The argument for the attribute specifies the format that will be used for the result from the action, and more than one type can be specified. The Produces attribute restricts the types that the MVC Framework will consider when processing an Accept header. To see the effect of the Produces attribute, use a PowerShell prompt to run the command shown in Listing 20-20.

Listing 20-20. Requesting Data

```
Invoke-WebRequest http://localhost:5000/api/content/object -Headers @{Accept="application/xml,application/
json;q=0.8"} | select @{n='Content-Type';e={ $_.Headers."Content-Type" }}, Content
```

The Accept header tells the MVC Framework that the client prefers XML data but will accept JSON. The Produces attribute means that XML data isn't available as the data format for the GetObject action method and so the JSON serializer is selected, which produces the following response:

```
Content-Type                     Content
------------                     -------
application/json; charset=utf-8 {"name":"Kayak","price":275.00,
                                  "categoryId":1,"supplierId":1}
```

Requesting a Format in the URL

The Accept header isn't always under the control of the programmer who is writing the client. In such situations, it can be helpful to allow the data format for the response to be requested using the URL. This feature is enabled by decorating an action method with the FormatFilter attribute and ensuring there is a format segment variable in the action method's route, as shown in Listing 20-21.

Listing 20-21. Enabling Formatting in the ContentController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;
using WebApp.Models;

namespace WebApp.Controllers {

    [ApiController]
    [Route("/api/[controller]")]
    public class ContentController : ControllerBase {
        private DataContext context;

        public ContentController(DataContext dataContext) {
            context = dataContext;
        }

        [HttpGet("string")]
        public string GetString() => "This is a string response";

        [HttpGet("object/{format?}")]
        [FormatFilter]
        [Produces("application/json", "application/xml")]
        public async Task<ProductBindingTarget> GetObject() {
            Product p = await context.Products.FirstAsync();
            return new ProductBindingTarget() {
                Name = p.Name, Price = p.Price, CategoryId = p.CategoryId,
                SupplierId = p.SupplierId
            };
        }
    }
}
```

The FormatFilter attribute is an example of a filter, which is an attribute that can modify requests and responses, as described in Chapter 30. This filter gets the value of the format segment variable from the route that matched the request and uses it to override the Accept header sent by the client. I have also expanded the range of types specified by the Produces attribute so that the action method can return both JSON and XML responses.

Each data format supported by the application has a shorthand: xml for XML data and json for JSON data. When the action method is targeted by a URL that contains one of these shorthand names, the Accept header is ignored, and the specified format is used. To see the effect, restart ASP.NET Core and use the browser to request http://localhost:5000/api/content/object/json and http://localhost:5000/api/content/object/xml, which produce the responses shown in Figure 20-6.

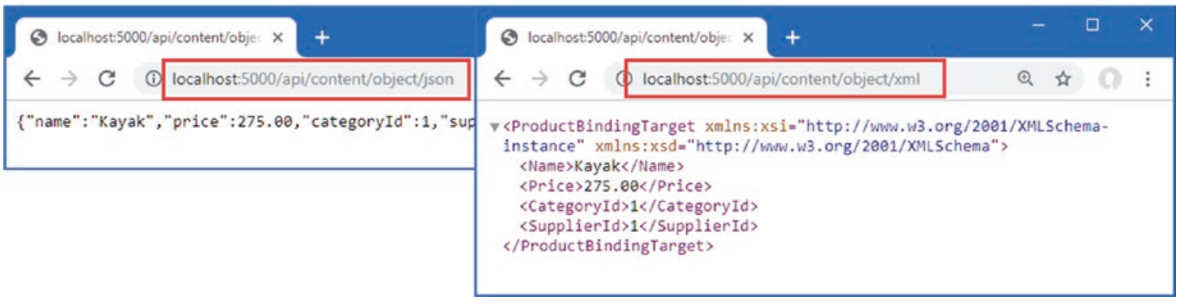

Figure 20-6. *Requesting data formats in the URL*

Restricting the Formats Received by an Action Method

Most content formatting decisions focus on the data formats the ASP.NET Core application sends to the client, but the same serializers that deal with results are used to deserialize the data sent by clients in request bodies. The deserialization process happens automatically, and most applications will be happy to accept data in all the formats they are configured to send. The example application is configured to send JSON and XML data, which means that clients can send JSON and XML data in requests.

The Consumes attribute can be applied to action methods to restrict the data types it will handle, as shown in Listing 20-22.

Listing 20-22. Adding Action Methods in the ContentController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;
using WebApp.Models;

namespace WebApp.Controllers {

    [ApiController]
    [Route("/api/[controller]")]
    public class ContentController : ControllerBase {
        private DataContext context;

        public ContentController(DataContext dataContext) {
            context = dataContext;
        }

        [HttpGet("string")]
        public string GetString() => "This is a string response";

        [HttpGet("object/{format?}")]
        [FormatFilter]
        [Produces("application/json", "application/xml")]
```

```
        public async Task<ProductBindingTarget> GetObject() {
            Product p = await context.Products.FirstAsync();
            return new ProductBindingTarget() {
                Name = p.Name, Price = p.Price, CategoryId = p.CategoryId,
                SupplierId = p.SupplierId
            };
        }

        [HttpPost]
        [Consumes("application/json")]
        public string SaveProductJson(ProductBindingTarget product) {
            return $"JSON: {product.Name}";
        }

        [HttpPost]
        [Consumes("application/xml")]
        public string SaveProductXml(ProductBindingTarget product) {
            return $"XML: {product.Name}";
        }
    }
}
```

The new action methods are decorated with the `Consumes` attribute, restricting the data types that each can handle. The combination of attributes means that HTTP POST attributes whose `Content-Type` header is `application/json` will be handled by the `SaveProductJson` action method. HTTP POST requests whose `Content-Type` header is `application/xml` will be handled by the `SaveProductXml` action method. Restart ASP.NET Core and use a PowerShell command prompt to run the command shown in Listing 20-23 to send JSON data to the example application.

Listing 20-23. Sending JSON Data

```
Invoke-RestMethod http://localhost:5000/api/content -Method POST -Body  (@{ Name="Swimming Goggles";
Price=12.75; CategoryId=1; SupplierId=1} | ConvertTo-Json) -ContentType "application/json"
```

The request is automatically routed to the correct action method, which produces the following response:

```
JSON: Swimming Goggles
```

Run the command shown in Listing 20-24 to send XML data to the example application.

Listing 20-24. Sending XML Data

```
Invoke-RestMethod http://localhost:5000/api/content -Method POST -Body  "<ProductBindingTarget><Name>Kayak
</Name><Price>275.00</Price><CategoryId>1</CategoryId><SupplierId>1</SupplierId></ProductBindingTarget>"
-ContentType "application/xml"
```

The request is routed to the `SaveProductXml` action method and produces the following response:

```
XML: Kayak
```

The MVC Framework will send a `415 - Unsupported Media Type` response if a request is sent with a `Content-Type` header that doesn't match the data types that the application supports.

Documenting and Exploring Web Services

When you are responsible for developing both the web service and its client, the purpose of each action and its results are obvious and are usually written at the same time. If you are responsible for a web service that is consumed by third-party developers, then you may need to provide documentation that describes how the web service works. The OpenAPI specification, which is also known as Swagger, describes web services in a way that can be understood by other programmers and consumed programmatically. In this section, I demonstrate how to use OpenAPI to describe a web service and show you how to fine-tune that description.

Resolving Action Conflicts

The OpenAPI discovery process requires a unique combination of the HTTP method and URL pattern for each action method. The process doesn't support the `Consumes` attribute, so a change is required to the `ContentController` to remove the separate actions for receiving XML and JSON data, as shown in Listing 20-25.

Listing 20-25. Removing an Action in the ContentController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;
using WebApp.Models;

namespace WebApp.Controllers {

    [ApiController]
    [Route("/api/[controller]")]
    public class ContentController : ControllerBase {
        private DataContext context;

        public ContentController(DataContext dataContext) {
            context = dataContext;
        }

        [HttpGet("string")]
        public string GetString() => "This is a string response";

        [HttpGet("object/{format?}")]
        [FormatFilter]
        [Produces("application/json", "application/xml")]
        public async Task<ProductBindingTarget> GetObject() {
            Product p = await context.Products.FirstAsync();
            return new ProductBindingTarget() {
                Name = p.Name, Price = p.Price, CategoryId = p.CategoryId,
                SupplierId = p.SupplierId
            };
        }

        [HttpPost]
        [Consumes("application/json")]
        public string SaveProductJson(ProductBindingTarget product) {
            return $"JSON: {product.Name}";
        }
```

```
        //[HttpPost]
        //[Consumes("application/xml")]
        //public string SaveProductXml(ProductBindingTarget product) {
        //    return $"XML: {product.Name}";
        //}
    }
}
```

Commenting out one of the action methods ensures that each remaining action has a unique combination of HTTP method and URL.

Installing and Configuring the Swashbuckle Package

The Swashbuckle package is the most popular ASP.NET Core implementation of the OpenAPI specification and will automatically generate a description for the web services in an ASP.NET Core application. The package also includes tools that consume that description to allow the web service to be inspected and tested.

Open a new PowerShell command prompt, navigate to the folder that contains the `WebApp.csproj` file, and run the commands shown in Listing 20-26 to install the NuGet package. If you are using Visual Studio, you can select Project ➤ Manage Nuget Packages and install the package through the Visual Studio package user interface.

Listing 20-26. Adding a Package to the Project

```
dotnet add package Swashbuckle.AspNetCore --version 5.0.0-rc2
```

Add the statements shown in Listing 20-27 to the `Startup` class to add the services and middleware provided by the Swashbuckle package.

Listing 20-27. Configuring Swashbuckle in the Startup.cs File in the WebApp Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
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 WebApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.OpenApi.Models;

namespace WebApp {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

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

```
        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<DataContext>(opts => {
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:ProductConnection"]);
                opts.EnableSensitiveDataLogging(true);
            });

            services.AddControllers()
                .AddNewtonsoftJson().AddXmlSerializerFormatters();

            services.Configure<MvcNewtonsoftJsonOptions>(opts => {
                opts.SerializerSettings.NullValueHandling
                    = Newtonsoft.Json.NullValueHandling.Ignore;
            });

            services.Configure<MvcOptions>(opts => {
                opts.RespectBrowserAcceptHeader = true;
                opts.ReturnHttpNotAcceptable = true;
            });

            services.AddSwaggerGen(options => {
                options.SwaggerDoc("v1",
                    new OpenApiInfo { Title = "WebApp", Version = "v1" });
            });
        }

        public void Configure(IApplicationBuilder app, DataContext context) {
            app.UseDeveloperExceptionPage();
            app.UseRouting();
            app.UseMiddleware<TestMiddleware>();
            app.UseEndpoints(endpoints => {
                endpoints.MapGet("/", async context => {
                    await context.Response.WriteAsync("Hello World!");
                });;
                endpoints.MapControllers();
            });
            app.UseSwagger();
            app.UseSwaggerUI(options => {
                options.SwaggerEndpoint("/swagger/v1/swagger.json", "WebApp");
            });
            SeedData.SeedDatabase(context);
        }
    }
}
```

There are two features set up by the statements in Listing 20-27. The feature generates an OpenAPI description of the web services that the application contains. You can see the description by restarting ASP.NET Core and using the browser to request the URL `http://localhost:5000/swagger/v1/swagger.json`, which produces the response shown in Figure 20-7. The OpenAPI format is verbose, but you can see each URL that the web service controllers support, along with details of the data each expects to receive and the range of responses that it will generate.

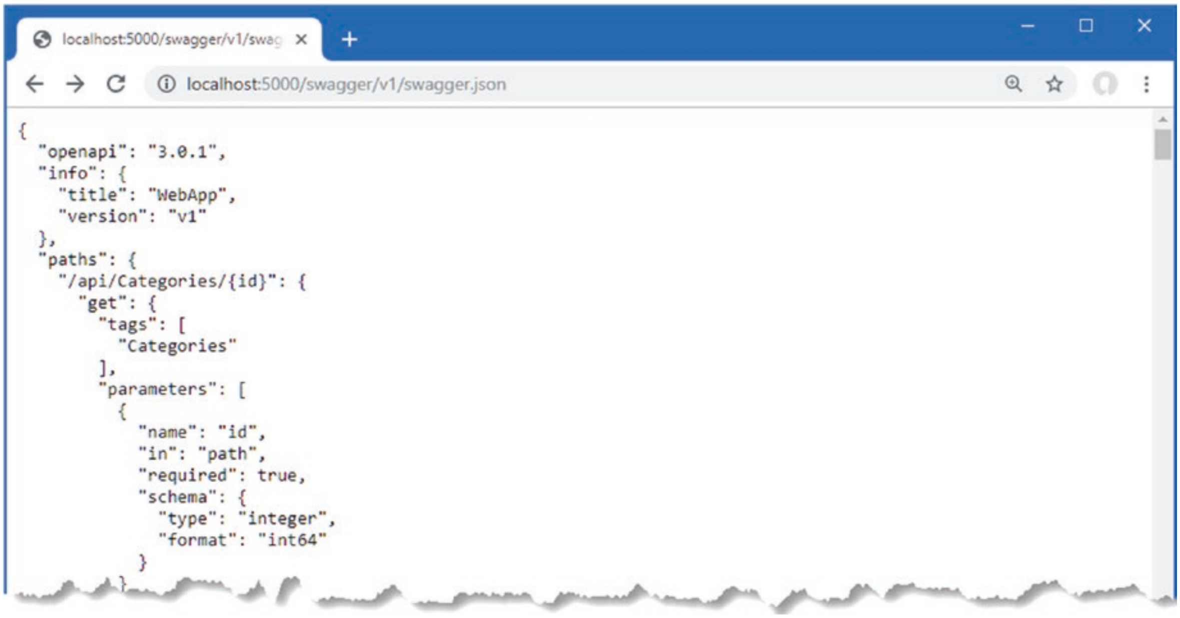

Figure 20-7. *The OpenAPI description of the web service*

The second feature is a UI that consumes the OpenAPI description of the web service and presents the information in a more easily understood way, along with support for testing each action. Use the browser to request `http://localhost:5000/swagger`, and you will see the interface shown in Figure 20-8. You can expand each action to see details, including the data that is expected in the request and the different responses that the client can expect.

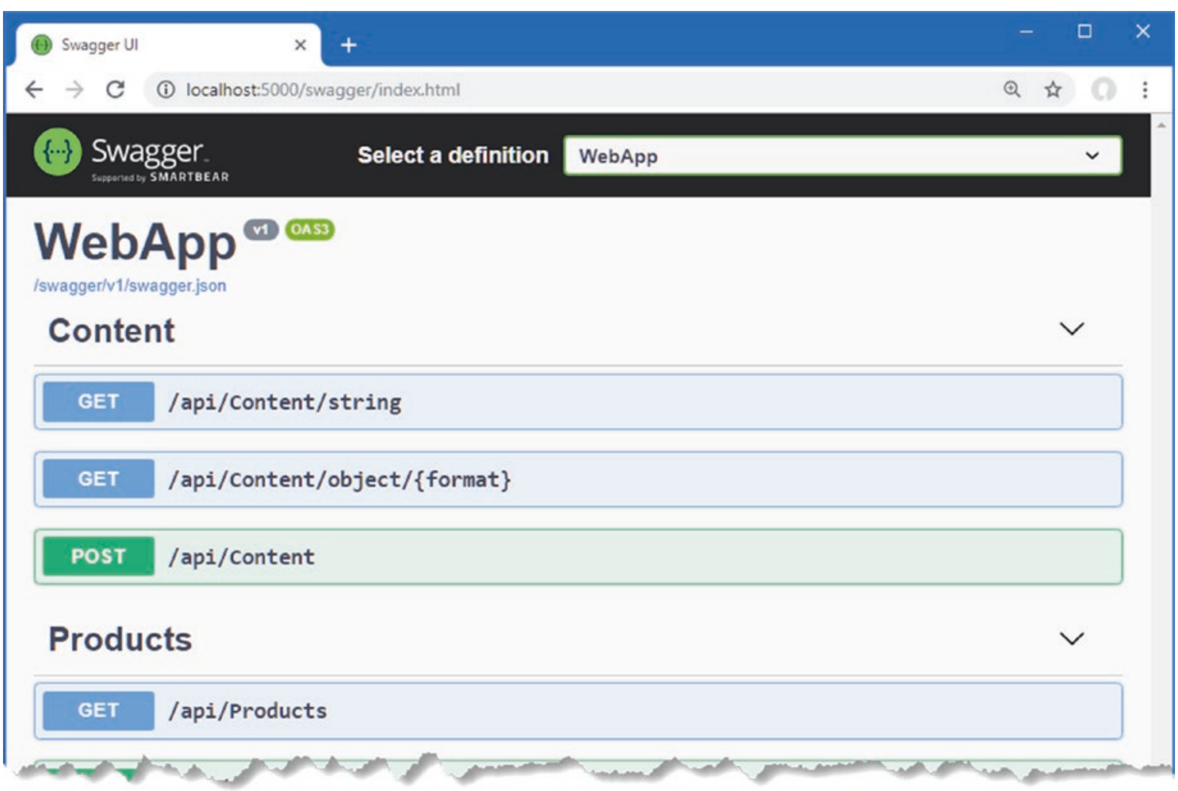

Figure 20-8. *The OpenAPI explorer interface*

Fine-Tuning the API Description

Relying on the API discovery process can produce a result that doesn't truly capture the web service. You can see this by examining the entry in the Products section that describes GET requests matched by the `/api/Product/{id}` URL pattern. Expand this item and examine the response section, and you will see there is only one status code response that will be returned, as shown in Figure 20-9.

Figure 20-9. *The data formats listed in the OpenAPI web service description*

The API discovery process makes assumptions about the responses produced by an action method and doesn't always reflect what can really happen. In this case, the `GetProduct` action method in the `ProductController` class can return another response that the discovery process hasn't detected.

