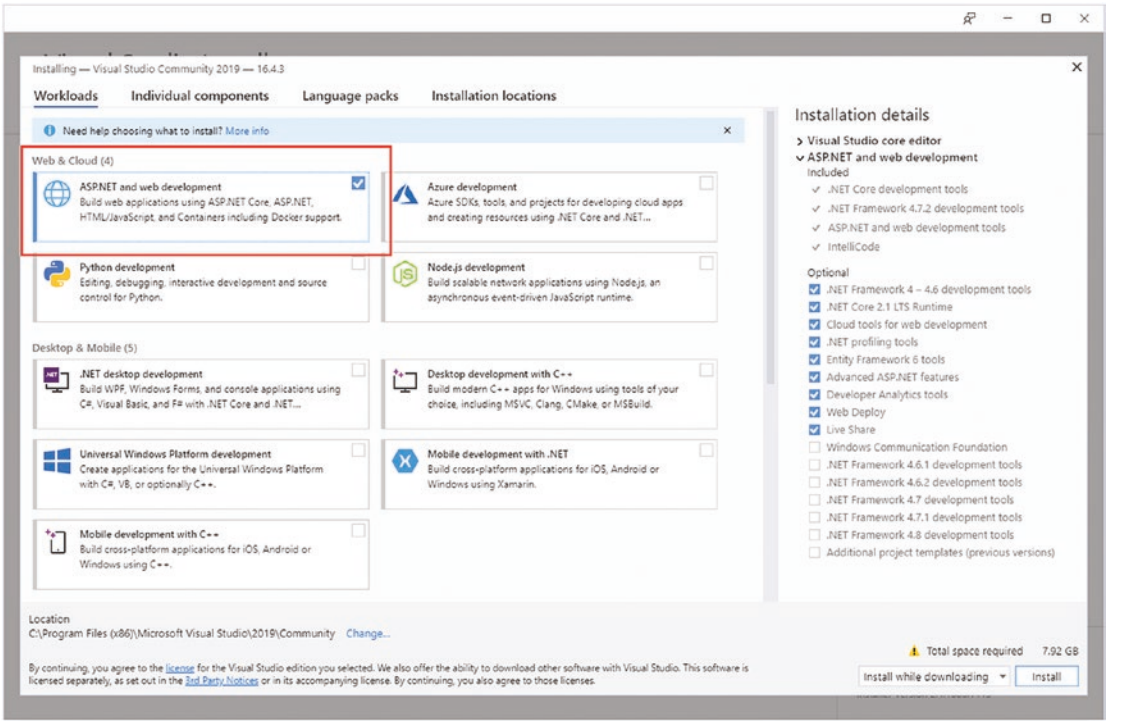

Figure 2-4. *Selecting the workload*

Select the "Individual components" section at the top of the window and ensure the SQL Server Express 2016 LocalDB option is checked, as shown in Figure 2-5. This is the database component that I will be using to store data.

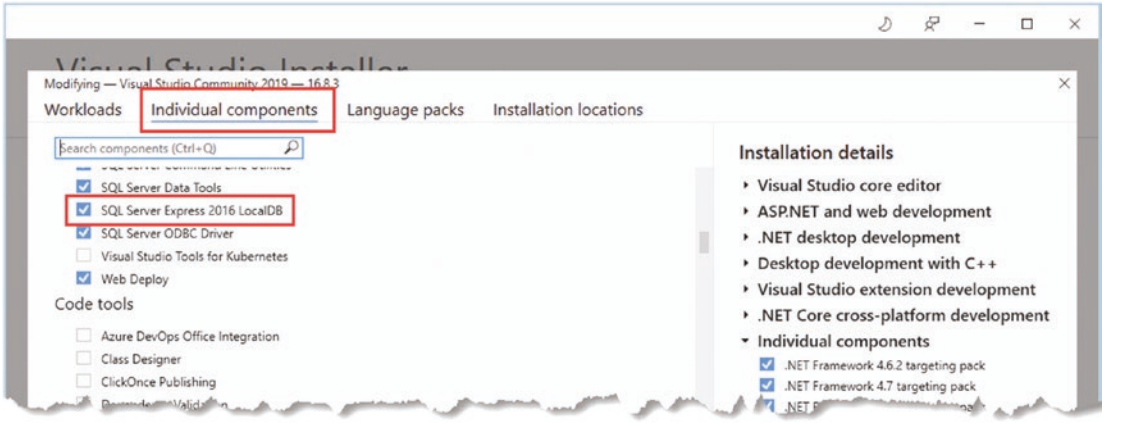

Figure 2-5. *Ensuring LocalDB is installed*

Click the Install button, and the files required for the selected workload will be downloaded and installed. To complete the installation, a reboot is required, as shown in Figure 2-6.

