CHAPTER 3

Creating the Example Project

In the previous chapter, I created an ASP.NET Core application that used Identity in the simplest way possible, which is to include Identity when the project is created and accept the default configuration.

But to explain how Identity works—and make it much more useful—I need a project that doesn't contain Identity at first and doesn't fit as neatly into the pattern that the default Identity configuration expects.

This application isn't complex. I need three types of application feature: features that can be accessed by anyone, features that can be accessed only once a user signs in, and features that can be accessed only by administrators.

Creating the Project

Open a new PowerShell command prompt from the Windows Start menu and run the commands shown in Listing 3-1.

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 3-1. Creating the Project

```
dotnet new globaljson --sdk-version 5.0.100 --output IdentityApp
dotnet new web --no-https --output IdentityApp --framework net5.0
dotnet new sln -o IdentityApp
dotnet sln IdentityApp add IdentityApp
```

Open the project for editing and make the changes shown in Listing 3-2 to the `launchSettings.json` file in the `Properties` folder to set the port that will be used to handle HTTP and requests.

Listing 3-2. Configuring HTTP Ports in the launchSettings.json File in the Properties Folder

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:5000",
```


A. Freeman, *Pro ASP.NET Core Identity*, https://doi.org/10.1007/978-1-4842-6858-2_3

```
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IdentityApp": {
      "commandName": "Project",
      "dotnetRunMessages": "true",
      "launchBrowser": true,
      "applicationUrl": "http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

Installing the Bootstrap CSS Framework

Use a command prompt to run the commands shown in Listing 3-3 in the IdentityApp folder to initialize the Library Manager tool and install the Bootstrap CSS package, which I use to style HTML content.

Listing 3-3. Installing the Client-Side CSS Package

```
dotnet tool uninstall --global Microsoft.Web.LibraryManager.Cli
dotnet tool install --global Microsoft.Web.LibraryManager.Cli --version 2.1.113
libman init -p cdnjs
libman install twitter-bootstrap@4.5.0 -d wwwroot/lib/twitter-bootstrap
```

Install Entity Framework Core

Open a new PowerShell command prompt and run the commands shown in Listing 3-4 in the IdentityApp folder.

Listing 3-4. Installing the Entity Framework Core Packages

```
dotnet add package Microsoft.EntityFrameworkCore.Design --version 5.0.0
dotnet add package Microsoft.EntityFrameworkCore.SqlServer --version 5.0.0
```

Entity Framework Core relies on a global tool package to manage databases and schemas. Run the commands shown in Listing 3-5 to remove any existing version of the tool package and to install the version required for the examples in this book.

Listing 3-5. Installing the Entity Framework Core Tools Package

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

Defining a Connection String

Add the configuration setting shown in Listing 3-6 to the `appsettings.json` file in the `IdentityApp` folder. This setting defines a connection string that identifies the database that Entity Framework Core will use to store `Product` data.

■ **Tip** Connection strings must be expressed as a single unbroken line, which is fine in the code editor but doesn't fit on the printed page and is the cause of the awkward formatting in Listing 3-6. When you define the connection string in your projects, make sure the value of the `AppDataConnection` item is on a single line.

Listing 3-6. Defining a Connection String in the appsettings.json File in the IdentityApp Folder

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "AppDataConnection": "Server=(localdb)\\MSSQLLocalDB;Database=IdentityAppData;
    MultipleActiveResultSets=true"
  }
}
```

Creating the Data Model

Create the `IdentityApp/Models` folder and add to it a class file named `Product.cs` with the code shown in Listing 3-7.

Listing 3-7. The Contents of the Product.cs File in the Models Folder

```
using System.ComponentModel.DataAnnotations.Schema;
namespace IdentityApp.Models {
```

```
    public class Product {
        public long Id { get; set; }

        public string Name { get; set; }

        [Column(TypeName = "decimal(8, 2)")]
        public decimal Price { get; set; }

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

The Product class has an Id property, which will be used as the primary database key, along with Name, Price, and Category properties. I applied the Column attribute to the Price property so that Entity Framework Core will know which numeric type to use when creating the database schema for storing Product objects.

To create the Entity Framework Core context class, add a file named ProductDbContext.cs to the Models folder with the code shown in Listing 3-8. This class provides access to the Product objects that Entity Framework Core stores in the database and seeds the database with sample data.

Listing 3-8. The Contents of the ProductDbContext.cs File in the Models Folder

```
using Microsoft.EntityFrameworkCore;

namespace IdentityApp.Models {
    public class ProductDbContext: DbContext {

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

        public DbSet<Product> Products { get; set; }

        protected override void OnModelCreating(ModelBuilder builder) {
            builder.Entity<Product>().HasData(
                new Product { Id = 1, Name = "Kayak",
                     Category = "Watersports", Price = 275 },
                new Product { Id = 2, Name = "Lifejacket",
                    Category = "Watersports", Price = 48.95m },
                new Product { Id = 3, Name = "Soccer Ball",
                    Category = "Soccer", Price = 19.50m },
                new Product { Id = 4, Name = "Corner Flags",
                    Category = "Soccer", Price = 34.95m },
                new Product { Id = 5, Name = "Stadium",
                    Category = "Soccer", Price = 79500 },
                new Product { Id = 6, Name = "Thinking Cap",
                    Category = "Chess", Price = 16 },
                new Product { Id = 7, Name = "Unsteady Chair",
                    Category = "Chess", Price = 29.95m },
                new Product { Id = 8, Name = "Human Chess Board",
                    Category = "Chess", Price = 75 },
```

```
                new Product { Id = 9, Name = "Bling-Bling King",
                    Category = "Chess", Price = 1200});
        }
    }
}
```

Creating MVC Controllers and Views

Create the `IdentityApp/Controllers` folder and add to it a class file named `HomeController.cs` with the content in Listing 3-9.