```
...
[HttpGet("{id}")]
public async Task<IActionResult> GetProduct(long id) {
    Product p = await context.Products.FindAsync(id);
    if (p == null) {
        return NotFound();
    }
    return Ok(new {
        ProductId = p.ProductId, Name = p.Name,
        Price = p.Price, CategoryId = p.CategoryId,
        SupplierId = p.SupplierId
    });
}
...
```

If a third-party developer attempts to implement a client for the web service using the OpenAPI data, they won't be expecting the 404 - Not Found response that the action sends when it can't find an object in the database.

Running the API Analyzer

ASP.NET Core includes an analyzer that inspects web service controllers and highlights problems like the one described in the previous section. To enable the analyzer, add the elements shown in Listing 20-28 to the `WebApp.cspoj` file. (If you are using Visual Studio, right-click the WebApp project item in the Solution Explorer and select Edit Project File from the popup menu.)

Listing 20-28. Enabling the Analyzer in the WebApp.csproj File in the WebApp Folder

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Mvc.NewtonsoftJson"
        Version="3.1.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="3.1.1">
      <IncludeAssets>runtime; build; native; contentfiles; analyzers;
           buildtransitive</IncludeAssets>
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer"
         Version="3.1.1" />
    <PackageReference Include="Swashbuckle.AspNetCore" Version="5.0.0-rc2" />
  </ItemGroup>

  <PropertyGroup>
    <IncludeOpenAPIAnalyzers>true</IncludeOpenAPIAnalyzers>
  </PropertyGroup>

</Project>
```

If you are using Visual Studio, you will see any problems detected by the API analyzer shown in the controller class file, as shown in Figure 20-10.

Figure 20-10. *A problem detected by the API analyzer*

If you are using Visual Studio Code, you will see warning messages when the project is compiled, either using the `dotnet build` command or when it is executed using the `dotnet run` command. When the project is compiled, you will see this message that describes the issue in the `ProductController` class:

```
Controllers\ProductsController.cs(28,9): warning API1000: Action method returns undeclared status code '404'.
[C:\WebApp\WebApp.csproj]
    1 Warning(s)
    0 Error(s)
```

Declaring the Action Method Result Type

To fix the problem detected by the analyzer, the ProducesResponseType attribute can be used to declare each of the response types that the action method can produce, as shown in Listing 20-29.

Listing 20-29. Declaring the Result in the ProductsController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;
using System.Collections.Generic;
using Microsoft.Extensions.Logging;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;

namespace WebApp.Controllers {

    [ApiController]
    [Route("api/[controller]")]
    public class ProductsController : ControllerBase {
        private DataContext context;

        public ProductsController(DataContext ctx) {
            context = ctx;
        }

        [HttpGet]
        public IAsyncEnumerable<Product> GetProducts() {
            return context.Products;
        }

        [HttpGet("{id}")]
        [ProducesResponseType(StatusCodes.Status200OK)]
        [ProducesResponseType(StatusCodes.Status404NotFound)]
        public async Task<IActionResult> GetProduct(long id) {
            Product p = await context.Products.FindAsync(id);
            if (p == null) {
                return NotFound();
            }
            return Ok(new {
                ProductId = p.ProductId, Name = p.Name,
                Price = p.Price, CategoryId = p.CategoryId,
                SupplierId = p.SupplierId
            });
        }

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

Restart ASP.NET Core and use a browser to request http://localhost:5000/swagger, and you will see the description for the action method has been updated to reflect the 404 response, as shown in Figure 20-11.

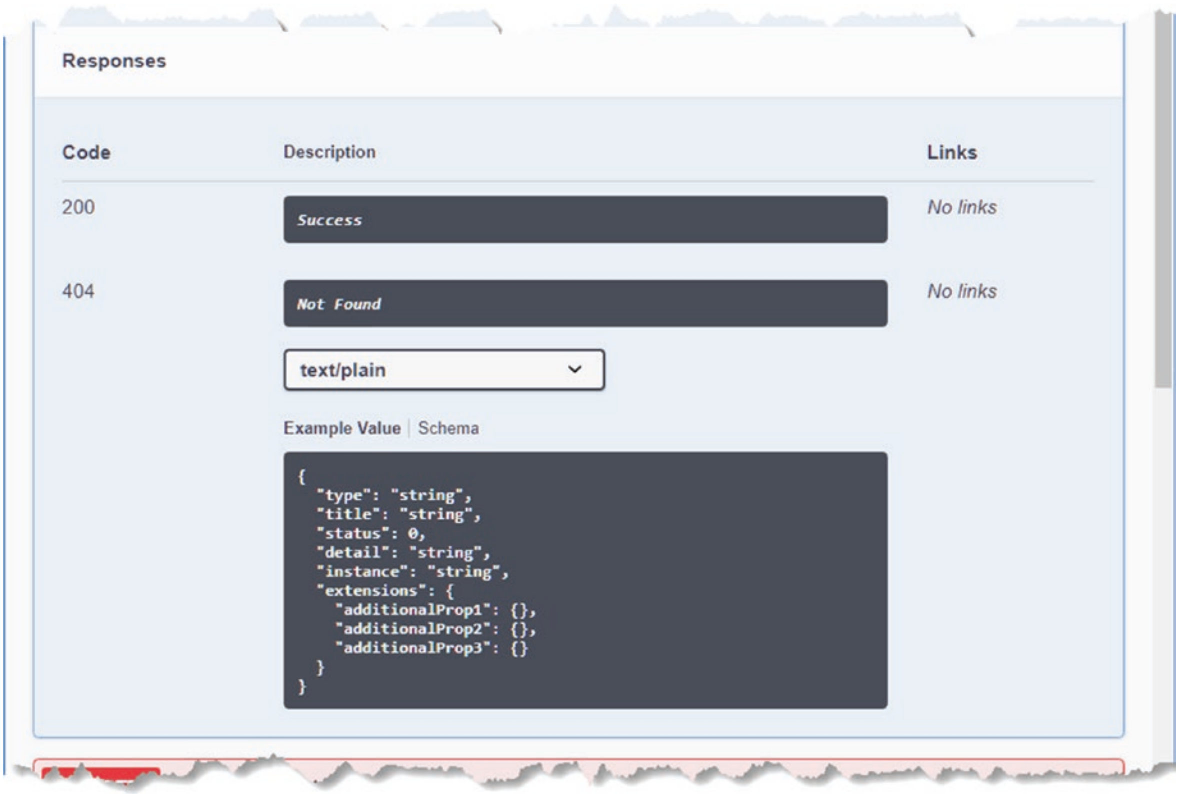

Figure 20-11. *Reflecting all the status codes produced by an action method*

Summary

In this chapter, I described some of the advanced features available for creating web services. I explained how to deal with related data in Entity Framework Core queries, how to support the HTTP PATCH method for handling selective updates, how content negotiation works, and how to use OpenAPI to describe the web services you create. In the next chapter, I describe how controllers can generate HTML responses.

CHAPTER 21

Using Controllers with Views, Part I

In this chapter, I introduce the *Razor view engine*, which is responsible for generating HTML responses that can be displayed directly to the user (as opposed to the JSON and XML responses, which are typically consumed by other applications). *Views* are files that contain C# expressions and HTML fragments that are processed by the view engine to generate HTML responses. I show how views work, explain how they are used in action methods, and describe the different types of C# expression they contain. In Chapter 22, I describe some of the other features that views support. Table 21-1 puts Razor views in context.

Table 21-1. *Putting Razor Views in Context*

Question	Answer
What are they?	Views are files that contain a mix of static HTML content and C# expressions.
Why are they useful?	Views are used to create HTML responses for HTTP requests. The C# expressions are evaluated and combined with the HTML content to create a response.
How are they used?	The `View` method defined by the `Controller` class creates an action response that uses a view.
Are there any pitfalls or limitations?	It can take a little time to get used to the syntax of view files and the way they combine code and content.
Are there any alternatives?	There are a number of third-party view engines that can be used in ASP.NET Core MVC, but their use is limited.

Table 21-2 summarizes the chapter.

Table 21-2. *Chapter Summary*

Problem	Solution	Listing
Enabling views	Use the `AddControllersWithViews` and `MapControllerRoute` methods to set up the required services and endpoints	1–5
Returning an HTML response from a controller action method	Use the `View` method to create a `ViewResult`	6
Creating dynamic HTML content	Create a Razor view that uses expressions for dynamic content	7–9, 20, 21
Selecting a view by name	Provide the view name as an argument to the `View` method	10, 11
Creating a view that can be used by multiple controllers	Create a shared view	12–14
Specifying a model type for a view	Use an `@model` expression	15–19
Generating content selectively	Use `@if`, `@switch` or `@foreach` expressions	22–26
Including C# code in a view	Use a code block	27

A. Freeman, *Pro ASP.NET Core 3*, https://doi.org/10.1007/978-1-4842-5440-0_21

Preparing for This Chapter

This chapter uses the WebApp project from Chapter 20. To prepare for this chapter, open a new PowerShell command prompt and run the command shown in Listing 21-1 in the WebApp folder to install a new package. If you are using Visual Studio, you can install the package by selecting Project ➤ Manage NuGet Packages.

Listing 21-1. Adding a Package to the Example Project

```
dotnet add package Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation --version 3.1.1
```

Next, replace the contents of the Startup class with the statements shown in Listing 21-2, which remove some of the services and middleware used in earlier chapters.

■ **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-3. See Chapter 1 for how to get help if you have problems running the examples.

Listing 21-2. Replacing the Contents of the Startup.cs File in the WebApp Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using WebApp.Models;

namespace WebApp {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        public IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<DataContext>(opts => {
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:ProductConnection"]);
                opts.EnableSensitiveDataLogging(true);
            });
            services.AddControllers();
        }

        public void Configure(IApplicationBuilder app, DataContext context) {
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseRouting();
            app.UseEndpoints(endpoints => {
                endpoints.MapControllers();
            });
```

```
            SeedData.SeedDatabase(context);
        }
    }
}
```

Dropping the Database

Open a new PowerShell command prompt, navigate to the folder that contains the WebApp.csproj file, and run the command shown in Listing 21-3 to drop the database.

Listing 21-3. Dropping the Database

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

Running the Example Application

Select Start Without Debugging or Run Without Debugging from the Debug menu or use the PowerShell command prompt to run the command shown in Listing 21-4.

Listing 21-4. Running the Example Application

```
dotnet run
```

The database will be seeded as part of the application startup. Once ASP.NET Core is running, use a web browser to request http://localhost:5000/api/products, which will produce the response shown in Figure 21-1.

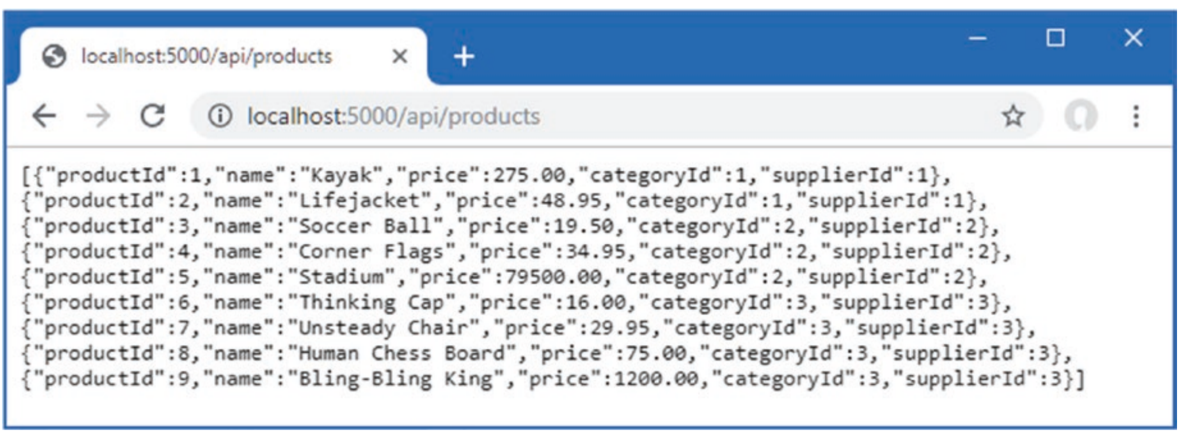

Figure 21-1. *Running the example application*

Getting Started with Views

I started this chapter with a web service controller to demonstrate the similarity with a controller that uses views. It is easy to think about web service and view controllers as being separate, but it is important to understand that the same underlying features are used for both types of response. In the sections that follow, I configure the application to support HTML applications and repurpose the Home controller so that it produces an HTML response.

Configuring the Application

The first step is to configure ASP.NET Core to enable HTML responses, as shown in Listing 21-5.

Listing 21-5. Changing the Configuration in the Startup.cs File in the WebApp Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using WebApp.Models;

namespace WebApp {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        public IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<DataContext>(opts => {
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:ProductConnection"]);
                opts.EnableSensitiveDataLogging(true);
            });
            services.AddControllersWithViews().AddRazorRuntimeCompilation();
        }

        public void Configure(IApplicationBuilder app, DataContext context) {
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseRouting();
            app.UseEndpoints(endpoints => {
                endpoints.MapControllers();
                endpoints.MapControllerRoute("Default",
                    "{controller=Home}/{action=Index}/{id?}");
            });
            SeedData.SeedDatabase(context);
        }
    }
}
```

HTML responses are created using views, which are files containing a mix of HTML elements and C# expressions. The `AddControllers` method I used in Chapter 19 to enable the MVC Framework only supports web service controllers. To enable support for views, the `AddControllersWithViews` method is used. The `AddRazorRuntimeCompilation` method is used to enable the feature provided by the package installed in Listing 21-1, which makes it easier to work with views during development, as explained shortly.