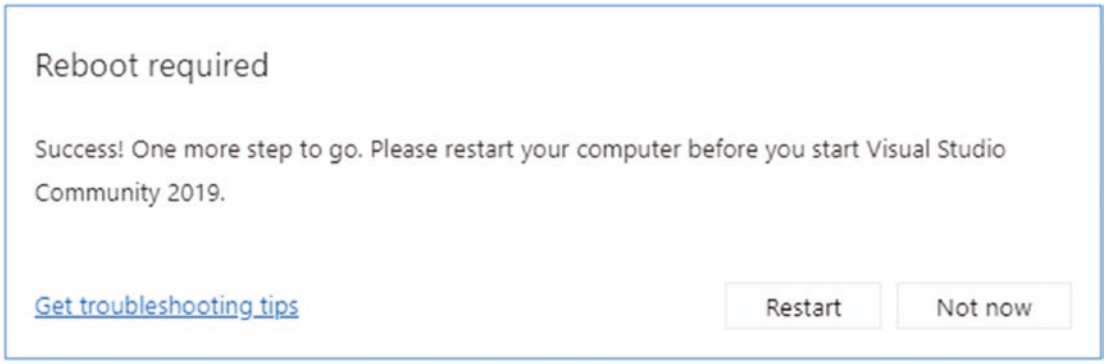

Figure 2-6. *Completing the installation*

Installing Visual Studio Code

If you have chosen to use Visual Studio Code, download the installer from `https://code.visualstudio.com`. No specific version is required, and you should select the current stable build. Run the installer and ensure you select the Add to PATH option, as shown in Figure 2-7.

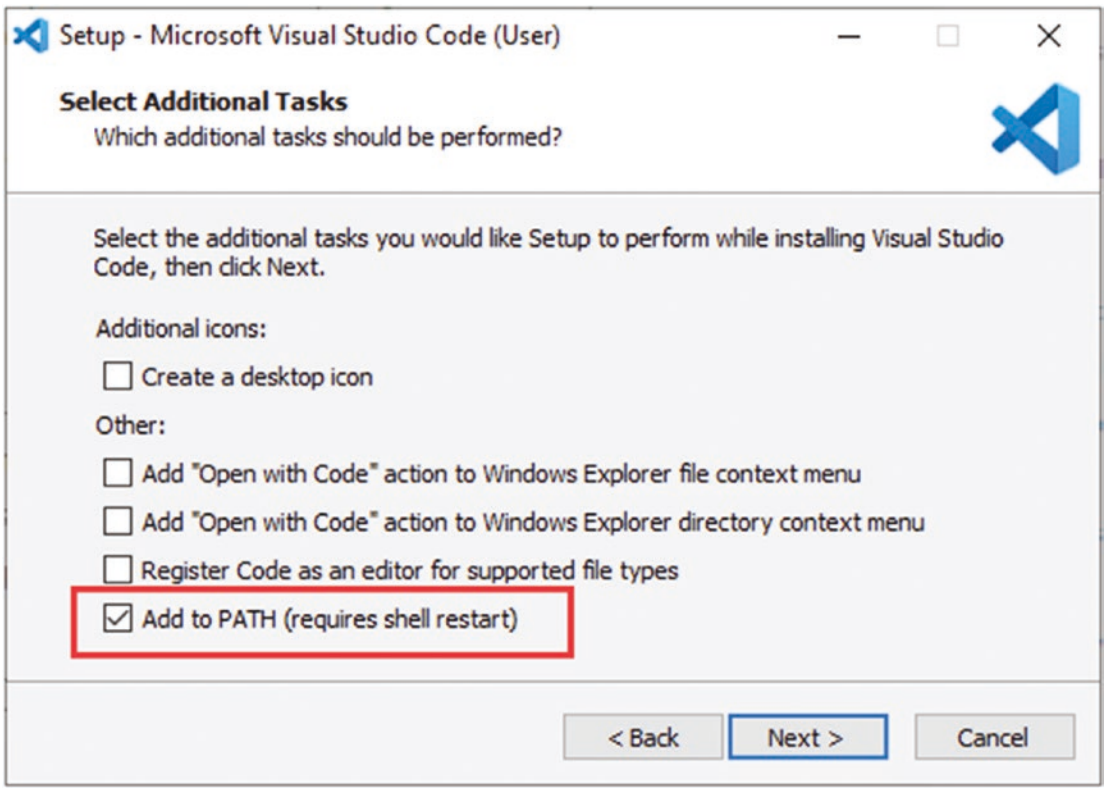

Figure 2-7. *Configuring the Visual Studio Code installation*

Installing SQL Server LocalDB

Many of the examples in this book require LocalDB, which is a zero-configuration version of SQL Server that can be installed as part of the SQL Server Express edition, which is available for use without charge from `https://www.microsoft.com/en-in/sql-server/sql-server-downloads`. Download and run the Express edition installer and select the Custom option, as shown in Figure 2-8.