Listing 3-9. The Contents of the HomeController.cs File in the Controllers Folder

```
using IdentityApp.Models;
using Microsoft.AspNetCore.Mvc;

namespace IdentityApp.Controllers {

    public class HomeController : Controller {
        private ProductDbContext DbContext;

        public HomeController(ProductDbContext ctx) => DbContext = ctx;

        public IActionResult Index() => View(DbContext.Products);
    }
}
```

This controller will present the first level of access, which will be available to anyone. To create the corresponding view, create the `IdentityApp/Views/Home` folder and add to it a Razor View named `Index.cshtml` file with the content shown in Listing 3-10.

Listing 3-10. The Contents of the Index.cshtml File in the Views/Home Folder

```
@model IQueryable<Product>

<h4 class="bg-primary text-white text-center p-2">MVC - Level 1 - Anyone</h4>

<div class="text-center">
    <h6 class="p-2">
        The store contains @Model.Count() products.
    </h6>
</div>
```

Add a class file named `StoreController.cs` in the `Controllers` folder with the content shown in Listing 3-11. This controller will present the second level of access, which is available to users who are signed into the application.

Listing 3-11. The Contents of the StoreController.cs File in the Controllers Folder

```
using IdentityApp.Models;
using Microsoft.AspNetCore.Mvc;

namespace IdentityApp.Controllers {

    public class StoreController : Controller {
        private ProductDbContext DbContext;

        public StoreController(ProductDbContext ctx) => DbContext = ctx;

        public IActionResult Index() => View(DbContext.Products);
    }
}
```

To provide the view for the Store controller's action method, create the IdentityApp/Views/Store folder and add a Razor View named Index.cshtml with the content shown in Listing 3-12. This view presents a table containing details of the Product objects in the database but does not provide any means to edit them.

Listing 3-12. The Contents of the Index.cshtml File in the Views/Store Folder

```
@model IQueryable<Product>

<h4 class="bg-primary text-white text-center p-2">MVC - Level 2 - Signed In Users</h4>

<div class="p-2">
    <table class="table table-sm table-striped table-bordered">
        <thead>
            <tr>
                <th>ID</th><th>Name</th><th>Category</th>
                <th class="text-right">Price</th>
            </tr>
        </thead>
        <tbody>
            @foreach (Product p in Model.OrderBy(p => p.Id)) {
                <tr>
                    <td>@p.Id</td>
                    <td>@p.Name</td>
                    <td>@p.Category</td>
                    <td class="text-right">$@p.Price.ToString("F2")</td>
                </tr>
            }
        </tbody>
    </table>
</div>
```

Add a class file named AdminController.cs to the Controllers folder and use it to define the controller shown in Listing 3-13. This controller will present the third level of content, which will be available only to administrators.

Listing 3-13. The Contents of the AdminController.cs File in the Controllers Folder

```
using IdentityApp.Models;
using Microsoft.AspNetCore.Mvc;

namespace IdentityApp.Controllers {

    public class AdminController : Controller {
        private ProductDbContext DbContext;

        public AdminController(ProductDbContext ctx) => DbContext = ctx;

        public IActionResult Index() => View(DbContext.Products);

        [HttpGet]
        public IActionResult Create() => View("Edit", new Product());

        [HttpGet]
        public IActionResult Edit(long id) {
            Product p = DbContext.Find<Product>(id);
            if (p != null) {
                return View("Edit", p);
            }
            return RedirectToAction(nameof(Index));
        }

        [HttpPost]
        public IActionResult Save(Product p) {
            DbContext.Update(p);
            DbContext.SaveChanges();
            return RedirectToAction(nameof(Index));
        }

        [HttpPost]
        public IActionResult Delete(long id) {
            Product p = DbContext.Find<Product>(id);
            if (p != null) {
                DbContext.Remove(p);
                DbContext.SaveChanges();
            }
            return RedirectToAction(nameof(Index));
        }
    }
}
```

Create the IdentityApp/Views/Admin folder and add to it a Razor View named Index.cshtml with the content shown in Listing 3-14.

Listing 3-14. The Contents of the Index.cshtml File in the Views/Admin Folder

```
@model IQueryable<Product>

<h4 class="bg-primary text-white text-center p-2">MVC Level 3 - Administrators</h4>

<div class="p-2">
    <table class="table table-sm table-striped table-bordered">
        <thead>
            <tr>
                <th>ID</th><th>Name</th><th>Category</th>
                <th class="text-right">Price</th><th></th>
            </tr>
        </thead>
        <tbody>
            @foreach (Product p in Model.OrderBy(p => p.Id)) {
                <tr>
                    <td>@p.Id</td>
                    <td>@p.Name</td>
                    <td>@p.Category</td>
                    <td class="text-right">$@p.Price.ToString("F2")</td>
                    <td class="text-center">
                        <form method="post">
                            <a class="btn btn-sm btn-warning" asp-action="edit"
                                asp-route-id="@p.Id">Edit</a>
                            <button class="btn btn-sm btn-danger"
                                asp-action="delete" asp-route-id="@p.Id">
                                    Delete
                            </button>
                        </form>
                    </td>
                </tr>
            }
        </tbody>
    </table>
</div>
<a class="btn btn-primary mx-2" asp-action="Create">Create</a>
```

This view presents a table that shows the `Product` details, along with buttons for creating, editing, and deleting data. To create the HTML that will be used to create and edit data, add a Razor View named `Edit.cshtml` to the `Views/Admin` folder with the contents shown in Listing 3-15.