The second change is the addition of the `MapControllerRoute` method in the endpoint routing configuration. Controllers that generate HTML responses don't use the same routing attributes that are applied to web service controllers and rely on a feature named *convention routing,* which I describe in the next section.

Creating an HTML Controller

Controllers for HTML applications are similar to those used for web services but with some important differences. To create an HTML controller, add a class file named `HomeController.cs` to the `Controllers` folder with the statements shown in Listing 21-6.

Listing 21-6. The Contents of the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;
using WebApp.Models;

namespace WebApp.Controllers {

    public class HomeController: Controller {
        private DataContext context;

        public HomeController(DataContext ctx) {
            context = ctx;
        }

        public async Task<IActionResult> Index(long id = 1) {
            return View(await context.Products.FindAsync(id));
        }
    }
}
```

The base class for HTML controllers is `Controller`, which is derived from the `ControllerBase` class used for web service controllers and provides additional methods that are specific to working with views.

```
...
public class HomeController:Controller {
...
```

Action methods in HTML controllers return objects that implement the `IActionResult` interface, which is the same result type used in Chapter 19 to return specific status code responses. The `Controller` base class provides the `View` method, which is used to select a view that will be used to create a response.

```
...
return View(await context.Products.FindAsync(id));
...
```

■ **Tip** Notice that the controller in Listing 21-6 hasn't been decorated with attributes. The `ApiController` attribute is applied only to web service controllers and should not be used for HTML controllers. The `Route` and HTTP method attributes are not required because HTML controllers rely on convention-based routing, which was configured in Listing 21-5 and which is introduced shortly.

The `View` method creates an instance of the `ViewResult` class, which implements the `IActonResult` interface and tells the MVC Framework that a view should be used to produce the response for the client. The argument to the `View` method is called the *view model* and provides the view with the data it needs to generate a response.

There are no views for the MVC Framework to use at the moment, but if you restart ASP.NET Core and use a browser to request `http://localhost:5000`, you will see an error message that shows how the MVC Framework responds to the `ViewResult` it received from the `Index` action method, as shown in Figure 21-2.

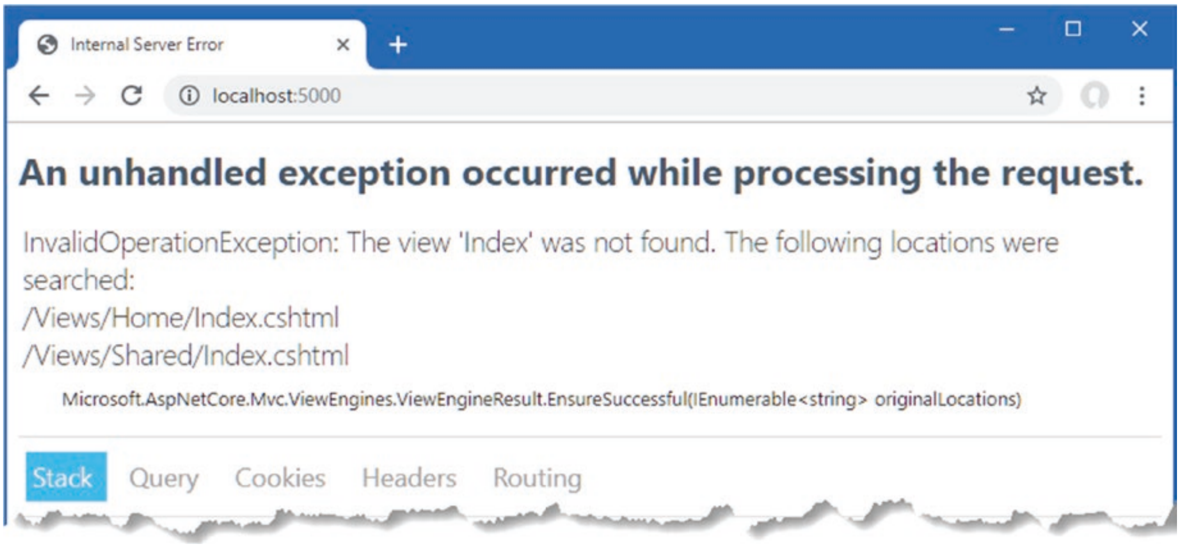

Figure 21-2. *Using a view result*

Behind the scenes, there are two important conventions at work, which are described in the following sections.

■ **Note** There are two features that can expand the range of search locations. The search will include the `/Pages/Shared` folder if the project uses Razor Pages, as explained in Chapter 23.

Understanding Convention Routing

HTML controllers rely on *convention routing* instead of the `Route` attribute. The convention in this term refers to the use of the controller class name and the action method name used to configure the routing system, which was done in Listing 21-6 by adding this statement to the endpoint routing configuration:

```
...
endpoints.MapControllerRoute("Default", "{controller=Home}/{action=Index}/{id?}");
...
```

The route that this statement sets up matches two- and three-segment URLs. The value of the first segment is used as the name of the controller class, without the `Controller` suffix, so that `Home` refers to the `HomeController` class. The second segment is the name of the action method, and the optional third segment allows action methods to receive a parameter named `id`. Default values are used to select the `Index` action method on the `Home` controller for URLs that do not contain all the segments. This is such a common convention that the same routing configuration can be set up without having to specify the URL pattern, as shown in Listing 21-7.

Listing 21-7. Using the Default Routing Convention in the Startup.cs File in the WebApp Folder

```
...
public void Configure(IApplicationBuilder app, DataContext context) {
    app.UseDeveloperExceptionPage();
    app.UseStaticFiles();
    app.UseRouting();
```

```
    app.UseEndpoints(endpoints => {
        endpoints.MapControllers();
        endpoints.MapDefaultControllerRoute();
    });
    SeedData.SeedDatabase(context);
}
...
```

The `MapDefaultControllerRoute` method avoids the risk of mistyping the URL pattern and sets up the convention-based routing. I have configured one route in this chapter, but an application can define as many routes as it needs, and later chapters expand the routing configuration to make examples easier to follow.

■ **Tip** The MVC Framework assumes that any `public` method defined by an HTML controller is an action method and that action methods support all HTTP methods. If you need to define a method in a controller that is not an action, you can make it `private` or, if that is not possible, decorate the method with the `NonAction` attribute. You can restrict an action method to support specific HTTP methods by applying attributes so that the `HttpGet` attribute denotes an action that handles GET requests, the `HttpPost` method denotes an action that handles POST requests, and so on.

Understanding the Razor View Convention

When the `Index` action method defined by the `Home` controller is invoked, it uses the value of the `id` parameter to retrieve an object from the database and passes it to the `View` method.

```
...
public async Task<IActionResult> Index(long id = 1) {
    return View(await context.Products.FindAsync(id));
}
...
```

When an action method invokes the `View` method, it creates a `ViewResult` that tells the MVC Framework to use the default convention to locate a view. The Razor view engine looks for a view with the same name as the action method, with the addition of the `cshtml` file extension, which is the file type used by the Razor view engine. Views are stored in the `Views` folder, grouped by the controller they are associated with. The first location searched is the `Views/Home` folder, since the action method is defined by the `Home` controller (the name of which is taken by dropping `Controller` from the name of the controller class). If the `Index.cshtml` file cannot be found in the `Views/Home` folder, then the `Views/Shared` folder is checked, which is the location where views that are shared between controllers are stored.

While most controllers have their own views, views can also be shared so that common functionality doesn't have to be duplicated, as demonstrated in the "Using Shared Views" section.

The exception response in Figure 21-2 shows the result of both conventions. The routing conventions are used to process the request using the `Index` action method defined by the `Home` controller, which tells the Razor view engine to use the view search convention to locate a view. The view engine uses the name of the action method and controller to build its search pattern and checks for the `Views/Home/Index.cshtml` and `Views/Shared/Index.cshtml` files.

Creating a Razor View

To provide the MVC Framework with a view to display, create the `Views/Home` folder and add to it a file named `Index.cshtml` with the content shown in Listing 21-8. If you are using Visual Studio, create the view by right-clicking the `Views/Home` folder, selecting Add ➤ New Item from the popup menu, and selecting the Razor View item in the ASP.NET Core ➤ Web category, as shown in Figure 21-3.

■ **Tip** There is a menu item for creating views in the Add popup menu, but this relies on the Visual Studio scaffolding feature, which adds template content to create different types of view. I don't rely on the scaffolding in this book and instead show you how to create views from scratch.

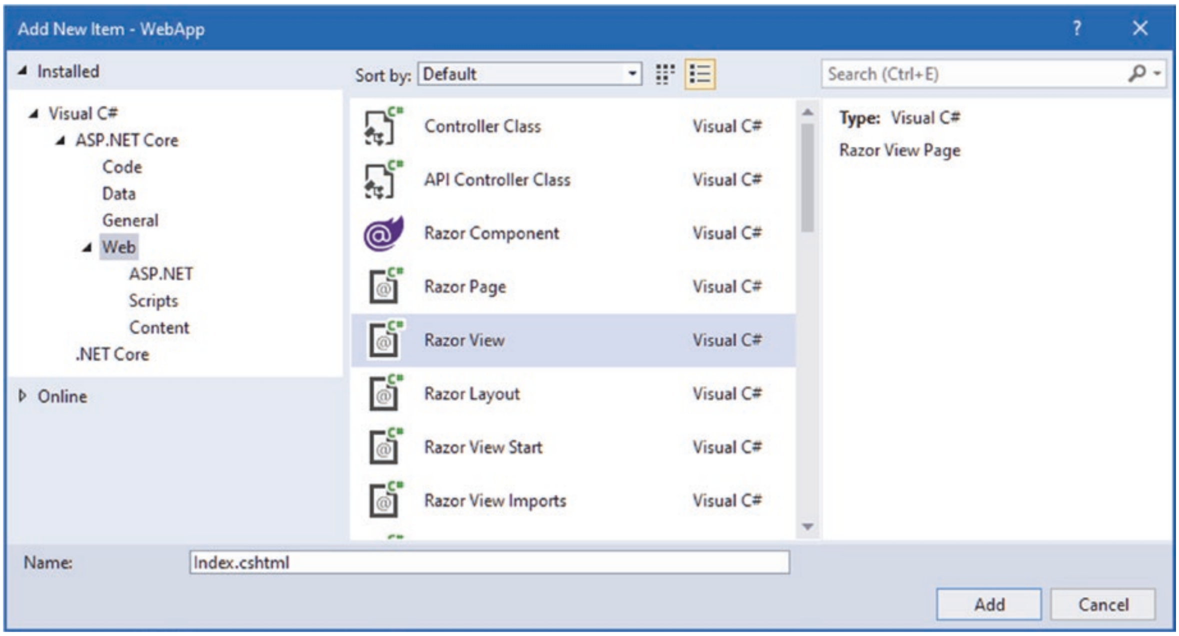

Figure 21-3. *Creating a view using Visual Studio*

Listing 21-8. The Contents of the Index.cshtml File in the Views/Home Folder

```
<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <h6 class="bg-primary text-white text-center m-2 p-2">Product Table</h6>
    <div class="m-2">
        <table class="table table-sm table-striped table-bordered">
            <tbody>
                <tr><th>Name</th><td>@Model.Name</td></tr>
                <tr><th>Price</th><td>@Model.Price.ToString("c")</td></tr>
            </tbody>
        </table>
    </div>
</body>
</html>
```

The view file contains standard HTML elements that are styled using the Bootstrap CSS framework, which is applied through the `class` attribute. The key view feature is the ability to generate content using C# expressions, like this:

```
...
<tr><th>Name</th><td>@Model.Name</td></tr>
<tr><th>Price</th><td>@Model.Price.ToString("c")</td></tr>
...
```

I explain how these expressions work in the "Understanding the Razor Syntax" section, but for now, it is enough to know that these expressions insert the value of the `Name` and `Price` properties from the `Product` view model passed to the `View` method by the action method in Listing 21-6. Restart ASP.NET Core and use a browser to request `http://localhost:5000`, and you will see the HTML response shown in Figure 21-4.

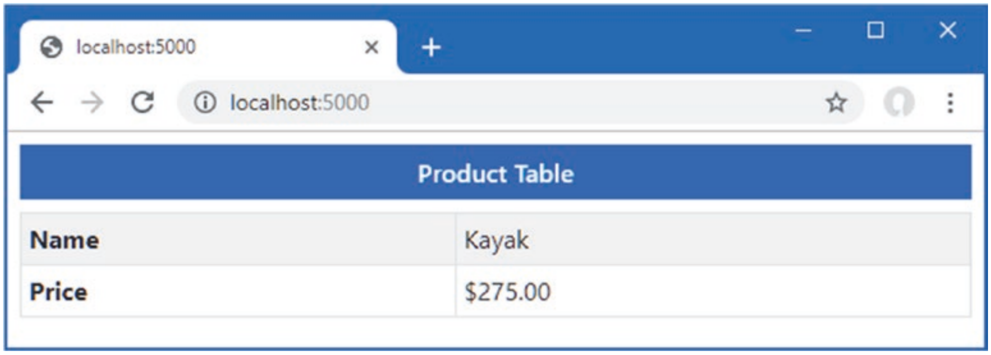

Figure 21-4. *A view response*

Modifying a Razor View

The package I added in Listing 21-1 and configured in Listing 21-5 detects and recompiles Razor views automatically, meaning that the ASP.NET Core runtime doesn't have to be restarted. To demonstrate the recompilation process, Listing 21-9 adds new elements to the `Index` view.

Listing 21-9. Adding Elements in the Index.cshtml File in the Views/Home Folder