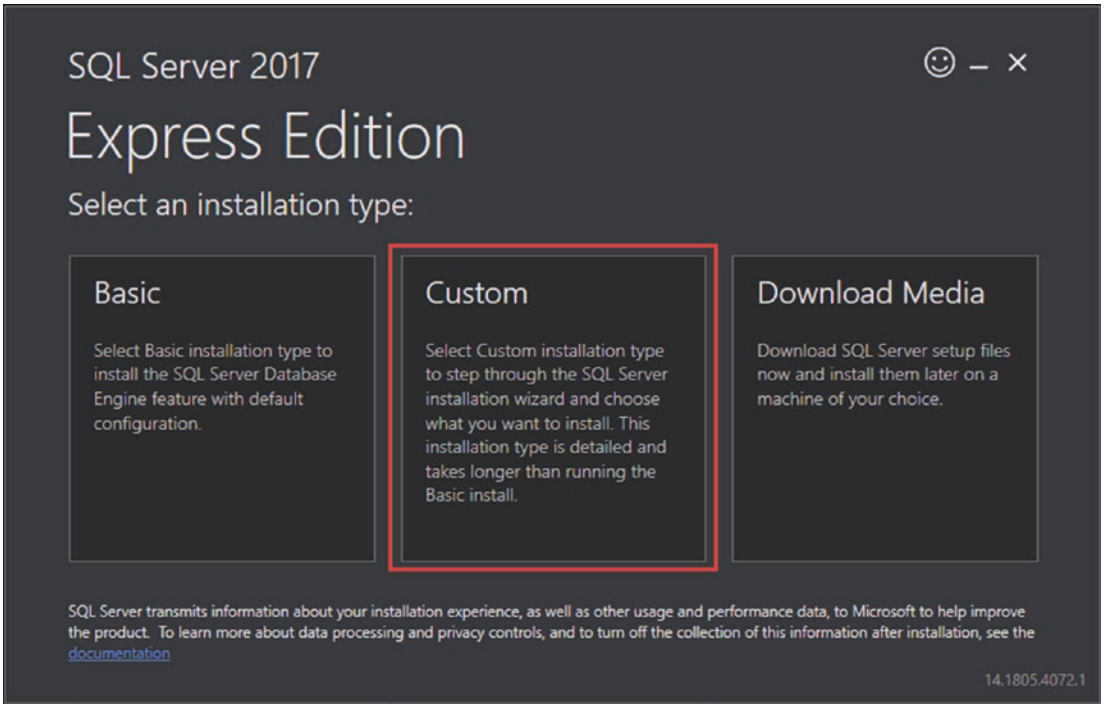

Figure 2-8. *Selecting the installation option for SQL Server*

Once you have selected the Custom option, you will be prompted to select a download location for the installation files. Click the Install button, and the download will begin.

When prompted, select the option to create a new SQL Server installation, as shown in Figure 2-9.

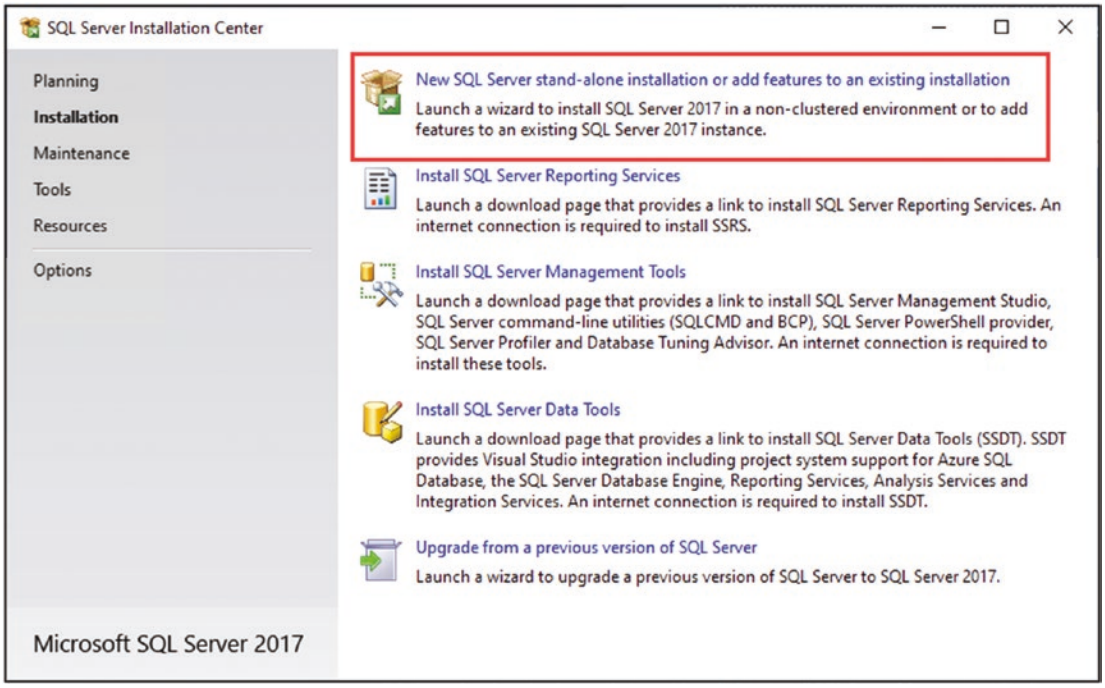

Figure 2-9. *Selecting an installation option*

Work through the installation process, selecting the default options as they are presented. When you reach the Feature Selection page, ensure that the LocalDB option is selected, as shown in Figure 2-10. (You may want to deselect the options for R and Python, which are not used in this book and take a long time to download and install.)