Listing 3-15. The Contents of the Edit.cshtml File in the Views/Admin Folder

```
@model Product

<h4 class="bg-primary text-white text-center p-2">MVC Level 3 - Administrators</h4>

<form method="post" asp-action="save" class="p-2">
    <div class="form-group">
        <label>ID</label>
        <input class="form-control" readonly asp-for="Id" />
```

```
    </div>
    <div class="form-group">
        <label>Name</label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label>Category</label>
        <input class="form-control" asp-for="Category" />
    </div>
    <div class="form-group">
        <label>Price</label>
        <input class="form-control" type="number" asp-for="Price" />
    </div>
    <div class="text-center">
        <button type="submit" class="btn btn-primary">Save</button>
        <a class="btn btn-secondary" asp-action="Index">Cancel</a>
    </div>
</form>
```

To enable tag helpers and import the data model namespace and some useful ASP.NET Core Identity namespaces, add a Razor View Imports file named `_ViewImports.cshtml` file in the `Views` folder with the content shown in Listing 3-16.

Listing 3-16. The Contents of the _ViewImports.cshtml File in the Views Folder

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@using IdentityApp.Models
@using Microsoft.AspNetCore.Identity
@using System.Security.Claims
```

To automatically specify a layout for the views in the application, add a Razor View Start file named `_ViewStart.cshtml` to the `Views` folder with the content shown in Listing 3-17.

Listing 3-17. The Contents of the _ViewStart.cshtml File in the Views Folder

```
@{
    Layout = "_Layout";
}
```

Create the `IdentityApp/Views/Shared` folder and add to it a Razor Layout named `_Layout.cshtml` with the content shown in Listing 3-18. This file provides the HTML structure into which views (and Razor Pages) will render their content, including a link for the CSS stylesheet from the Bootstrap package.

Listing 3-18. The Contents of the _Layout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Identity App</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
```

```
<body>
    <partial name="_NavigationPartial" />
    @RenderBody()
</body>
</html>
```

The layout relies on a partial view to display content that will allow easy navigation between the different levels of content. Add a Razor View named `_NavigationPartial.cshtml` to the `Views/Shared` folder with the content shown in Listing 3-19.

Listing 3-19. The Contents of the _NavigationPartial.cshtml File in the Views/Shared Folder

```
<div class="text-center m-2">
    <a class="btn btn-secondary btn-sm" asp-controller="Home">Level 1</a>
    <a class="btn btn-secondary btn-sm" asp-controller="Store">Level 2</a>
    <a class="btn btn-secondary btn-sm" asp-controller="Admin">Level 3</a>
</div>
```

Creating Razor Pages

Create the `IdentityApp/Pages` folder and add to it a Razor Page named `Landing.cshtml` with the content shown in Listing 3-20. This page will present the first level of access, which is available to anyone.

Listing 3-20. The Contents of the Landing.cshtml File in the Pages Folder

```
@page "/pages"
@model IdentityApp.Pages.LandingModel

<h4 class="bg-info text-white text-center p-2">Pages - Level 1 - Anyone</h4>

<div class="text-center">
    <h6 class="p-2">
        The store contains @Model.DbContext.Products.Count() products.
    </h6>
</div>
```

To define the page model class, add the code shown in Listing 3-21 to the `Landing.cshtml.cs` file in the `Pages` folder. (You will have to create this file if you are using Visual Studio Code. I repeat this note throughout this book because even experienced readers become used to the way that Visual Studio creates files and don't understand why they are missing in Visual Studio Code.)

Listing 3-21. The Contents of the Landing.cshtml.cs File in the Pages Folder

```
using IdentityApp.Models;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace IdentityApp.Pages {
    public class LandingModel : PageModel {

        public LandingModel(ProductDbContext ctx) => DbContext = ctx;
```

```
        public ProductDbContext DbContext { get; set; }

    }
}
```

Add a Razor Page named Store.cshtml to the Pages folder with the content shown in Listing 3-22. This page will be available to signed-in users.

Listing 3-22. The Contents of the Store.cshtml File in the Pages Folder

```
@page "/pages/store"
@model IdentityApp.Pages.StoreModel

<h4 class="bg-info text-white text-center p-2">Pages - Level 2 - Signed In Users</h4>

<div class="p-2">
    <table class="table table-sm table-striped table-bordered">
        <thead>
            <tr>
                <th>ID</th><th>Name</th><th>Category</th>
                <th class="text-right">Price</th>
            </tr>
        </thead>
        <tbody>
            @foreach (Product p in Model.DbContext.Products.OrderBy(p => p.Id)) {
                <tr>
                    <td>@p.Id</td>
                    <td>@p.Name</td>
                    <td>@p.Category</td>
                    <td class="text-right">$@p.Price.ToString("F2")</td>
                </tr>
            }
        </tbody>
    </table>
</div>
```

To define the page model, add the code shown in Listing 3-23 to the Store.cshtml.cs file. (You will have to create this file if you are using Visual Studio Code.)