```
<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <h6 class="bg-primary text-white text-center m-2 p-2">Product Table</h6>
    <div class="m-2">
        <table class="table table-sm table-striped table-bordered">
            <tbody>
                <tr><th>Name</th><td>@Model.Name</td></tr>
                <tr><th>Price</th><td>@Model.Price.ToString("c")</td></tr>
                <tr><th>Category ID</th><td>@Model.CategoryId</td></tr>
            </tbody>
        </table>
    </div>
</body>
</html>
```

Save the changes to the view and reload the browser window without restarting ASP.NET Core. The changes to the view will be detected, and there will be a brief pause as the views are compiled, after which the response shown in Figure 21-5 will be displayed.

■ **Note** This feature applies only to views and not the C# classes in a project. If you make a change to a class file, then you will have to restart ASP.NET Core for the change to take effect.

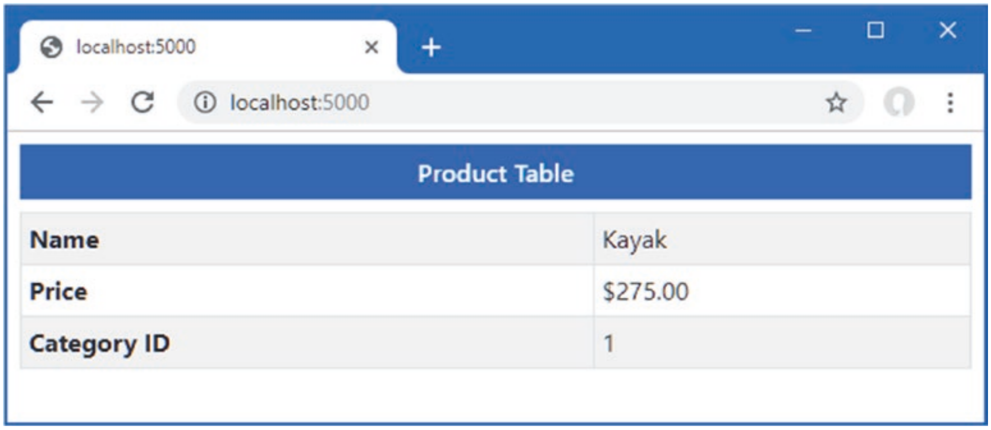

Figure 21-5. *Modifying a Razor view*

Selecting a View by Name

The action method in Listing 21-6 relies entirely on convention, leaving Razor to select the view that is used to generate the response. Action methods can select a view by providing a name as an argument to the `View` method, as shown in Listing 21-10.

Listing 21-10. Selecting a View in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;
using WebApp.Models;

namespace WebApp.Controllers {

    public class HomeController: Controller {
        private DataContext context;

        public HomeController(DataContext ctx) {
            context = ctx;
        }

        public async Task<IActionResult> Index(long id = 1) {
            Product prod = await context.Products.FindAsync(id);
            if (prod.CategoryId == 1) {
                return View("Watersports", prod);
            } else {
                return View(prod);
            }
        }
    }
}
```

The action method selects the view based on the `CategoryId` property of the `Product` object that is retrieved from the database. If the `CategoryId` is 1, the action method invokes the `View` method with an additional argument that selects a view named `Watersports`.

```
...
return View("Watersports", prod);
...
```

Notice that the action method doesn't specify the file extension or the location for the view. It is the job of the view engine to translate `Watersports` into a view file. To create the view, add a Razor view file named `Watersports.cshtml` file to the `Views/Home` folder with the content shown in Listing 21-11.

Listing 21-11. The Contents of the Watersports.cshtml File in the Views/Home Folder

```
<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <h6 class="bg-secondary text-white text-center m-2 p-2">Watersports</h6>
    <div class="m-2">
        <table class="table table-sm table-striped table-bordered">
            <tbody>
                <tr><th>Name</th><td>@Model.Name</td></tr>
                <tr><th>Price</th><td>@Model.Price.ToString("c")</td></tr>
                <tr><th>Category ID</th><td>@Model.CategoryId</td></tr>
            </tbody>
        </table>
    </div>
</body>
</html>
```

The new view follows the same pattern as the `Index` view but has a different title above the table. Since the `HomeController` class has been changed, restart ASP.NET Core and request `http://localhost:5000/home/index/1` and `http://localhost:5000/home/index/4`. The action method selects the `Watersports` view for the first URL and the default view for the second URL, producing the two responses shown in Figure 21-6.

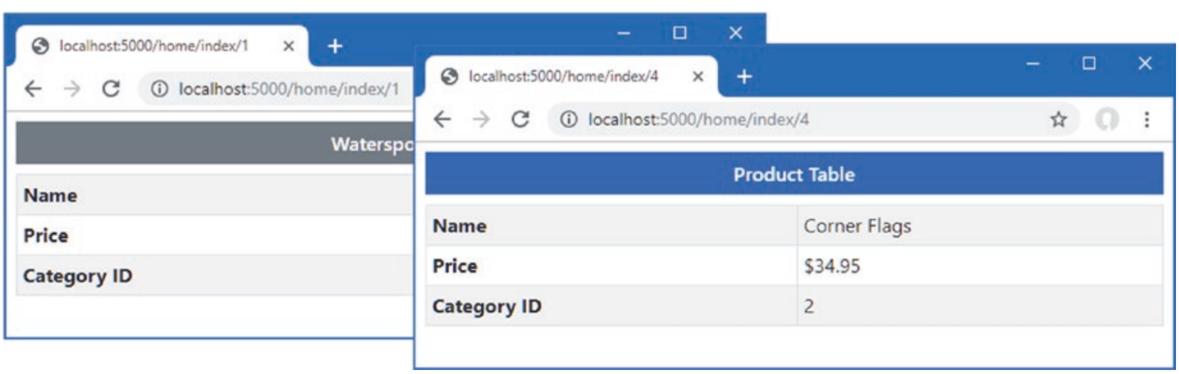

Figure 21-6. *Selecting views*

Using Shared Views

When the Razor view engine locates a view, it looks in the `View/[controller]` folder and then the `Views/Shared` folder. This search pattern means that views that contain common content can be shared between controllers, avoiding duplication. To see how this process works, add a Razor view file named `Common.cshtml` to the `Views/Shared` folder with the content shown in Listing 21-12.

Listing 21-12. The Contents of the Common.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <h6 class="bg-secondary text-white text-center m-2 p-2">Shared View</h6>
</body>
</html>
```

Next, add an action method to the Home controller that uses the new view, as shown in Listing 21-13.

Listing 21-13. Adding an Action in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;
using WebApp.Models;

namespace WebApp.Controllers {

    public class HomeController: Controller {
        private DataContext context;

        public HomeController(DataContext ctx) {
            context = ctx;
        }

        public async Task<IActionResult> Index(long id = 1) {
            Product prod = await context.Products.FindAsync(id);
            if (prod.CategoryId == 1) {
                return View("Watersports", prod);
            } else {
                return View(prod);
            }
        }

        public IActionResult Common() {
            return View();
        }
    }
}
```

The new action relies on the convention of using the method name as the name of the view. When a view doesn't require any data to display to the user, the View method can be called without arguments. Next, create a new controller by adding a class file named SecondController.cs to the Controllers folder, with the code shown in Listing 21-14.

Listing 21-14. The Contents of the SecondController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;

namespace WebApp.Controllers {

    public class SecondController : Controller {
```

```
        public IActionResult Index() {
            return View("Common");
        }
    }
}
```

The new controller defines a single action, named `Index`, which invokes the `View` method to select the `Common` view. Restart ASP.NET Core and navigate to `http://localhost:5000/home/common` and `http://localhost:5000/second`, both of which will render the `Common` view, producing the responses shown in Figure 21-7.

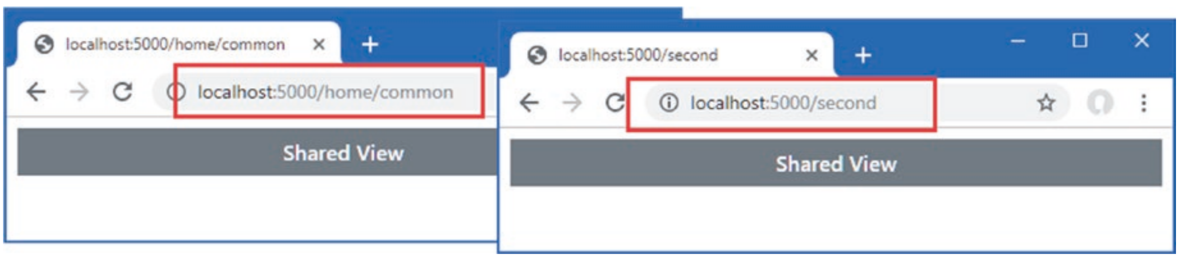

Figure 21-7. *Using a shared view*

SPECIFYING A VIEW LOCATION

The Razor view engine will look for a controller-specific view before a shared view. You can change this behavior by specifying the complete path to a view file, which can be useful if you want to select a shared view that would otherwise be ignored because there is a controller-specific view with the same name.

```
...
public IActionResult Index() {
    return View("/Views/Shared/Common.cshtml");
}
...
```

When specifying the view, the path relative to the project folder must be specified, starting with the `/` character. Notice that the full name of the file, including the file extension, is used.

This is a technique that should be used sparingly because it creates a dependency on a specific file, rather than allowing the view engine to select the file.

Working with Razor Views

Razor views contain HTML elements and C# expressions. Expressions are mixed in with the HTML elements and denoted with the `@` character, like this:

```
...
<tr><th>Name</th><td>@Model.Name</td></tr>
...
```

When the view is used to generate a response, the expressions are evaluated, and the results are included in the content sent to the client. This expression gets the name of the `Product` view model object provided by the action method and produces output like this:

```
...
<tr><th>Name</th><td>Corner Flags</td></tr>
...
```

This transformation can seem like magic, but Razor is simpler than it first appears. Razor views are converted into C# classes that inherit from the `RazorPage` class, which are then compiled like any other C# class.

■ **Tip** You can see the generated view classes by examining the contents of the `obj/Debug/netcoreapp3.0/Razor/Views` folder with the Windows File Explorer.

The view from Listing 21-11, for example, would be transformed into a class like this:

```
using Microsoft.AspNetCore.Mvc.Razor;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.ViewFeatures;

namespace AspNetCore {

    public class Views_Home_Watersports : RazorPage<dynamic> {

        public async override Task ExecuteAsync() {
            WriteLiteral("<!DOCTYPE html>\r\n<html>\r\n");
            WriteLiteral("<head>");
            WriteLiteral(@"<link
                href=""/lib/twitter-bootstrap/css/bootstrap.min.css""
                rel=""stylesheet"" />");
            WriteLiteral("</head>");
            WriteLiteral("<body>");
            WriteLiteral(@"<h6 class=""bg-secondary text-white text-center
                m-2 p-2"">Watersports</h6>\r\n<div class=""m-2"">\r\n<table
                class=""table table-sm table-striped table-bordered"">\r\n
                <tbody>\r\n>");
            WriteLiteral("<th>Name</th><td>");
            Write(Model.Name);
            WriteLiteral("</td></tr>");
            WriteLiteral("<tr><th>Price</th><td>");
            Write(Model.Price.ToString("c"));
            WriteLiteral("</td></tr>\r\n<tr><th>Category ID</th><td>");
            Write(Model.CategoryId);
            WriteLiteral("</td></tr>\r\n</tbody>\r\n</table>\r\n</div>");
            WriteLiteral("</body></html>");
        }

        public IUrlHelper Url { get; private set; }
        public IViewComponentHelper Component { get; private set; }
        public IJsonHelper Json { get; private set; }
        public IHtmlHelper<dynamic> Html { get; private set; }
        public IModelExpressionProvider ModelExpressionProvider { get; private set; }
    }
}
```

This class is a simplification of the code that is generated so that I can focus on the features that are most important for this chapter. The first point to note is that the class generated from the view inherits from the RazorPage<T> class.