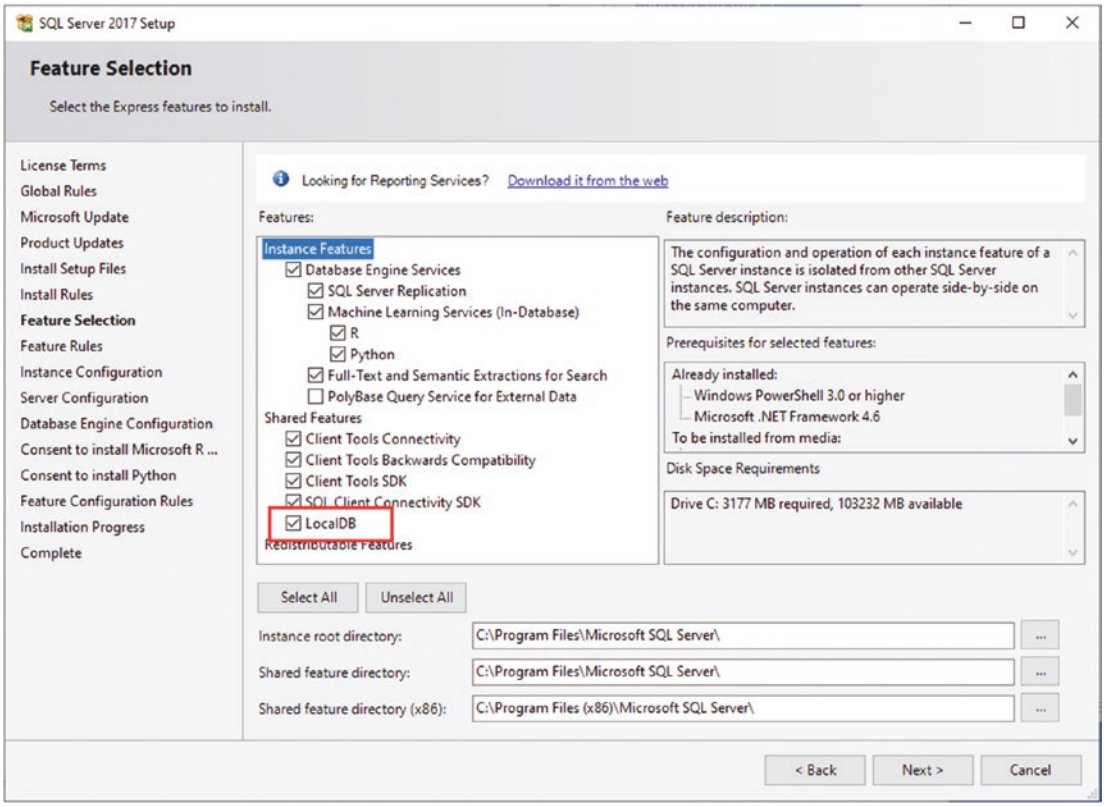

Figure 2-10. *Selecting the LocalDB feature*

On the Instance Configuration page, select the "Default instance" option, as shown in Figure 2-11.