Listing 3-23. The Contents of the Store.cshtml.cs File in the Pages Folder

```
using IdentityApp.Models;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace IdentityApp.Pages {
    public class StoreModel : PageModel {
        public StoreModel(ProductDbContext ctx) => DbContext = ctx;

        public ProductDbContext DbContext { get; set; }
    }
}
```

Next, add a Razor Page named Admin.cshtml to the IdentityApp/Pages folder with the content shown in Listing 3-24. This page will be available only to administrators.

Listing 3-24. The Contents of the Admin.cshtml File in the Pages Folder

```
@page "/pages/admin"
@model IdentityApp.Pages.AdminModel

<h4 class="bg-info text-white text-center p-2">Pages Level 3 - Administrators</h4>

<div class="p-2">
    <table class="table table-sm table-striped table-bordered">
        <thead>
            <tr>
                <th>ID</th><th>Name</th><th>Category</th>
                <th class="text-right">Price</th><th></th>
            </tr>
        </thead>
        <tbody>
            @foreach (Product p in Model.DbContext.Products.OrderBy(p => p.Id)) {
                <tr>
                    <td>@p.Id</td>
                    <td>@p.Name</td>
                    <td>@p.Category</td>
                    <td class="text-right">$@p.Price.ToString("F2")</td>
                    <td class="text-center">
                        <form method="post">
                            <button class="btn btn-sm btn-danger"
                                asp-route-id="@p.Id">
                                    Delete
                            </button>
                            <a class="btn btn-sm btn-warning" asp-page="Edit"
                                asp-route-id="@p.Id">Edit</a>
                        </form>
                    </td>
                </tr>
            }
        </tbody>
    </table>
</div>
<a class="btn btn-info mx-2" asp-page="Edit">Create</a>
```

To create the page model class for the Admin page, add the code shown in Listing 3-25 to the Admin.cshtml.cs file. (You will have to create this file if you are using Visual Studio Code.)

Listing 3-25. The Contents of the Admin.cshtml.cs File in the Pages Folder

```
using IdentityApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace IdentityApp.Pages {
```

```
    public class AdminModel : PageModel {

        public AdminModel(ProductDbContext ctx) => DbContext = ctx;

        public ProductDbContext DbContext { get; set; }

        public IActionResult OnPost(long id) {
            Product p = DbContext.Find<Product>(id);
            if (p != null) {
                DbContext.Remove(p);
                DbContext.SaveChanges();
            }
            return Page();
        }
    }
}
```

Add a Razor Page named `Edit.cshtml` to the `Pages` folder with the content shown in Listing 3-26. This page will display the HTML form that will be used to create and edit `Product` objects.

Listing 3-26. The Contents of the Edit.cshtml File in the Pages Folder

```
@page "/pages/edit/{id:long?}"
@model IdentityApp.Pages.EditModel

<h4 class="bg-info text-white text-center p-2">Product Page</h4>

<form method="post" class="p-2">
    <div class="form-group">
        <label>ID</label>
        <input class="form-control" readonly asp-for="@Model.Product.Id" />
    </div>
    <div class="form-group">
        <label>Name</label>
        <input class="form-control" asp-for="@Model.Product.Name" />
    </div>
    <div class="form-group">
        <label>Category</label>
        <input class="form-control" asp-for="@Model.Product.Category" />
    </div>
    <div class="form-group">
        <label>Price</label>
        <input class="form-control" type="number" asp-for="@Model.Product.Price" />
    </div>
    <div class="text-center">
        <button type="submit" class="btn btn-secondary">Save</button>
        <a class="btn btn-secondary" asp-page="Admin">Cancel</a>
    </div>
</form>
```

Add the code shown in Listing 3-27 to the `Edit.cshtml.cs` file to define the page model class for the editor. (You will have to create this file if you are using Visual Studio Code.)

Listing 3-27. The Contents of the Edit.cshtml.cs File in the Pages Folder

```
using IdentityApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace IdentityApp.Pages {

    public class EditModel : PageModel {

        public EditModel(ProductDbContext ctx) => DbContext = ctx;

        public ProductDbContext DbContext { get; set; }
        public Product Product { get; set; }

        public void OnGet(long id) {
            Product = DbContext.Find<Product>(id) ?? new Product();
        }

        public IActionResult OnPost([Bind(Prefix = "Product")] Product p) {
            DbContext.Update(p);
            DbContext.SaveChanges();
            return RedirectToPage("Admin");
        }
    }
}
```

Add a Razor View Imports file named `_ViewImports.cshtml` to the `Pages` folder and add the content shown in Listing 3-28, which will enable tag helpers in Razor Pages and import some namespaces used in the views (and some that are useful for working with ASP.NET Core Identity).

Listing 3-28. The Contents of the _ViewImports.cshtml File in the Pages Folder

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@using Microsoft.AspNetCore.Mvc.RazorPages
@using Microsoft.AspNetCore.Identity
@using System.Security.Claims
@using IdentityApp.Pages
@using IdentityApp.Models
```

Add a Razor View Start file named `_ViewStart.cshtml` to the `Pages` folder with the content shown in Listing 3-29.

Listing 3-29. The Contents of the _ViewStart.cshtml in the Pages Folder

```
@{
    Layout = "_Layout";
}
```

Add a Razor Layout named `_NavigationPartial.cshtml` to the `Pages` folder with the content shown in Listing 3-30.

Listing 3-30. The Contents of the _NavigationPartial.cshtml File in the Pages Folder