```
...
public class Views_Home_Watersports : RazorPage<dynamic> {
...
```

Table 21-3 describes the most useful properties and methods defined by RazorPage<T>.

CACHING RESPONSES

Responses from views can be cached by applying the ResponseCache attribute to action methods (or to the controller class, which caches the responses from all the action methods). See Chapter 17 for details of how response caching is enabled.

Table 21-3. *The RazorPage<T> Members*

Name	Description
Context	This property returns the HttpContext object for the current request.
Layout	This property is used to set the view layout, as described in Chapter 22.
Model	This property returns the view model passed to the View method by the action.
RenderBody()	This method is used in layouts to include content from a view, as described in Chapter 22.
RenderSection()	This method is used in layouts to include content from a section in a view, as described in Chapter 22.
TempData	This property is used to access the temp data feature, which is described in Chapter 22.
ViewBag	This property is used to access the view bag, which is described in Chapter 22.
ViewContext	This property returns a ViewContext object that provides context data.
ViewData	This property returns the view data, which I used for unit testing controllers in the SportsStore application.
Write(str)	This method writes a string, which will be safely encoded for use in HTML.
WriteLiteral(str)	This method writes a string without encoding it for safe use in HTML.

The expressions in the view are translated into calls to the Write method, which encodes the result of the expression so that it can be included safely in an HTML document. The WriteLiteral method is used to deal with the static HTML regions of the view, which don't need further encoding.

■ **Tip** See Chapter 22 for more details about HTML encoding.

The result is a fragment like this from the CSHTML file:

```
...
<tr><th>Name</th><td>@Model.Name</td></tr>
...
```

This is converted into a series of C# statements like these in the ExecuteAsync method:

```
...
WriteLiteral("<th>Name</th><td>");
Write(Model.Name);
WriteLiteral("</td></tr>");
...
```

When the ExecuteAsync method is invoked, the response is generated with a mix of the static HTML and the expressions contained in the view. When the statements in the generated class are executed, the combination of the HTML fragments and the results from evaluating the expressions are written to the response, producing HTML like this:

```
...
<th>Name</th><td>Kayak</td></tr>
...
```

In addition to the properties and methods inherited from the RazorPage<T> class, the generated view class defines the properties described in Table 21-4, some of which are used for features described in later chapters.

Table 21-4. *The Additional View Class Properties*

Name	Description
Component	This property returns a helper for working with view components, which is accessed through the vc tag helper described in Chapter 25.
Html	This property returns an implementation of the IHtmlHelper interface. This property is used to manage HTML encoding, as described in Chapter 22.
Json	This property returns an implementation of the IJsonHelper interface, which is used to encode data as JSON, as described in Chapter 22.
ModelExpressionProvider	This property provides access to expressions that select properties from the model, which is used through tag helpers, described in Chapters 25–27.
Url	This property returns a helper for working with URLs, as described in Chapter 26.

Setting the View Model Type

The generated class for the Watersports.cshtml file is derived from RazorPage<T>, but Razor doesn't know what type will be used by the action method for the view model, so it has selected dynamic as the generic type argument. This means that the @Model expression can be used with any property or method name, which is evaluated at runtime when a response is generated. To demonstrate what happens when a nonexistent member is used in an exception, add the content shown in Listing 21-15 to the Watersports.cshtml file.

Listing 21-15. Adding Content in the Watersports.cshtml File in the Views/Home Folder

```
<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <h6 class="bg-secondary text-white text-center m-2 p-2">Watersports</h6>
    <div class="m-2">
        <table class="table table-sm table-striped table-bordered">
            <tbody>
                <tr><th>Name</th><td>@Model.Name</td></tr>
                <tr><th>Price</th><td>@Model.Price.ToString("c")</td></tr>
                <tr><th>Category ID</th><td>@Model.CategoryId</td></tr>
                <tr><th>Tax Rate</th><td>@Model.TaxRate</td></tr>
            </tbody>
        </table>
    </div>
</body>
</html>
```

Use a browser to request `http://localhost:5000`, and you will see the exception shown in Figure 21-8.

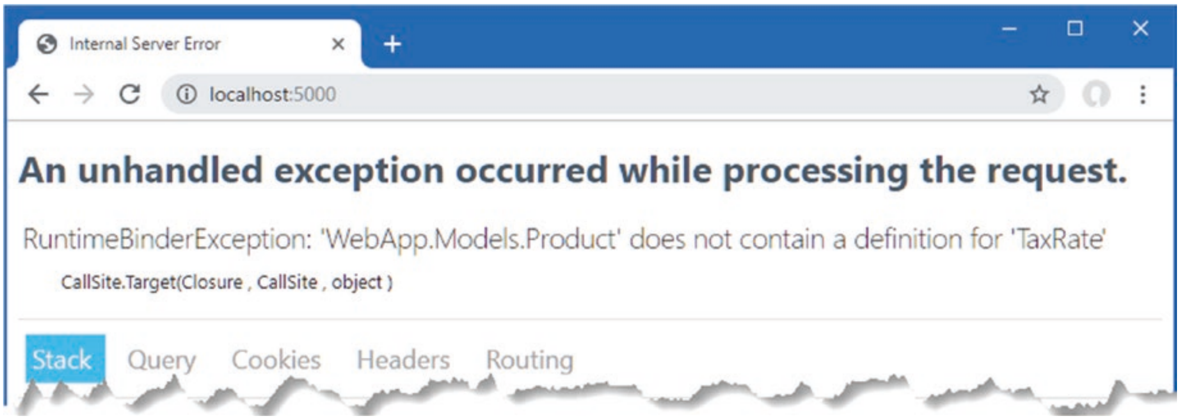

Figure 21-8. *Using a nonexistent property in a view expression*

To check expressions during development, the type of the `Model` object can be specified using the `model` keyword, as shown in Listing 21-16.

■ **Tip** It is easy to get the two terms confused. `Model`, with an uppercase `M`, is used in expressions to access the view model object provided by the action method, while `model`, with a lowercase `m`, is used to specify the type of the view model.

Listing 21-16. Declaring the Model Type in the Watersports.cshtml File in the Views/Home Folder

```
@model WebApp.Models.Product
<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <h6 class="bg-secondary text-white text-center m-2 p-2">Watersports</h6>
    <div class="m-2">
        <table class="table table-sm table-striped table-bordered">
            <tbody>
                <tr><th>Name</th><td>@Model.Name</td></tr>
                <tr><th>Price</th><td>@Model.Price.ToString("c")</td></tr>
                <tr><th>Category ID</th><td>@Model.CategoryId</td></tr>
                <tr><th>Tax Rate</th><td>@Model.TaxRate</td></tr>
            </tbody>
        </table>
    </div>
</body>
</html>
```

An error warning will appear in the editor after a few seconds, as Visual Studio or Visual Studio Code checks the view in the background, as shown in Figure 21-9. The compiler will also report an error if you build the project or use the `dotnet build` or `dotnet run` command.

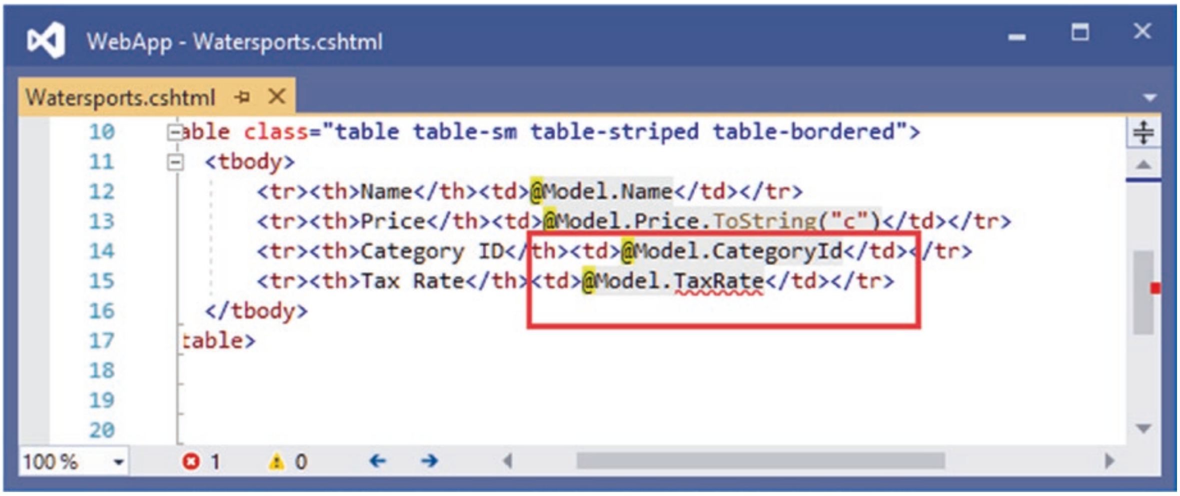

Figure 21-9. *An error warning in a view file*

When the C# class for the view is generated, the view model type is used as the generic type argument for the base class, like this:

```
...
public class Views_Home_Watersports : RazorPage<Product> {
...
```

Specifying a view model type allows Visual Studio and Visual Studio Code to suggest property and method names as you edit views. Replace the nonexistent property with the one shown in Listing 21-17.

Listing 21-17. Replacing a Property in the Watersports.cshtml File in the Views/Home Folder

```
@model WebApp.Models.Product
<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <h6 class="bg-secondary text-white text-center m-2 p-2">Watersports</h6>
    <div class="m-2">
        <table class="table table-sm table-striped table-bordered">
            <tbody>
                <tr><th>Name</th><td>@Model.Name</td></tr>
                <tr><th>Price</th><td>@Model.Price.ToString("c")</td></tr>
                <tr><th>Category ID</th><td>@Model.CategoryId</td></tr>
                <tr><th>Supplier ID</th><td>@Model.SupplierId</td></tr>
            </tbody>
        </table>
    </div>
</body>
</html>
```

As you type, the editor will prompt you with the possible member names defined by the view model class, as shown in Figure 21-10. This figure shows the Visual Studio code editor, but Visual Studio Code has a comparable feature.

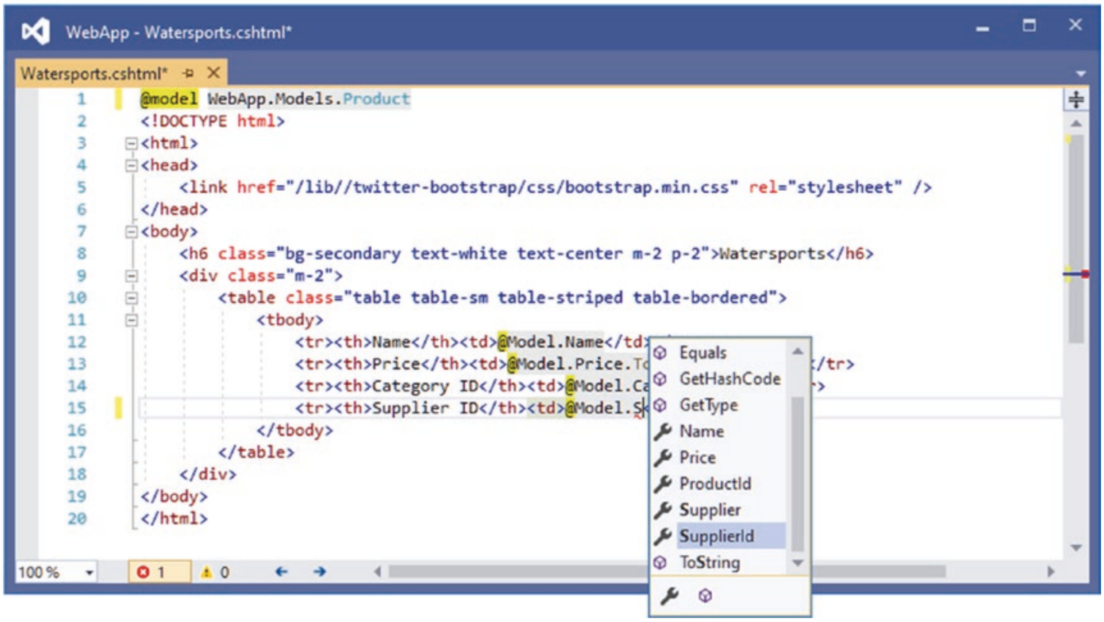

Figure 21-10. Editor suggestions when using a view model type

Using a View Imports File

When I declared the view model object at the start of the `Watersports.cshtml` file, I had to include the namespace that contains the class, like this:

```
...
@model WebApp.Models.Product
...
```

By default, all types that are referenced in a Razor view must be qualified with a namespace. This isn't a big deal when the only type reference is for the model object, but it can make a view more difficult to read when writing more complex Razor expressions such as the ones I describe later in this chapter.

You can specify a set of namespaces that should be searched for types by adding a *view imports* file to the project. The view imports file is placed in the `Views` folder and is named `_ViewImports.cshtml`.

■ **Note** Files in the `Views` folder whose names begin with an underscore (the _ character) are not returned to the user, which allows the file name to differentiate between views that you want to render and the files that support them. View imports files and layouts (which I describe shortly) are prefixed with an underscore.

If you are using Visual Studio, right-click the `Views` folder in the Solution Explorer, select Add ➤ New Item from the pop-up menu, and select the Razor View Imports template from the ASP.NET Core category, as shown in Figure 21-11.

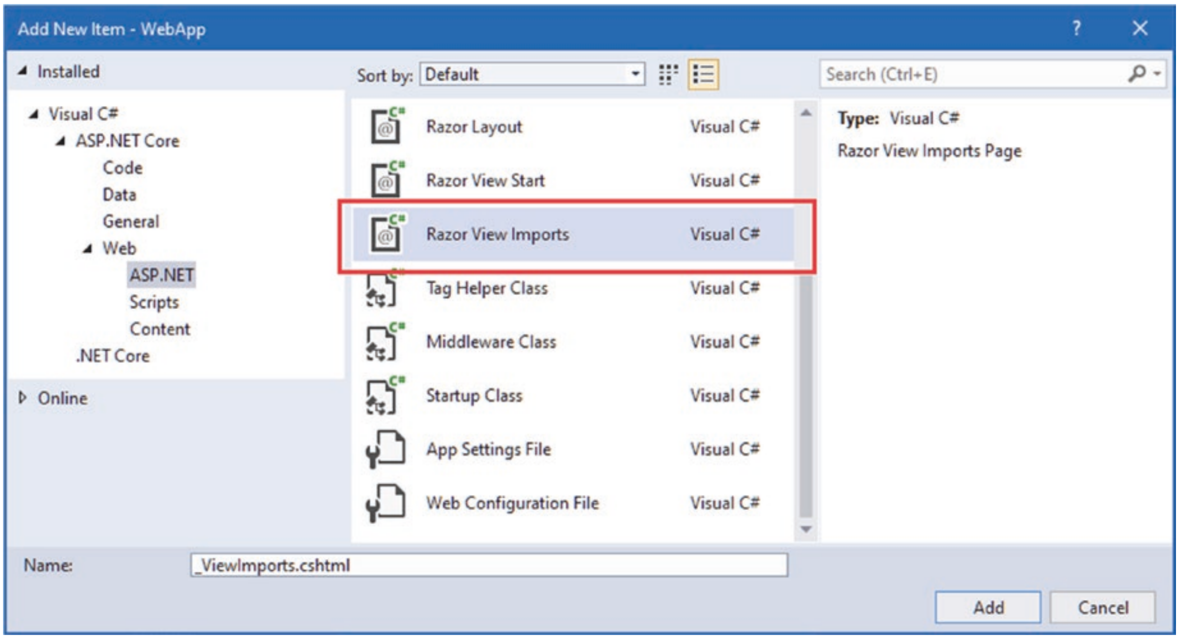

Figure 21-11. *Creating a view imports file*

Visual Studio will automatically set the name of the file to `_ViewImports.cshtml`, and clicking the Add button will create the file, which will be empty. If you are using Visual Studio Code, simply select the `Views` folder and add a new file called `_ViewImports.cshtml`.

Regardless of which editor you used, add the expression shown Listing 21-18.

Listing 21-18. The Contents of the _ViewImports.cshtml File in the Views Folder

```
@using WebApp.Models
```

The namespaces that should be searched for classes used in Razor views are specified using the `@using` expression, followed by the namespace. In Listing 21-18, I have added an entry for the `WebApp.Models` namespace that contains the view model class used in the `Watersports.cshtml` view.

Now that the namespace is included in the view imports file, I can remove the namespace from the view, as shown in Listing 21-19.

■ **Tip** You can also add an `@using` expression to individual view files, which allows types to be used without namespaces in a single view.

Listing 21-19. Simplifying the Model Type in the Watersports.cshtml File in the Views/Home Folder