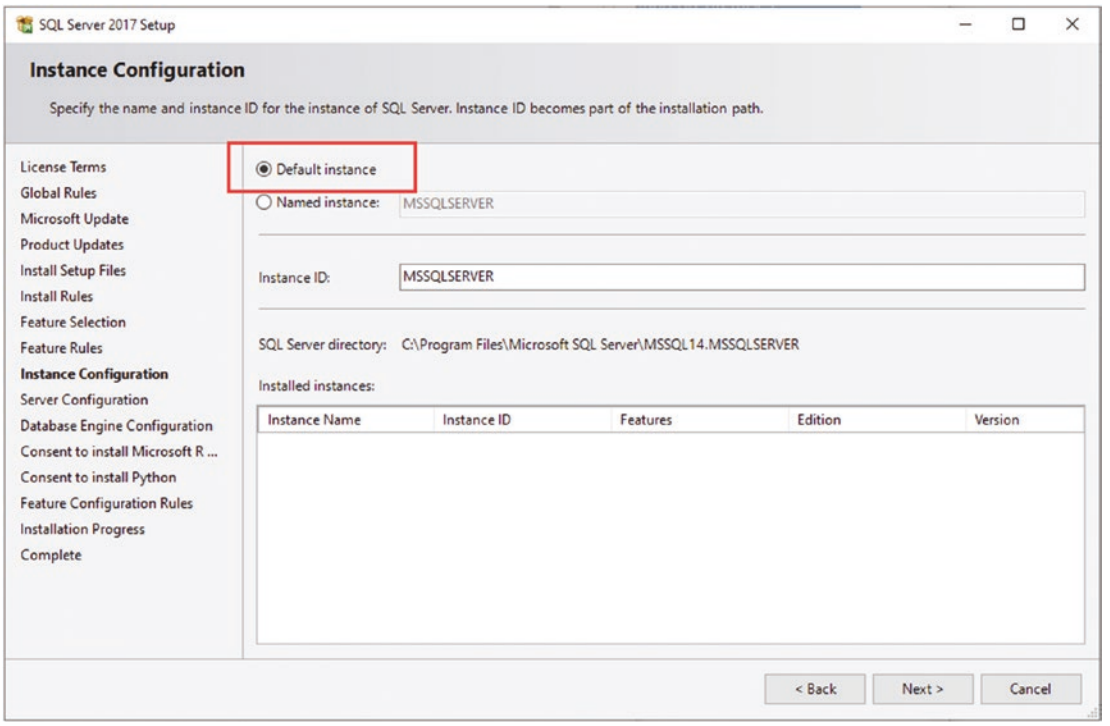

Figure 2-11. *Configuring the database*

Continue to work through the installation process, selecting the default values. Once the installation is complete, install the latest cumulative update for SQL Server. At the time of writing, the latest update is available at `https://support.microsoft.com/en-us/help/4577467/kb4577467-cumulative-update-22-for-sql-server-2017`, although newer updates may have been released by the time you read this chapter.

■ **Caution** It can be tempting to skip the update stage, but it is important to perform this step to get the expected results from the examples in this book. As an example, the base installation of SQL Server has a bug that prevents LocalDB from creating database files, which will cause problems when you create a project later in this chapter.

Creating an Application with Identity

In this section, I am going to create an ASP.NET Core project that uses ASP.NET Core Identity. This isn't a complex process because I have chosen a project that fits neatly with the default Identity, which is an application where users can register themselves and that requires no administration tools. The application is a to-do list, with support for multiple users, and the result is a simple demonstration of how Identity can be used with minimum configuration or interference with the normal ASP.NET Core development practices.

■ **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.

Open a new PowerShell command prompt, navigate to a convenient location, and run the commands shown in Listing 2-4 to create a new project named IdentityTodo. The second command contains a trailing backtick so that both lines will be treated as part of the same command by PowerShell.

Listing 2-4. Creating a New Project

```
dotnet new globaljson --sdk-version 5.0.100 --output IdentityTodo
dotnet new webapp --auth Individual --use-local-db true `
  --output IdentityTodo --framework net5.0
dotnet new sln -o IdentityTodo
dotnet sln IdentityTodo add IdentityTodo
```

These commands create a basic project that includes ASP.NET Core Identity. Once you have created the project, run the commands shown in Listing 2-5 to navigate to the project folder and build the project.

Listing 2-5. Building the Project

```
cd IdentityTodo
dotnet build
```

If the build is successful, there will be no output from the dotnet build command. You may receive this error instead:

```
Could not execute because the application was not found or a compatible .NET SDK is not
installed.
```

This means you have not installed the correct version of the .NET SDK. Return to the instructions at the start of this chapter and install the SDK, which is required even when you are using Visual Studio.

Preparing the Project

I use the command-line tools to build and run the ASP.NET Core projects throughout this book. Open the project (by opening the IdentityTodo.sln file with Visual Studio or the IdentityTodo folder in Visual Studio Code) and change the contents of the launchSettings.json file in the Properties folder, as shown in Listing 2-6, to set the ports that will be used to listen for HTTP requests.

Listing 2-6. Setting Ports in the launchSettings.json File in the Properties Folder

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
```

```
    "iisExpress": {
      "applicationUrl": "http://localhost:5000",
      "sslPort": 44350
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IdentityTodo": {
      "commandName": "Project",
      "dotnetRunMessages": "true",
      "launchBrowser": true,
      "applicationUrl": "https://localhost:44350;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

I use port 5000 for HTTP requests and port 44350 for HTTPS throughout this book, and these changes apply the ports for both the command line and when the project is started with Visual Studio.

Creating the Data Model

Add a class file named `TodoItem.cs` to the `Data` folder with the code shown in Listing 2-7. I use a folder named `Models` in later chapters, which has been the ASP.NET Core convention, but for this chapter, I am going to use the folder structure provided by the template.