```
<div class="text-center m-2">
    <a class="btn btn-secondary btn-sm" asp-page="Landing">Level 1</a>
    <a class="btn btn-secondary btn-sm" asp-page="Store">Level 2</a>
    <a class="btn btn-secondary btn-sm" asp-page="Admin">Level 3</a>
</div>
```

The Razor Pages share a layout with the MVC controllers, and only the contents of the partial view will be different, allowing easy navigation between pages.

Configure the Application

Listing 3-31 shows the changes required to the `Startup` class to set up Entity Framework Core, the MVC Framework, and Razor Pages.

Listing 3-31. Configuring the Application in the Startup.cs File in the IdentityApp Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using IdentityApp.Models;

namespace IdentityApp {

    public class Startup {

        public Startup(IConfiguration config) => Configuration = config;

        private IConfiguration Configuration { get; set; }

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

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            if (env.IsDevelopment()) {
                app.UseDeveloperExceptionPage();
            }
            app.UseStaticFiles();
            app.UseRouting();
```

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

You should already be familiar with the methods used in Listing 3-31, which provide access to the configuration data in the `appsettings.json` file and configure the services and middleware required for static content, Razor Pages, and the MVC Framework.

Creating the Database

To create Entity Framework Core migration for the product database, use a PowerShell command prompt to run the command shown in Listing 3-32 in the `IdentityApp` folder.

Listing 3-32. Creating the Migration and Database

```
dotnet ef migrations add Initial
```

Once the migration has been created, run the commands shown in Listing 3-33, which will remove the `IdentityAppData` database if it exists and then re-create it.

Listing 3-33. Deleting and Creating the Database

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

Running the Example Application

Using a PowerShell command prompt, run the command shown in Listing 3-34 to start the example application.

Listing 3-34. Running the Example Application

```
dotnet run
```

One ASP.NET Core has started, request `http://localhost:5000`. This request will be handled by the `Index` action of the `Home` controller, which presents the first level of content, available to anyone. The layouts defined earlier included buttons that allow easy movement between the controllers and the different levels of content they present. There are no access restrictions in place at the moment, which means that all the content is accessible to anyone, as shown in Figure 3-1.

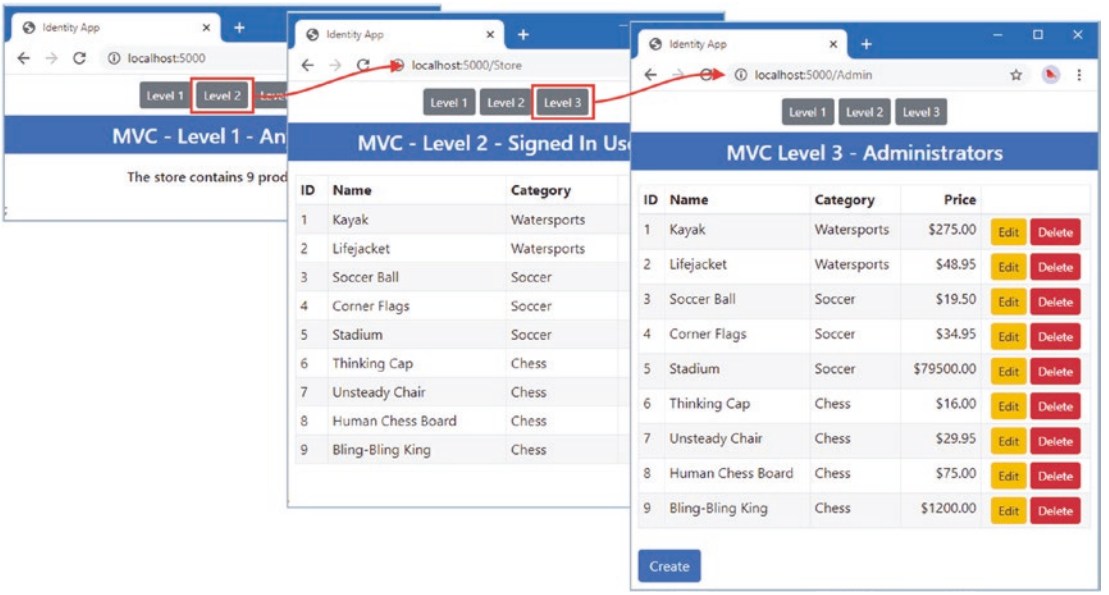

Figure 3-1. *The content produced by the MVC Framework*

Request `http://localhost:5000/pages` to see the same functionality implemented using Razor Pages. I have used a different color scheme and clear labels to make it obvious that Razor Pages have produced the content, as shown in Figure 3-2.