```
@model Product
<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
```

```
    <h6 class="bg-secondary text-white text-center m-2 p-2">Watersports</h6>
    <div class="m-2">
        <table class="table table-sm table-striped table-bordered">
            <tbody>
                <tr><th>Name</th><td>@Model.Name</td></tr>
                <tr><th>Price</th><td>@Model.Price.ToString("c")</td></tr>
                <tr><th>Category ID</th><td>@Model.CategoryId</td></tr>
                <tr><th>Supplier ID</th><td>@Model.SupplierId</td></tr>
            </tbody>
        </table>
    </div>
</body>
</html>
```

Save the view file and use a browser to request `http://localhost:5000`, and you will see the response shown in Figure 21-12.

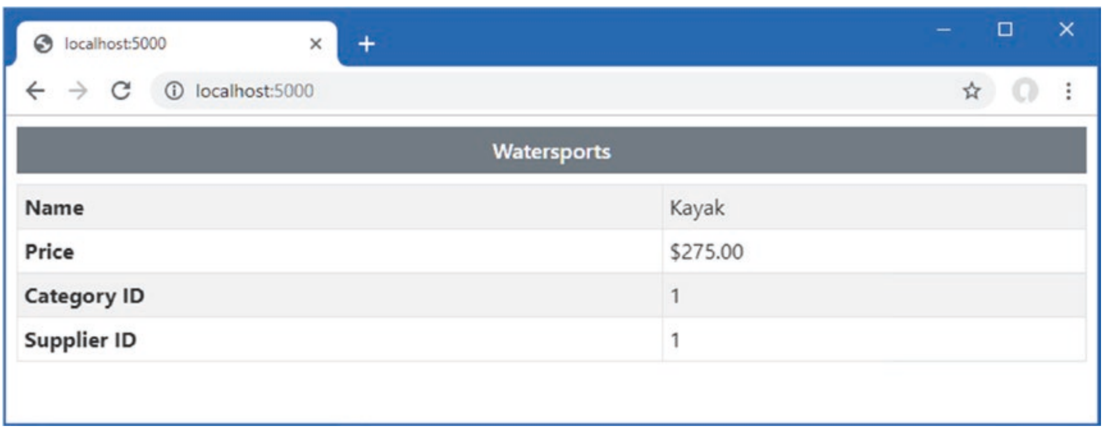

Figure 21-12. *Using a view imports file*

Understanding the Razor Syntax

The Razor compiler separates the static fragments of HTML from the C# expressions, which are then handled separately in the generated class file. There are several types of expression that can be included in views, which I describe in the sections that follow.

Understanding Directives

Directives are expressions that give instructions to the Razor view engine. The `@model` expression is a directive, for example, that tells the view engine to use a specific type for the view model, while the `@using` directive tells the view engine to import a namespace. Table 21-5 describes the most useful Razor directives.

Table 21-5. *Useful Razor Directives*

Name	Description
`@model`	This directive specifies the type of the view model.
`@using`	This directive imports a namespace.
`@page`	This directive denotes a Razor Page, described in Chapter 23.
`@section`	This directive denotes a layout section, as described In Chapter 22.
`@addTagHelper`	This directive adds tag helpers to a view, as described in Chapter 25.
`@namespace`	This directive sets the namespace for the C# class generated from a view.
`@functions`	This directive adds C# properties and methods to the C# class generated from a view and is commonly used in Razor Pages, as described in Chapter 23.
`@attribute`	This directive adds an attribute to the C# class generated from a view. I use this feature to apply authorization restrictions in Chapter 38.
`@implements`	This directive declares that the C# class generated from a view inherits an interface or is derived from a base class. This feature is demonstrated in Chapter 33.
`@inherits`	This directive sets the base class for the C# class generated from a view. This feature is demonstrated in Chapter 33.
`@inject`	This directive provides a view with direct access to a service through dependency injection. This feature is demonstrated in Chapter 23.

Understanding Content Expressions

Razor content expressions produce content that is included in the output generated by a view. Table 21-6 describes the most useful content expressions, which are demonstrated in the sections that follow.

Table 21-6. *Useful Razor Content Expressions*

Name	Description
`@<expression>`	This is the basic Razor expression, which is evaluated, and the result it produces is inserted into the response.
`@if`	This expression is used to select regions of content based on the result of an expression. See the “Using Conditional Expressions” section for examples.
`@switch`	This expression is used to select regions of content based on the result of an expression. See the “Using Conditional Expressions” section for examples.
`@foreach`	This expression generates the same region of content for each element in a sequence. See the “Enumerating Sequences” for examples.
`@{ ... }`	This expression defines a code block. See the “Using Razor Code Blocks” section for an example.
`@:`	This expression denotes a section of content that is not enclosed in HTML elements. See the “Using Conditional Expressions” section for an example.
`@try`	This expression is used to catch exceptions.
`@await`	This expression is used to perform an asynchronous operation, the result of which is inserted into the response. See Chapter 24 for examples.

Setting Element Content

The simplest expressions are evaluated to produce a single value that is used as the content for an HTML element in the response sent to the client. The most common type of expression inserts a value from the view model object, like these expressions from the `Watersports.cshtml` view file:

```
...
<tr><th>Name</th><td>@Model.Name</td></tr>
<tr><th>Price</th><td>@Model.Price.ToString("c")</td></tr>
...
```

This type of expression can read property values or invoke methods, as these examples demonstrate. Views can contain more complex expressions, but these need to be enclosed in parentheses so that the Razor compiler can differentiate between the code and static content, as shown in Listing 21-20.

Listing 21-20. Adding Expressions in the Watersports.cshtml File in the Views/Home Folder

```
@model Product
<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <h6 class="bg-secondary text-white text-center m-2 p-2">Watersports</h6>
    <div class="m-2">
        <table class="table table-sm table-striped table-bordered">
            <tbody>
                <tr><th>Name</th><td>@Model.Name</td></tr>
                <tr><th>Price</th><td>@Model.Price.ToString("c")</td></tr>
                <tr><th>Tax</th><td>@Model.Price * 0.2m</td></tr>
                <tr><th>Tax</th><td>@(Model.Price * 0.2m)</td></tr>
            </tbody>
        </table>
    </div>
</body>
</html>
```

Use a browser to request `http://localhost:5000`; the response, shown in Figure 21-13, shows why parentheses are important.

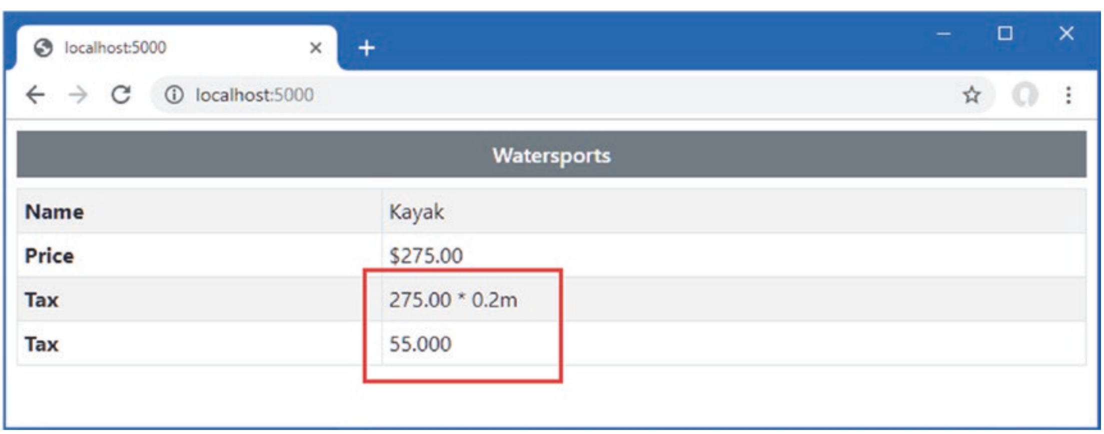

Figure 21-13. *Expressions with and without parentheses*

The Razor view compiler matches expressions conservatively and has assumed that the asterisk and the numeric value in the first expression are static content. This problem is avoided by parentheses for the second expression.

Setting Attribute Values

An expression can be used to set the values of element attributes, as shown in Listing 21-21.

Listing 21-21. Setting an Attribute in the Watersports.cshtml File in the Views/Home Folder

```
@model Product
<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <h6 class="bg-secondary text-white text-center m-2 p-2">Watersports</h6>
    <div class="m-2">
        <table class="table table-sm table-striped table-bordered"
               data-id="@Model.ProductId">
            <tbody>
                <tr><th>Name</th><td>@Model.Name</td></tr>
                <tr><th>Price</th><td>@Model.Price.ToString("c")</td></tr>
                <tr><th>Tax</th><td>@Model.Price * 0.2m</td></tr>
                <tr><th>Tax</th><td>@(Model.Price * 0.2m)</td></tr>
            </tbody>
        </table>
    </div>
</body>
</html>
```

I used the Razor expressions to set the value for some `data` attributes on the `table` element.

■ **Tip** Data attributes, which are attributes whose names are prefixed by `data-`, have been an informal way of creating custom attributes for many years and have been made part of the formal standard as part of HTML5. They are most often applied so that JavaScript code can locate specific elements or so that CSS styles can be more narrowly applied.

If you request `http://localhost:5000` and look at the HTML source that is sent to the browser, you will see that Razor has set the values of the attribute, like this:

```
...
<table class="table table-sm table-striped table-bordered" data-id="1">
    <tbody>
        <tr><th>Name</th><td>Kayak</td></tr>
        <tr><th>Price</th><td>$275.00</td></tr>
        <tr><th>Tax</th><td>275.00 * 0.2m</td></tr>
        <tr><th>Tax</th><td>55.000</td></tr>
    </tbody>
</table>
...
```

Using Conditional Expressions

Razor supports conditional expressions, which means that the output can be tailored based on the view model. This technique is at the heart of Razor and allows you to create complex and fluid responses from views that are simple to read and maintain. In Listing 21-22, I have added a conditional statement to the `Watersports` view.

Listing 21-22. Using an If Expression in the Watersports.cshtml File in the Views/Home Folder

```
@model Product
<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <h6 class="bg-secondary text-white text-center m-2 p-2">Watersports</h6>
    <div class="m-2">
        <table class="table table-sm table-striped table-bordered"
               data-id="@Model.ProductId">
            <tbody>
                @if (Model.Price > 200) {
                    <tr><th>Name</th><td>Luxury @Model.Name</td></tr>
                } else {
                    <tr><th>Name</th><td>Basic @Model.Name</td></tr>
                }
                <tr><th>Price</th><td>@Model.Price.ToString("c")</td></tr>
                <tr><th>Tax</th><td>@Model.Price * 0.2m</td></tr>
                <tr><th>Tax</th><td>@(Model.Price * 0.2m)</td></tr>
            </tbody>
        </table>
    </div>
</body>
</html>
```

The `@` character is followed by the `if` keyword and a condition that will be evaluated at runtime. The `if` expression supports optional `else` and `elseif` clauses and is terminated with a close brace (the } character). If the condition is met, then the content in the `if` clause is inserted into the response; otherwise, the content in the `else` clause is used instead.

Notice that the `@` prefix isn't required to access a `Model` property in the condition.

```
...
@if (Model.Price > 200) {
...
```

But the `@` prefix is required inside the `if` and `else` clauses, like this:

```
...
<tr><th>Name</th><td>Luxury @Model.Name</td></tr>
...
```

To see the effect of the conditional statement, use a browser to request `http://localhost:5000/home/index/1` and `http://localhost:5000/home/index/2`. The conditional statement will produce different HTML elements for these URLs, as shown in Figure 21-14.

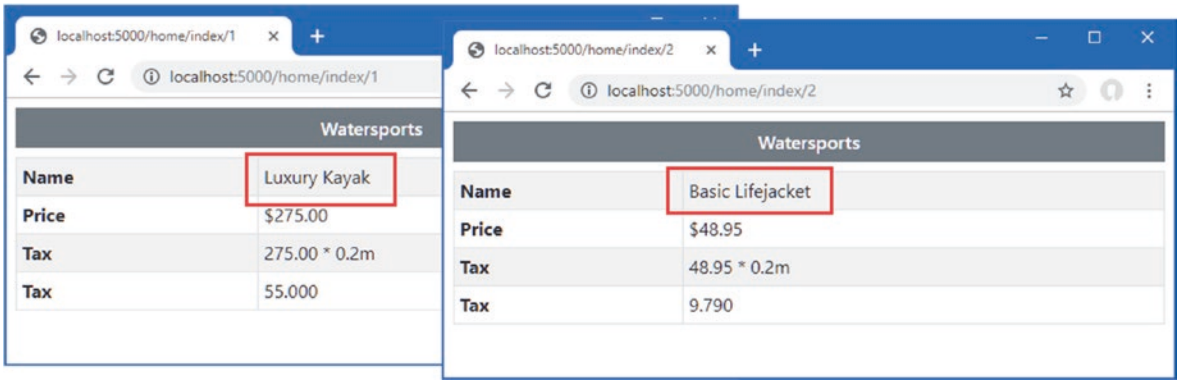

Figure 21-14. *Using a conditional statement*

Razor also supports `@switch` expressions, which can be a more concise way of handling multiple conditions, as shown in Listing 21-23.

Listing 21-23. Using a Switch Expression in the Watersports.cshtml File in the Views/Home Folder

```
@model Product
<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <h6 class="bg-secondary text-white text-center m-2 p-2">Watersports</h6>
    <div class="m-2">
        <table class="table table-sm table-striped table-bordered"
               data-id="@Model.ProductId">
            <tbody>
                @switch (Model.Name) {
                    case "Kayak":
                        <tr><th>Name</th><td>Small Boat</td></tr>
                        break;
                    case "Lifejacket":
                        <tr><th>Name</th><td>Flotation Aid</td></tr>
                        break;
                    default:
                        <tr><th>Name</th><td>@Model.Name</td></tr>
                        break;
                }
                <tr><th>Price</th><td>@Model.Price.ToString("c")</td></tr>
                <tr><th>Tax</th><td>@Model.Price * 0.2m</td></tr>
                <tr><th>Tax</th><td>@(Model.Price * 0.2m)</td></tr>
            </tbody>
        </table>
    </div>
</body>
</html>
```

Conditional expressions can lead to the same blocks of content being duplicated for each result clause. In the `switch` expression, for example, each `case` clause differs only in the content of the `td` element, while the `tr` and `th` elements remain the same. To remove this duplication, conditional expressions can be used within an element, as shown in Listing 21-24.

Listing 21-24. Setting Content in the Watersports.cshtml File in the Views/Home Folder