Listing 2-7. The Contents of the TodoItem.cs File in the Data Folder

```
namespace IdentityTodo.Data {

    public class TodoItem {

        public long Id { get; set; }

        public string Task { get; set; }

        public bool Complete { get; set; }

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

This class will be used to represent to-do items, which will be stored in a database using Entity Framework Core. Entity Framework Core is also used to store Identity data, and the project template has created a database context class.

I usually recommend keeping the Identity data in a separate database, but for this chapter, I am going to focus on simplicity and use the same database for all the data. Add the property shown in Listing 2-8 to the `ApplicationDbContext.cs` file in the `Data` folder.

Listing 2-8. Adding a Property in the ApplicationDbContext.cs File in the Data Folder

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;

namespace IdentityTodo.Data {

    public class ApplicationDbContext : IdentityDbContext {

        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options) {}

        public DbSet<TodoItem> TodoItems { get; set; }
    }
}
```

Creating and Applying the Database Migrations

The example application requires two databases: one to store the to-do items and one for the Identity user accounts. To install the command-line tool package that will be used to manage the databases, use a PowerShell command prompt to run the command shown in Listing 2-9.

Listing 2-9. Installing the Entity Framework Core Tools Package

```
dotnet tool uninstall --global dotnet-ef
dotnet tool install --global dotnet-ef --version 5.0.0
```

The first command removes any existing version of the Entity Framework Core tools package, and you may see an error if no version of this package is installed. The second command installs the version of the tools package required for the examples in this book.

Use a PowerShell prompt to run the commands shown in Listing 2-10 in the `IdentityTodo` folder to create a database migration that will add support for storing to-do items.

Listing 2-10. Creating the Database Migrations

```
dotnet ef migrations add AddTodos
```

Run the commands shown in Listing 2-11 in the IdentityTodo folder to remove any existing database, which may have been previously created if you are re-reading this chapter, and create a new database using the migration created in Listing 2-11.

Listing 2-11. Creating the Database

```
dotnet ef database drop --force
dotnet ef database update
```

Configuring ASP.NET Core Identity

A configuration change is required to prepare ASP.NET Core Identity, as shown in Listing 2-12.

Listing 2-12. Configuring the Application in the Startup.cs File in the IdentityTodo Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.UI;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.HttpsPolicy;
using Microsoft.EntityFrameworkCore;
using IdentityTodo.Data;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace IdentityTodo {
    public class Startup {
        public Startup(IConfiguration configuration) {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

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

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

```
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            if (env.IsDevelopment()) {
                app.UseDeveloperExceptionPage();
                app.UseMigrationsEndPoint();
            } else {
                app.UseExceptionHandler("/Error");
                app.UseHsts();
            }

            app.UseHttpsRedirection();
            app.UseStaticFiles();

            app.UseRouting();

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

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

Identity is added to the application using the `AddDefaultIdentity` extension method, and the default configuration created by the project template sets the configuration so that user accounts cannot be used until they are confirmed, which requires the user to click a link they are emailed. I explain the confirmation process in Chapters 8 and 9 and describe it in detail in Chapter 17, but I don't want to use it in this chapter, so I have set the `RequireConfirmedAccount` configuration option to `false`. (All the Identity configuration options are described in Chapter 5.)

Creating the Application Content

To present the user with their list of to-do items, replace the contents of the `Index.cshtml` file in the `Pages` folder with those shown in Listing 2-13.

Listing 2-13. Replacing the Contents of the Index.cshtml File in the Pages Folder

```
@page
@model IndexModel
@{
    ViewData["Title"] = "To Do List";
}

<h2 class="text-center">To Do List</h2>
<h4 class="text-center">(@User.Identity.Name)</h4>

<form method="post" asp-page-handler="ShowComplete" class="m-2">
    <div class="form-check">
        <input type="checkbox" class="form-check-input" asp-for="ShowComplete"
               onchange="this.form.submit()"/>
```

```
        <label class="form-check-label">Show Completed Items</label>
    </div>
</form>

<table class="table table-sm table-striped table-bordered m-2">
    <thead><tr><th>Task</th><th/></tr></thead>
    <tbody>
        @if (Model.TodoItems.Count() == 0) {
            <tr>
                <td colspan="2" class="text-center py-4">
                    You have done everything!
                </td>
            </tr>
        } else {
            @foreach (TodoItem item in Model.TodoItems) {
                <tr>
                    <td class="p-2">@item.Task</td>
                    <td class="text-center py-2">
                        <form method="post" asp-page-handler="MarkItem">
                            <input type="hidden" name="id" value="@item.Id" />
                            <input type="hidden" asp-for="ShowComplete" />
                            <button type="submit" class="btn btn-sm btn-secondary">
                                @(item.Complete ? "Mark Not Done" : "Done")
                            </button>
                        </form>
                    </td>
                </tr>
            }
        }
    </tbody>
    <tfoot>
        <tr>
            <td class="pt-4">
                <form method="post" asp-page-handler="AddItem" id="addItem">
                    <input type="hidden" asp-for="ShowComplete" />
                    <input name="task" placeholder="Enter new to do"
                        class="form-control" />
                </form>
            </td>
            <td class="text-center pt-4">
                <button type="submit" form="addItem"
                        class="btn btn-sm btn-secondary">
                    Add
                </button>
            </td>
        </tr>
    </tfoot>
</table>
```

This content presents the user with a table containing their to-do list, along with the ability to add items to the list, mark items as done, and include completed items in the table. To define the features that support the content in Listing 2-13, replace the contents of the `Index.cshtml.cs` file in the `Pages` folder with the code shown in Listing 2-14.

Listing 2-14. Replacing the Contents of the Index.cshtml.cs File in the Pages Folder