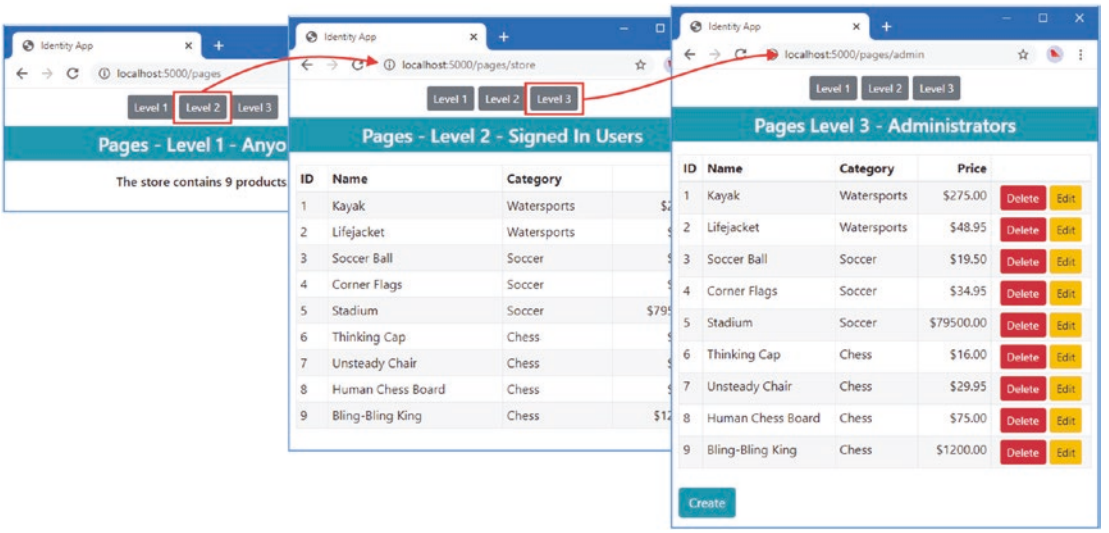

Figure 3-2. *The content produced by Razor Pages*

Enabling HTTPS Connections

There is one more set of changes required to prepare the example application. ASP.NET Core relies on cookies and HTTP request headers to authenticate requests, which presents the risk that an eavesdropper might intercept an HTTP request and use the cookie or header it contains to send a request that will appear as though it has been sent by the user.

When using an ASP.NET Core application that requires authentication, it is important to ensure that all requests are sent using HTTPS, which encrypts the messages between the browser and ASP.NET Core to guard against eavesdropping.

HTTPS VS. SSL VS. TLS

HTTPS is the combination of HTTP and the transport layer security (TLS) or secure sockets layer (SSL). TLS has replaced the obsolete SSL protocol, but the term SSL has become synonymous with secure networking and is often used even when TLS is responsible for securing a connection. If you are interested in security and cryptography, then the details of HTTPS are worth exploring, and `https://en.wikipedia.org/wiki/HTTPS` is a good place to start.

Generating a Test Certificate

An important HTTPS feature is the use of a certificate that allows web browsers to confirm they are communicating with the right web server and not an impersonator. To make application development simpler, the .NET SDK includes a test certificate that can be used for HTTPS. Use a PowerShell command prompt to run the commands shown in Listing 3-35 in the `IdentityApp` folder to generate a new test certificate and add it to the set of certificates that Windows will trust.

Listing 3-35. Generating and Trusting a New Certificate

```
dotnet dev-certs https --clean
dotnet dev-certs https --trust
```

Click Yes to the prompts to delete the existing certificate it has already been trusted, and click Yes to trust the new certificate, as shown in Figure 3-3.

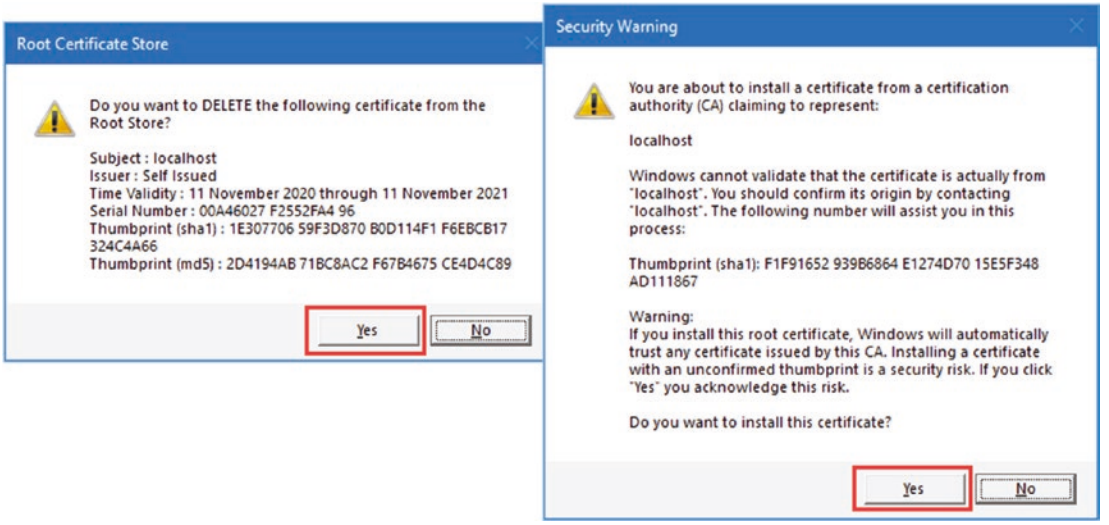

Figure 3-3. *Generating and trusting a test certificate for HTTPS*

USING A CERTIFICATE FOR DEPLOYMENT

The .NET test certificate can be used only during development, and you will need to use a real certificate when you are ready to deploy a project. If you don't have a certificate, I recommend `https://letsencrypt.org/`, which is a nonprofit organization that issues certificates for free. As part of the registration process, you will need to prove you control the domain for which you require a certificate, but Let's Encrypt provides tools for this process.

Once you have a certificate—regardless of how you obtain one—you can find instructions for configuring ASP.NET Core at `https://docs.microsoft.com/en-us/aspnet/core/security/authentication/certauth?view=aspnetcore-5.0`.

Enabling HTTPS

To enable HTTPS, make the changes shown in Listing 3-36 to the `launchSettings.json` file in the `Properties` folder.