```
@model Product
<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <h6 class="bg-secondary text-white text-center m-2 p-2">Watersports</h6>
    <div class="m-2">
        <table class="table table-sm table-striped table-bordered"
               data-id="@Model.ProductId">
            <tbody>
                <tr><th>Name</th><td>
                @switch (Model.Name) {
                    case "Kayak":
                        @:Small Boat
                        break;
                    case "Lifejacket":
                        @:Flotation Aid
                        break;
                    default:
                        @Model.Name
                        break;
                }
                </td></tr>
                <tr><th>Price</th><td>@Model.Price.ToString("c")</td></tr>
                <tr><th>Tax</th><td>@Model.Price * 0.2m</td></tr>
                <tr><th>Tax</th><td>@(Model.Price * 0.2m)</td></tr>
            </tbody>
        </table>
    </div>
</body>
</html>
```

The Razor compiler needs help with literal values that are not enclosed in HTML elements, requiring the `@:` prefix, like this:

```
...
@:Small Boat
...
```

The compiler copes with HTML elements because it detects the open tag, but this additional help is required for text content. To see the effect of the `switch` statement, use a web browser to request `http://localhost:5000/home/index/2`, which produces the response shown in Figure 21-15.

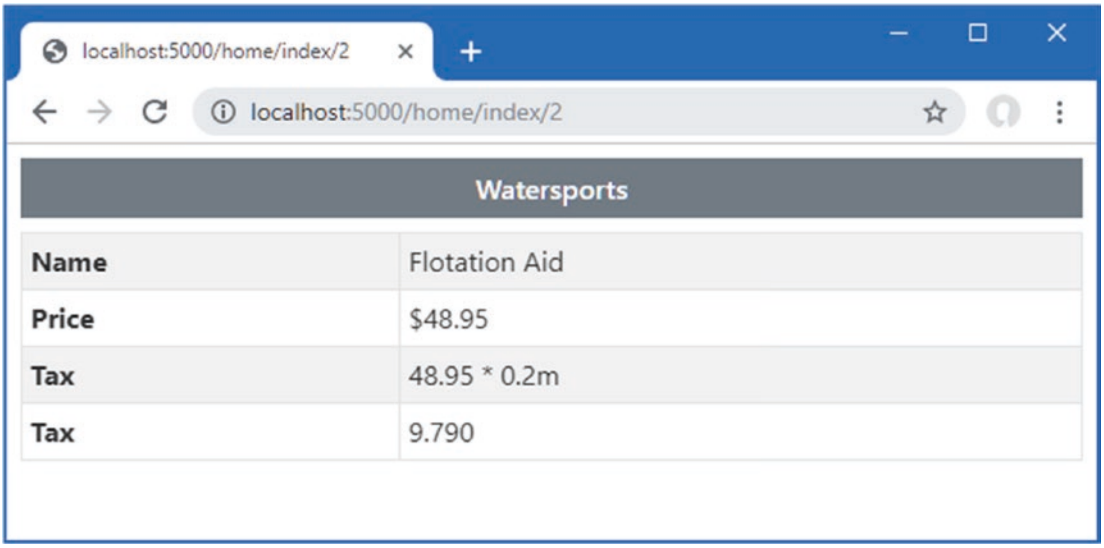

Figure 21-15. *Using a switch expression with literal content*

Enumerating Sequences

The Razor @foreach expression generates content for each object in an array or a collection, which is a common requirement when processing data. Listing 21-25 adds an action method to the Home controller that produces a sequence of objects.

Listing 21-25. Adding an Action in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;
using WebApp.Models;

namespace WebApp.Controllers {

    public class HomeController : Controller {
        private DataContext context;

        public HomeController(DataContext ctx) {
            context = ctx;
        }

        public async Task<IActionResult> Index(long id = 1) {
            Product prod = await context.Products.FindAsync(id);
            if (prod.CategoryId == 1) {
                return View("Watersports", prod);
            } else {
                return View(prod);
            }
        }

        public IActionResult Common() {
            return View();
        }
```

```
        public IActionResult List() {
            return View(context.Products);
        }
    }
}
```

The new action is called `List`, and it provides its view with the sequence of `Product` objects obtained from the Entity Framework Core data context. Add a Razor view file named `List.cshtml` to the `Views/Home` folder and add the content shown in Listing 21-26.

Listing 21-26. The Contents of the List.cshtml File in the Views/Home Folder

```
@model IEnumerable<Product>
<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <h6 class="bg-secondary text-white text-center m-2 p-2">Products</h6>
    <div class="m-2">
        <table class="table table-sm table-striped table-bordered">
            <thead>
                <tr><th>Name</th><th>Price</th></tr>
            </thead>
            <tbody>
                @foreach (Product p in Model) {
                    <tr><td>@p.Name</td><td>@p.Price</td></tr>
                }
            </tbody>
        </table>
    </div>
</body>
</html>
```

The `foreach` expression follows the same format as the C# `foreach` statement. In the example, the variable `p` is assigned each object in the sequence provided by the action method. The content within the expression is duplicated for each object and inserted into the response after the expressions it contains are evaluated. In this case, the content in the `foreach` expression generates a table row with cells that have their own expressions.

```
...
<tr><td>@p.Name</td><td>@p.Price</td></tr>
...
```

Restart ASP.NET Core so that the new action method will be available and use a browser to request `http://localhost:5000/home/list`, which produces the result shown in Figure 21-16, showing how the `foreach` expression populates a table body.

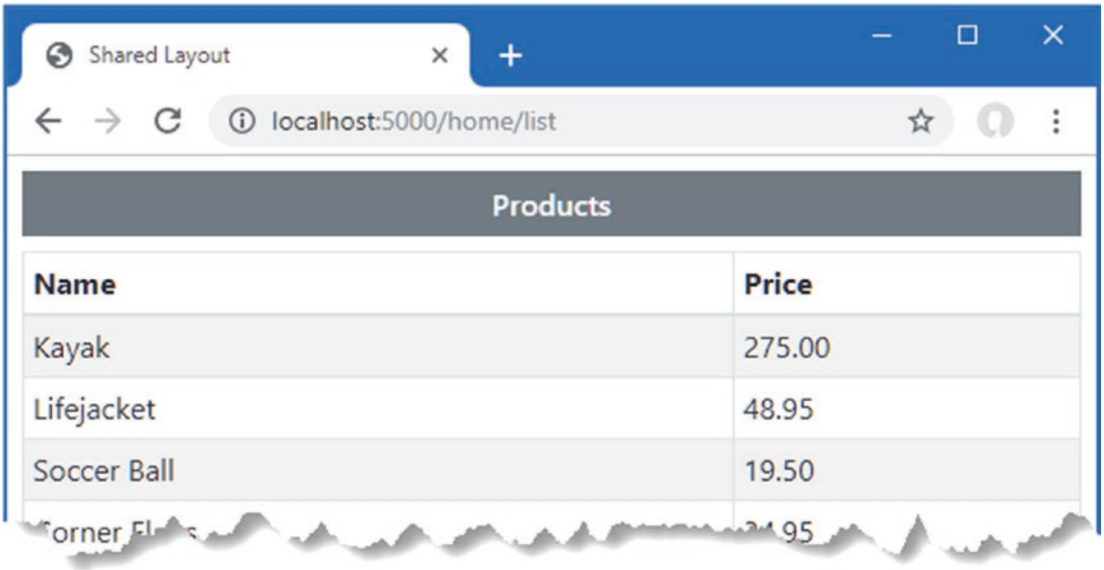

Figure 21-16. *Using a foreach expression*

Using Razor Code Blocks

Code blocks are regions of C# content that do not generate content but that can be useful to perform tasks that support the expressions that do. Listing 21-27 adds a code block that calculates an average value.

■ **Tip** The most common use of code blocks is to select a layout, which is described in Chapter 21.

Listing 21-27. Using a Code Block in the List.cshtml File in the Views/Home Folder

```
@model IEnumerable<Product>
@{
    decimal average = Model.Average(p => p.Price);
}
<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <h6 class="bg-secondary text-white text-center m-2 p-2">Products</h6>
    <div class="m-2">
        <table class="table table-sm table-striped table-bordered">
            <thead>
                <tr><th>Name</th><th>Price</th><th></th></tr>
            </thead>
            <tbody>
                @foreach (Product p in Model) {
                    <tr>
                        <td>@p.Name</td><td>@p.Price</td>
                        <td>@((p.Price / average * 100).ToString("F1"))
                                % of average</td>
                    </tr>
                }
```

```
            </tbody>
        </table>
    </div>
</body>
</html>
```

The code block is denoted by `@{` and `}` and contains standard C# statements. The code block in Listing 21-27 uses LINQ to calculate a value that is assigned to a variable named `average`, which is used in an expression to set the contents of a table cell, avoiding the need to repeat the average calculation for each object in the view model sequence. Use a browser to request `http://localhost:5000/home/list`, and you will see the response shown in Figure 21-17.

■ **Note** Code blocks can become difficult to manage if they contain more than a few statements. For more complex tasks, consider using the view bag, described in Chapter 22, or adding a nonaction method to the controller.

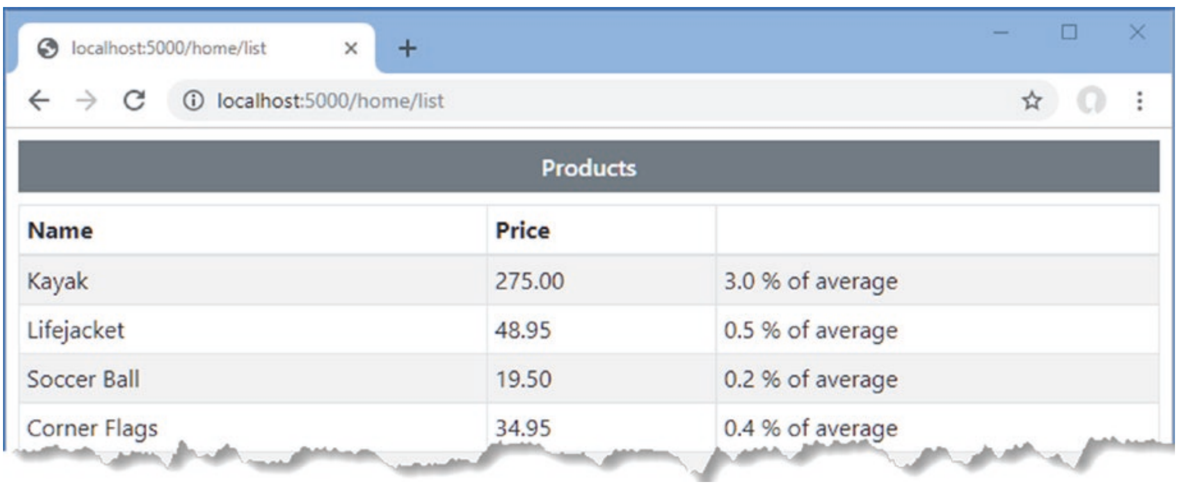

Figure 21-17. *Using a code block*

Summary

In this chapter, I introduced Razor views, which are used to create HTML responses from action methods. I explained how views are defined, how they are transformed into C# classes, and how the expressions they contain can be used to generate dynamic content. In the next chapter, I continue to describe how controllers can be used with views.

CHAPTER 22

Using Controllers with Views, Part II

In this chapter, I describe more of the features provided by Razor views. I show you how to pass additional data to a view using the view bang and how to use layouts and layout sections to reduce duplication. I also explain how the results from expressions are encoded and how to disable the encoding process. Table 22-1 summarizes the chapter.

Table 22-1. *Chapter Summary*

Problem	Solution	Listing
Providing unstructured data to a view	Use the view bag	5, 6
Providing temporary data to a view	Use temp data	7, 8
Using the same content in multiple views	Use a layout	9–12, 15–18
Selecting the default layout for views	Use a view start file	13, 14
Interleaving unique and common content	Use layout sections	19–24
Creating reusable sections of content	Use a partial view	25–29
Inserting HTML into a response using a Razor expression	Encode the HTML	30–32
Including JSON in a view	Use the JSON encoder	33

Preparing for This Chapter

This chapter uses the WebApp project from Chapter 21. To prepare for this chapter, replace the contents of the `HomeController.cs` file with the code shown in Listing 22-1.

Listing 22-1. The Contents of the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;
using WebApp.Models;

namespace WebApp.Controllers {

    public class HomeController: Controller {
        private DataContext context;

        public HomeController(DataContext ctx) {
            context = ctx;
        }

        public async Task<IActionResult> Index(long id = 1) {
            return View(await context.Products.FindAsync(id));
        }
```


A. Freeman, *Pro ASP.NET Core 3*, https://doi.org/10.1007/978-1-4842-5440-0_22

```
        public IActionResult List() {
            return View(context.Products);
        }
    }
}
```

One of the features used in this chapter requires the session feature, which was described in Chapter 16. To enable sessions, add the statements shown in Listing 22-2 to the Startup class.

Listing 22-2. Enabling Sessions in the Startup.cs File in the WebApp Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using WebApp.Models;

namespace WebApp {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        public IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<DataContext>(opts => {
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:ProductConnection"]);
                opts.EnableSensitiveDataLogging(true);
            });
            services.AddControllersWithViews().AddRazorRuntimeCompilation();

            services.AddDistributedMemoryCache();
            services.AddSession(options => {
                options.Cookie.IsEssential = true;
            });
        }

        public void Configure(IApplicationBuilder app, DataContext context) {
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseSession();
            app.UseRouting();
            app.UseEndpoints(endpoints => {
                endpoints.MapControllers();
                endpoints.MapDefaultControllerRoute();
            });
            SeedData.SeedDatabase(context);
        }
    }
}
```

Dropping the Database

Open a new PowerShell command prompt, navigate to the folder that contains the `WebApp.csproj` file, and run the command shown in Listing 22-3 to drop the database.

■ **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-3`. See Chapter 1 for how to get help if you have problems running the examples.

Listing 22-3. Dropping the Database

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

Running the Example Application

Once the database has been dropped, select Start Without Debugging or Run Without Debugging from the Debug menu, or use the PowerShell command prompt to run the command shown in Listing 22-4.

Listing 22-4. Running the Example Application

```
dotnet run
```

The database will be seeded as part of the application startup. Once ASP.NET Core is running, use a web browser to request `http://localhost:5000`, which will produce the response shown in Figure 22-1.

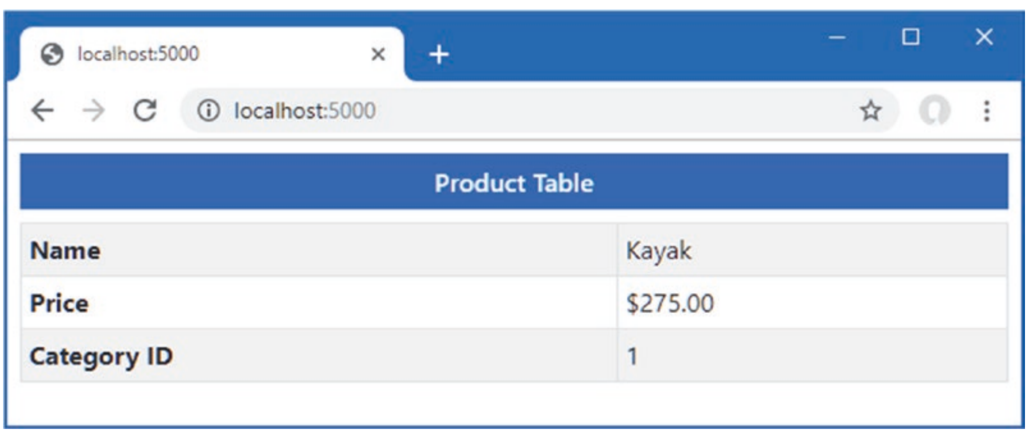

Figure 22-1. *Running the example application*

Using the View Bag

Action methods provide views with data to display with a view model, but sometimes additional information is required. Action methods can use the *view bag* to provide a view with extra data, as shown in Listing 22-5.