```
using IdentityTodo.Data;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace IdentityTodo.Pages {

    [Authorize]
    public class IndexModel : PageModel {
        private ApplicationDbContext Context;

        public IndexModel(ApplicationDbContext ctx) {
            Context = ctx;
        }

        [BindProperty(SupportsGet = true)]
        public bool ShowComplete { get; set; }

        public IEnumerable<TodoItem> TodoItems { get; set; }

        public void OnGet() {
            TodoItems = Context.TodoItems
                .Where(t => t.Owner == User.Identity.Name).OrderBy(t => t.Task);
            if (!ShowComplete) {
                TodoItems = TodoItems.Where(t => !t.Complete);
            }
            TodoItems = TodoItems.ToList();
        }

        public IActionResult OnPostShowComplete() {
            return RedirectToPage(new { ShowComplete });
        }

        public async Task<IActionResult> OnPostAddItemAsync(string task) {
            if (!string.IsNullOrEmpty(task)) {
                TodoItem item = new TodoItem {
                    Task = task,
                    Owner = User.Identity.Name,
                    Complete = false
                };
```

```
                await Context.AddAsync(item);
                await Context.SaveChangesAsync();
            }
            return RedirectToPage(new { ShowComplete });
        }

        public async Task<IActionResult> OnPostMarkItemAsync(long id) {
            TodoItem item = Context.TodoItems.Find(id);
            if (item != null) {
                item.Complete = !item.Complete;
                await Context.SaveChangesAsync();
            }
            return RedirectToPage(new { ShowComplete });
        }
    }
}
```

There is no direct use of Identity in the code in Listing 2-14 or the Razor content in Listing 2-13. That's because Identity fits neatly into the features provided by the ASP.NET Core platform. The `Authorize` attribute that decorates the page model class in Listing 2-14 tells ASP.NET Core that only authenticated users should be able to access this Razor Page, and the configuration in the `Startup` class, shown in Listing 2-12, has set up Identity as the means by which user accounts are created and used.

Running the Example Application

Start ASP.NET Core by running the command shown in Listing 2-15 in the `IdentityTodo` folder.

Listing 2-15. Running the Example Application

```
dotnet run
```

The project will be compiled and started. After a few seconds, you will see messages that ASP.NET Core is listening for requests on the configured ports, like this:

```
...
Building...
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:44350
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\IdentityTodo
...
```

Open a new browser window and request `https://localhost:44350`. This request targets the `Index.cshtml` Razor Page, but since that page has been decorated with the `Authorize` attribute, you will be redirected to a page that prompts you to sign in or create an account, as shown in Figure 2-12. This content is provided by the Identity UI package, which provides a standard set of Razor Pages for managing accounts. There are some placeholder items, such as a message about setting up external providers, which I explain in Chapter 11.

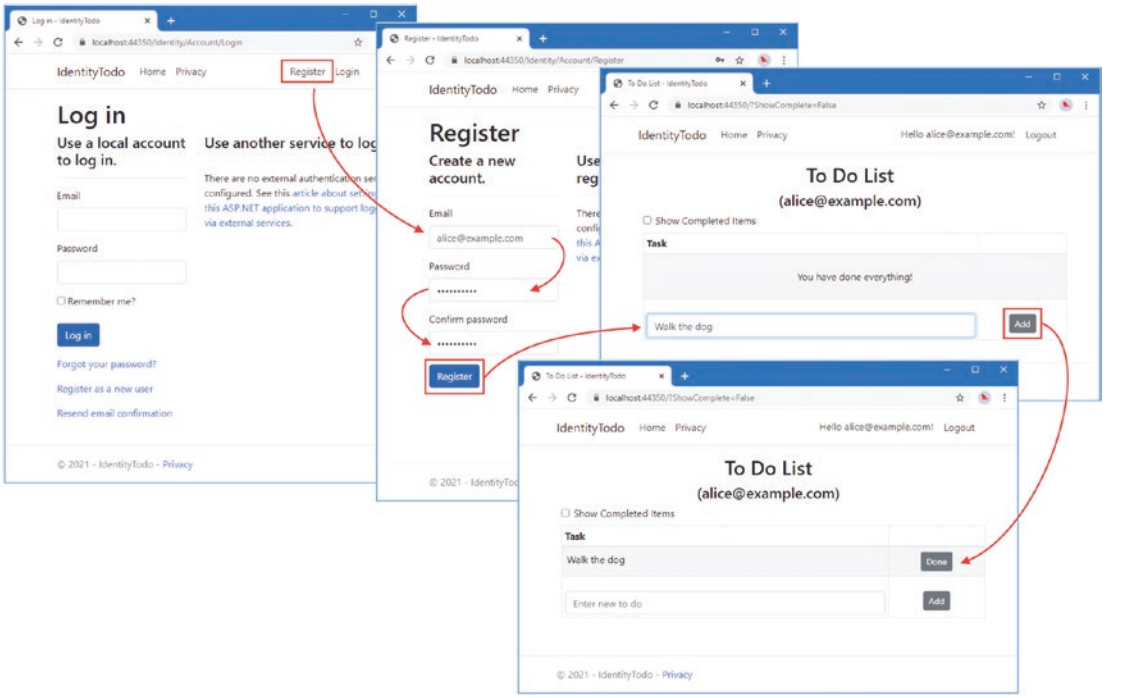

Figure 2-12. *The Identity UI sign-in screen*

Click the Register link in the top right of the browser window to navigate to the Razor Page used to register a new account. Enter **alice@example.com** as the email address, and enter the password **MySecret1$** in the Password and Confirm Password fields. (Identity has a punitive default password policy, which I explain how to change in Chapter 8.) Click the Register button, and a new user account will be created and used to sign into the application. Enter a task in the text field, such as **Walk the dog**, and click the Add button. A new item will be stored in the database.

Click the `alice@example.com` email address shown at the top of the browser window, and you will be presented with an account self-management portal, as shown in Figure 2-13. This is another feature provided by the Identity UI package and can be adapted to suit the needs of different projects, as I explain in Chapter 6. If the Identity UI package doesn't suit your project, Identity also provides a complete API for creating custom workflows, which I describe in Chapters 7 to 11.

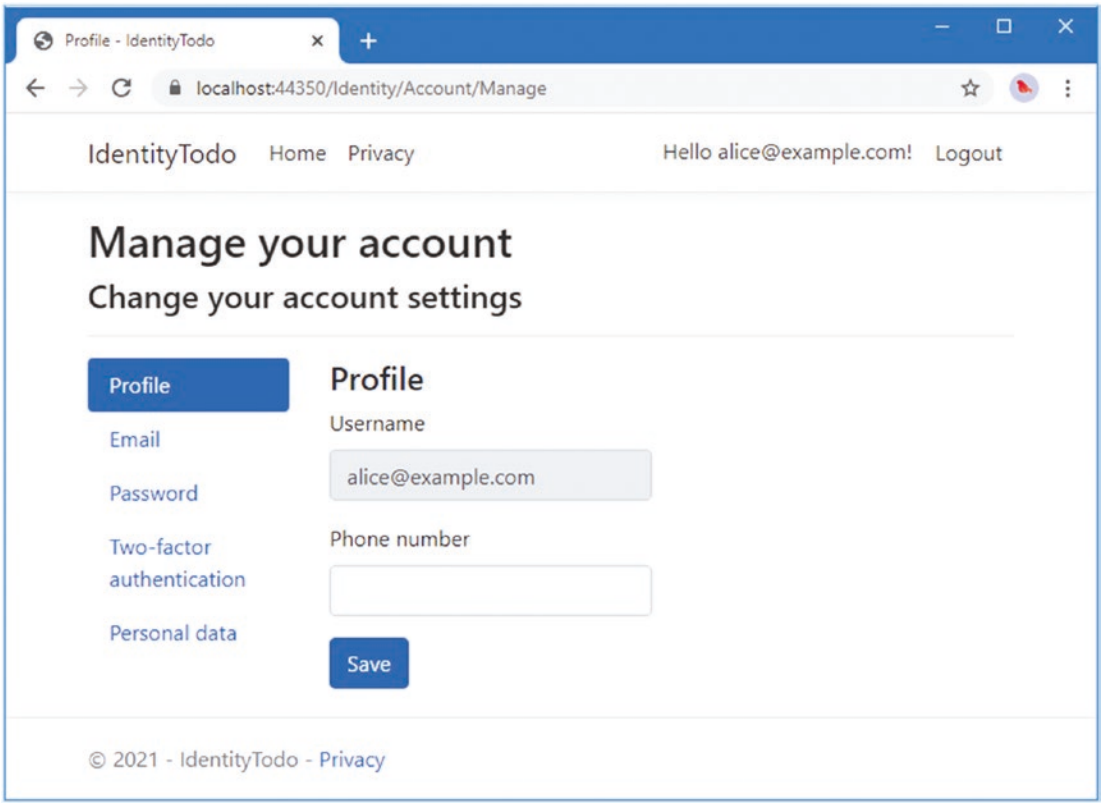

Figure 2-13. *The IdentityUI self-management portal*

Summary

In this chapter, I described the tools required for ASP.NET Core and ASP.NET Core Identity development. I created an application using a project template that includes Identity and demonstrated the features that Identity can offer with minimal configuration. In the next chapter, I create a more complex project and use it to start exploring the ASP.NET Core Identity features in detail.