Listing 3-36. Enabling HTTPS 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"
      }
    },
    "IdentityApp": {
      "commandName": "Project",
      "dotnetRunMessages": "true",
      "launchBrowser": true,
      "applicationUrl": "http://localhost:5000;https://localhost:44350",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

I have chosen port 44350 for the example application, largely because IIS supports HTTPS only between ports 44300 and 44399.

I use ports 5000 (for HTTP) and 44350 (for HTTPS) during development because it avoids operating system restrictions on using low-numbered ports. Deployed applications will typically use port 80 for HTTP and 443 for HTTPS, but this will depend on your hosting environment. You may need to use different ports or find that HTTPS is handled for you as part of a shared infrastructure, which is often the case when deploying to a corporate data center. If you are unsure, check with your administrators or consult the documentation for your chosen deployment platform.

Enabling HTTPS Redirection

ASP.NET Core provides a feature that will redirect HTTP requests to the HTTPS port supported by the application. To enable HTTPS redirection, add the statement shown in Listing 3-37 to the `Startup` class.

Listing 3-37. Enabling HTTPS Redirection in the Startup.cs File in the IdentityApp Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using IdentityApp.Models;

namespace IdentityApp {

    public class Startup {

        public Startup(IConfiguration config) => Configuration = config;

        private IConfiguration Configuration { get; set; }
```

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

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

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            if (env.IsDevelopment()) {
                app.UseDeveloperExceptionPage();
            }

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

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

The `AddHttpsRedirection` method is used to configure the HTTPS redirection. For the example application, I need to specify that the redirection will be to port 44350, overriding the default port 443. The `UseHttpRedirection` method applies an ASP.NET Core middleware component that responds with a redirection when HTTP requests are received, using the settings specified by the `AddHttpsRedirection` method.

Restart ASP.NET Core and request `http://localhost:5000`. The example application will respond with a redirection, causing the browser to request `https://localhost:44350`, as shown in Figure 3-4.

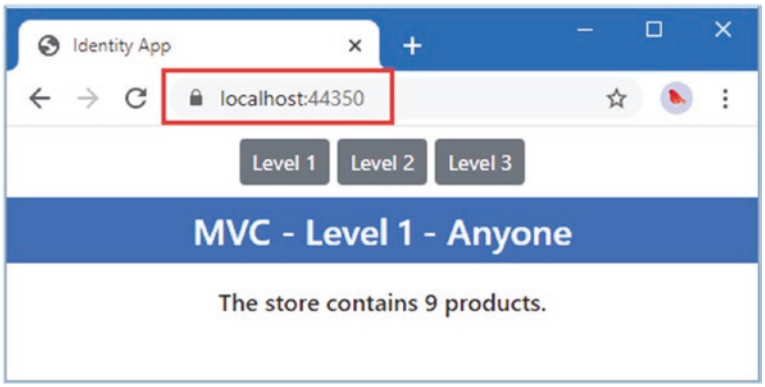

Figure 3-4. *Enabling HTTPS in the example application*

Restricting Access with an Authorization Policy

The final step in this chapter is to apply access restrictions so that ASP.NET Core will only allow users who meet the authorization policy access to protected actions or pages. The changes in this section break the application. ASP.NET Core provides a complete set of features for enforcing authentication and authorization, which I describe in Part 2. The following changes tell ASP.NET Core to restrict access but do not provide the features required to allow requests to be authenticated or to allow users to sign into the application. I address this in later chapters by installing and configuring ASP.NET Core Identity, but, until then, the result is that requests for restricted content will produce an exception.

Applying the Level 2 Authorization Policy

The `Authorize` attribute is used to restrict access, and Listing 3-38 applies the attribute to the `Store` controller.

Listing 3-38. Restricting Access in the StoreController.cs File in the Controllers Folder

```
using IdentityApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Authorization;

namespace IdentityApp.Controllers {

    [Authorize]
    public class StoreController : Controller {
        private ProductDbContext DbContext;

        public StoreController(ProductDbContext ctx) => DbContext = ctx;

        public IActionResult Index() => View(DbContext.Products);
    }
}
```

When the attribute is applied without any arguments, the effect is to restrict access to any signed-in user. I applied the attribute to the class, which applies this authorization policy to all of the action methods defined by the controller. In Listing 3-39, I have applied the attribute to the page model class of the `Store` Razor Page, which is the counterpart to the `Store` controller.

Listing 3-39. Restricting Access in the Store.cshtml.cs File in the Pages Folder

```
using IdentityApp.Models;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Authorization;

namespace IdentityApp.Pages {

    [Authorize]
    public class StoreModel : PageModel {
        public StoreModel(ProductDbContext ctx) => DbContext = ctx;

        public ProductDbContext DbContext { get; set; }
    }
}
```

The attribute can also be applied to Razor Pages with the `@attribute` expression, which I use for convenience in later examples, although the effect is the same.

Applying the Level 3 Authorization Policy

The `Authorize` attribute can be used to define more specific access restrictions. The most common approach is to restrict access to users who have been assigned to a specific *role*. In Listing 3-40, I have applied the `Authorize` attribute to the `Admin` controller using the `Roles` argument.

Listing 3-40. Restricting Access in the AdminController.cs File in the Controllers Folder

```
using IdentityApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Authorization;

namespace IdentityApp.Controllers {

    [Authorize(Roles = "Admin")]
    public class AdminController : Controller {
        private ProductDbContext DbContext;

        public AdminController(ProductDbContext ctx) => DbContext = ctx;

        public IActionResult Index() => View(DbContext.Products);

        [HttpGet]
        public IActionResult Create() => View("Edit", new Product());

        [HttpGet]
        public IActionResult Edit(long id) {
            Product p = DbContext.Find<Product>(id);
            if (p != null) {
                return View("Edit", p);
            }
            return RedirectToAction(nameof(Index));
        }

        [HttpPost]
        public IActionResult Save(Product p) {
            DbContext.Update(p);
            DbContext.SaveChanges();
            return RedirectToAction(nameof(Index));
        }

        [HttpPost]
        public IActionResult Delete(long id) {
            Product p = DbContext.Find<Product>(id);
            if (p != null) {
                DbContext.Remove(p);
                DbContext.SaveChanges();
            }
```

```
            return RedirectToAction(nameof(Index));
        }
    }
}
```

Listing 3-41 applies the same policy to the Admin page model class, which is the Level 3 Razor Page and the counterpart to the controller in Listing 3-40.

Listing 3-41. Restricting Access in the Admin.cshtml.cs File in the Pages Folder

```
using IdentityApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Authorization;

namespace IdentityApp.Pages {

    [Authorize(Roles = "Admin")]
    public class AdminModel : PageModel {

        public AdminModel(ProductDbContext ctx) => DbContext = ctx;

        public ProductDbContext DbContext { get; set; }

        public IActionResult OnPost(long id) {
            Product p = DbContext.Find<Product>(id);
            if (p != null) {
                DbContext.Remove(p);
                DbContext.SaveChanges();
            }
            return Page();
        }
    }
}
```

I need to apply the same restriction to the page model class for the Edit page, as shown in Listing 3-42, which handles editing on behalf of the Admin page.

Listing 3-42. Restricting Access in the Edit.cshtml.cs File in the Pages Folder

```
using IdentityApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Authorization;

namespace IdentityApp.Pages {

    [Authorize(Roles = "Admin")]
    public class EditModel : PageModel {

        public EditModel(ProductDbContext ctx) => DbContext = ctx;

        public ProductDbContext DbContext { get; set; }
```

```
        public Product Product { get; set; }

        public void OnGet(long id) {
            Product = DbContext.Find<Product>(id) ?? new Product();
        }

        public IActionResult OnPost([Bind(Prefix = "Product")] Product p) {
            DbContext.Update(p);
            DbContext.SaveChanges();
            return RedirectToPage("Admin");
        }
    }
}
```

Configuring the Application

The remaining step is to enable the ASP.NET Core features that handle authorization and authentication, as shown in Listing 3-43. As I explain in Part 2, these are the features with which ASP.NET Core Identity integrates but are provided by ASP.NET Core.

Listing 3-43. Enabling Features in the Startup.cs File in the IdentityApp Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using IdentityApp.Models;

namespace IdentityApp {

    public class Startup {

        public Startup(IConfiguration config) => Configuration = config;

        private IConfiguration Configuration { get; set; }

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

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

```
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            if (env.IsDevelopment()) {
                app.UseDeveloperExceptionPage();
            }

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

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

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

The `UseAuthentication` and `UseAuthorization` methods set up, as their names suggest, the ASP.NET Core authentication and authorization features. You don't need to understand how these features work to use ASP.NET Core Identity, but you will find full details in Part 2.

Restart ASP.NET Core and request `https://localhost:44350`. This request will be handled by the `Home` controller, to which no authorization restrictions have been applied and which will return a normal response, as shown in Figure 3-5. Request `https://localhost:44350/store`, and you will receive an error, also shown in Figure 3-5. This request is handled by the `Store` controller, to which the `Authorize` attribute has been applied. ASP.NET Core tries to establish the identity of the user who has sent the request but cannot do so because the required services are missing, causing an exception.

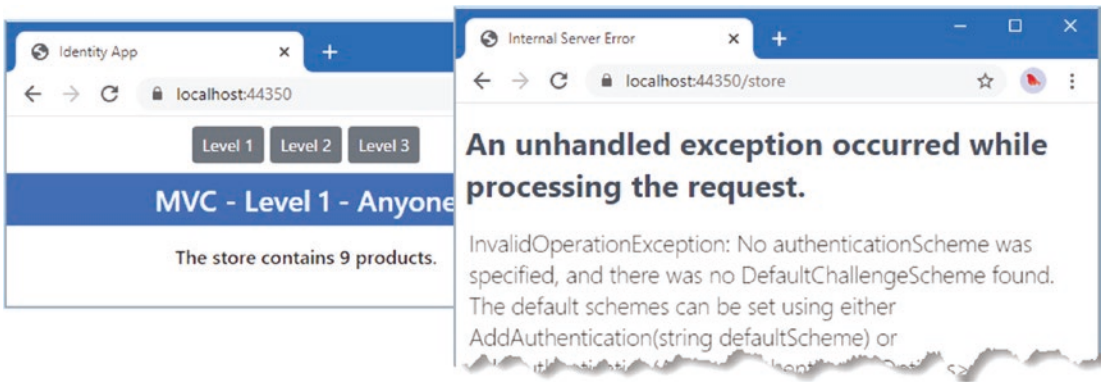

Figure 3-5. *The effect of applying authorization restrictions*

Summary

In this chapter, I created the example application that I will use throughout this part of the book. The application is simple but defines three levels of access control, which I use to explain how ASP.NET Core Identity works and how it integrates into the ASP.NET Core platform. In the next chapter, I show you how to install Identity and use the built-in Identity UI package.