Listing 22-5. Using the View Bag in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;
using WebApp.Models;
using Microsoft.EntityFrameworkCore;

namespace WebApp.Controllers {

    public class HomeController: Controller {
        private DataContext context;

        public HomeController(DataContext ctx) {
            context = ctx;
        }

        public async Task<IActionResult> Index(long id = 1) {
            ViewBag.AveragePrice = await context.Products.AverageAsync(p => p.Price);
            return View(await context.Products.FindAsync(id));
        }

        public IActionResult List() {
            return View(context.Products);
        }
    }
}
```

The `ViewBag` property is inherited from the `Controller` base class and returns a `dynamic` object. This allows action methods to create new properties just by assigning values to them, as shown in the listing. The values assigned to the `ViewBag` property by the action method are available to the view through a property also called `ViewBag`, as shown in Listing 22-6.

Listing 22-6. Using the View Bag in the Index.cshtml File in the Views/Home Folder

```
<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <h6 class="bg-primary text-white text-center m-2 p-2">Product Table</h6>
    <div class="m-2">
        <table class="table table-sm table-striped table-bordered">
            <tbody>
                <tr><th>Name</th><td>@Model.Name</td></tr>
                <tr>
                    <th>Price</th>
                    <td>
                        @Model.Price.ToString("c")
                        (@(((Model.Price / ViewBag.AveragePrice)
                            * 100).ToString("F2"))% of average price)
                    </td>
                </tr>
                <tr><th>Category ID</th><td>@Model.CategoryId</td></tr>
            </tbody>
        </table>
    </div>
</body>
</html>
```

The ViewBag property conveys the object from the action to the view, alongside the view model object. In the listing, the action method queries for the average of the Product.Price properties in the database and assigns it to a view bag property named AveragePrice, which the view uses in an expression. Restart ASP.NET Core and use a browser to request http://localhost:5000, which produces the response shown in Figure 22-2.

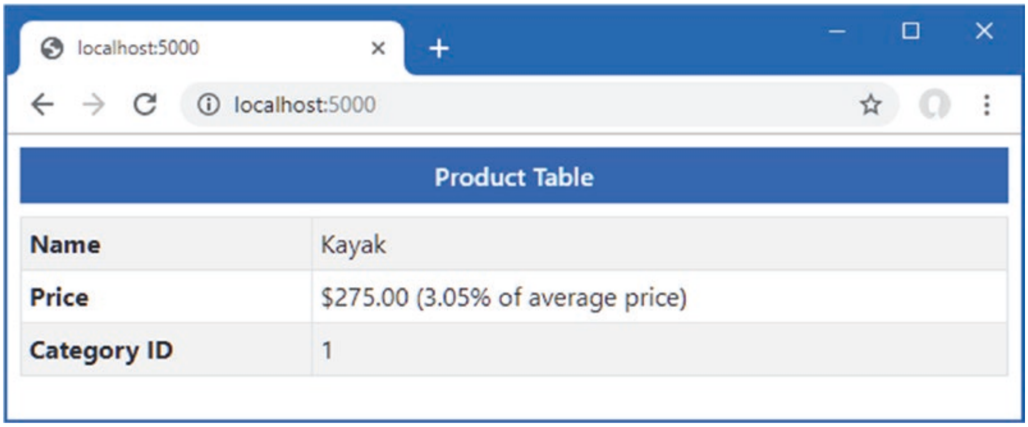

Figure 22-2. *Using the view bag*

WHEN TO USE THE VIEW BAG

The view bag works best when it is used to provide the view with small amounts of supplementary data without having to create new view model classes for each action method. The problem with the view bag is that the compiler cannot check the use of the properties on dynamic objects, much like views that don't use an @model expression. It can be difficult to judge when a new view model class should be used, and my rule of thumb is to create a new view model class when the same view model property is used by multiple actions or when an action method adds more than two or three properties to the view bag.

Using Temp Data

The temp data feature allows a controller to preserve data from one request to another, which is useful when performing redirections. Temp data is stored using a cookie unless session state is enabled when it is stored as session data. Unlike session data, temp data values are marked for deletion when they are read and removed when the request has been processed.

Add a class file called CubedController.cs to the WebApp/Controllers folder and use it to define the controller shown in Listing 22-7.

Listing 22-7. The Contents of the CubedController.cs File in the Controllers Folder

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

namespace WebApp.Controllers {
    public class CubedController: Controller {

        public IActionResult Index() {
            return View("Cubed");
        }

        public IActionResult Cube(double num) {
            TempData["value"] = num.ToString();
            TempData["result"] = Math.Pow(num, 3).ToString();
```

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

The Cubed controller defines an Index method that selects a view named Cubed. There is also a Cube action, which relies on the model binding process to obtain a value for its num parameter from the request (a process described in detail in Chapter 28). The Cubed action method performs its calculation and stores the num value and the calculation result using TempData property, which returns a dictionary that is used to store key/value pairs. Since the temp data feature is built on top of the sessions feature, only values that can be serialized to strings can be stored, which is why I convert both double values to strings in Listing 22-7. Once the values are stored as temp data, the Cube method performs a redirection to the Index method. To provide the controller with a view, add a Razor view file named Cubed.cshtml to the WebApp/Views/Shared folder with the content shown in Listing 22-8.

Listing 22-8. The Contents of the Cubed.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <h6 class="bg-secondary text-white text-center m-2 p-2">Cubed</h6>
    <form method="get" action="/cubed/cube" class="m-2">
        <div class="form-group">
            <label>Value</label>
            <input name="num" class="form-control" value="@(TempData["value"])" />
        </div>
        <button class="btn btn-primary" type="submit">Submit</button>
    </form>
    @if (TempData["result"] != null) {
        <div class="bg-info text-white m-2 p-2">
            The cube of @TempData["value"] is @TempData["result"]
        </div>
    }
</body>
</html>
```

The base class used for Razor views provides access to the temp data through a TempData property, allowing values to be read within expressions. In this case, temp data is used to set the content of an input element and display a results summary. Reading a temp data value doesn't remove it immediately, which means that values can be read repeatedly in the same view. It is only once the request has been processed that the marked values are removed.

To see the effect, restart ASP.NET Core, use a browser to navigate to http://localhost:5000/cubed, enter a value into the form field, and click the Submit button. The browser will send a request that will set the temp data and trigger the redirection. The temp data values are preserved for the new request, and the results are displayed to the user. But reading the data values marks them for deletion, and if you reload the browser, the contents of the input element and the results summary are no longer displayed, as shown in Figure 22-3.

■ **Tip** The object returned by the TempData property provides a Peek method, which allows you to get a data value without marking it for deletion, and a Keep method, which can be used to prevent a previously read value from being deleted. The Keep method doesn't protect a value forever. If the value is read again, it will be marked for removal once more. Use session data if you want to store items so that they won't be removed when the request is processed.

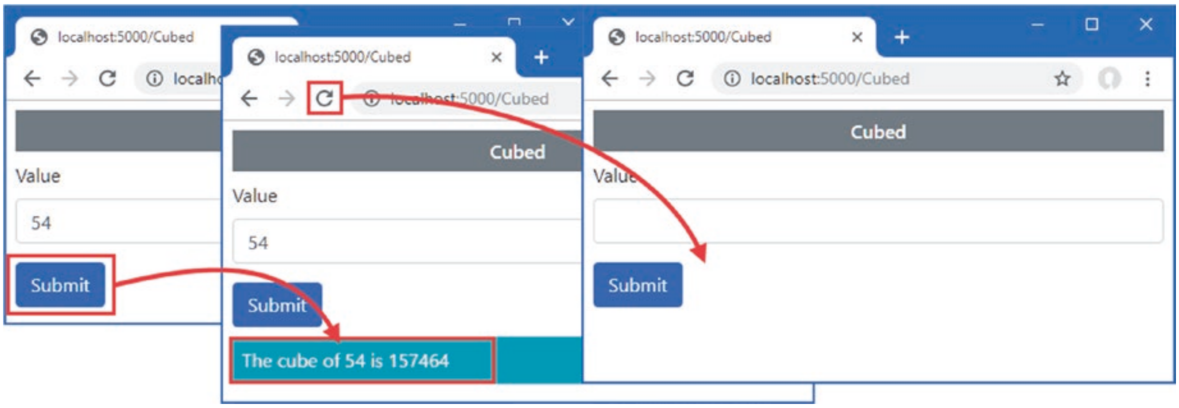

Figure 22-3. Using temp data

USING THE TEMP DATA ATTRIBUTE

Controllers can define properties that are decorated with the `TempData` attribute, which is an alternative to using the `TempData` property, like this:

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

namespace WebApp.Controllers {
    public class CubedController: Controller {

        public IActionResult Index() {
            return View("Cubed");
        }

        public IActionResult Cube(double num) {
            Value = num.ToString();
            Result = Math.Pow(num, 3).ToString();
            return RedirectToAction(nameof(Index));
        }

        [TempData]
        public string Value { get; set; }

        [TempData]
        public string Result { get; set; }
    }
}
```

The values assigned to these properties are automatically added to the temp data store, and there is no difference in the way they are accessed in the view. My preference is to use the `TempData` dictionary to store values because it makes the intent of the action method obvious to other developers. However, both approaches are entirely valid, and choosing between them is a matter of preference.

Working with Layouts

The views in the example application contain duplicate elements that deal with setting up the HTML document, defining the head section, loading the Bootstrap CSS file, and so on. Razor supports *layouts*, which avoid this sort of duplication by consolidating common content in a single file that can be used by any view.

Layouts are typically stored in the Views/Shared folder because they are usually used by the action methods of more than one controller. If you are using Visual Studio, right-click the Views/Shared folder, select Add ➤ New Item from the popup menu, and choose the Razor Layout template, as shown in Figure 22-4. Make sure the name of the file is _Layout.cshtml and click the Add button to create the new file. Replace the content added to the file by Visual Studio with the elements shown in Listing 22-9.

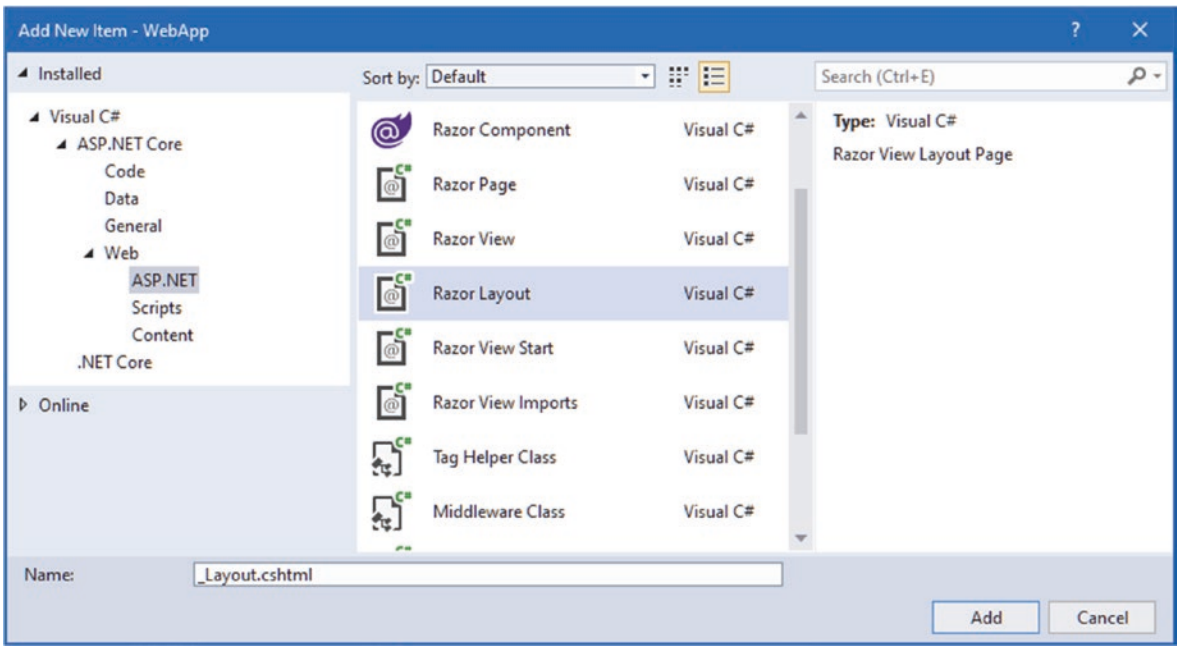

Figure 22-4. *Creating a layout*

If you are using Visual Studio Code, create a file named _Layout.cshtml in the Views/Shared folder and add the content shown in Listing 22-9.

Listing 22-9. The Contents of the _Layout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <h6 class="bg-primary text-white text-center m-2 p-2">Shared View</h6>
    @RenderBody()
</body>
</html>
```

The layout contains the common content that will be used by multiple views. The content that is unique to each view is inserted into the response by calling the RenderBody method, which is inherited by the RazorPage<T> class, as described in Chapter 21. Views that use layouts can focus on just their unique content, as shown in Listing 22-10.

Listing 22-10. Using a Layout in the Index.cshtml File in the Views/Home Folder

```
@model Product
@{
    Layout = "_Layout";
}
<div class="m-2">
    <table class="table table-sm table-striped table-bordered">
        <tbody>
            <tr><th>Name</th><td>@Model.Name</td></tr>
            <tr>
                <th>Price</th>
                <td>
                    @Model.Price.ToString("c")
                    (@(((Model.Price / ViewBag.AveragePrice)
                        * 100).ToString("F2"))% of average price)
                </td>
            </tr>
            <tr><th>Category ID</th><td>@Model.CategoryId</td></tr>
        </tbody>
    </table>
</div>
```

The layout is selected by adding a code block, denoted by the `@{` and `}` characters, that sets the `Layout` property inherited from the `RazorPage<T>` class. In this case, the `Layout` property is set to the name of the layout file. As with normal views, the layout is specified without a path or file extension, and the Razor engine will search in the `/Views/[controller]` and `/Views/Shared` folders to find a matching file. Use the browser to request `http://localhost:5000`, and you will see the response shown in Figure 22-5.

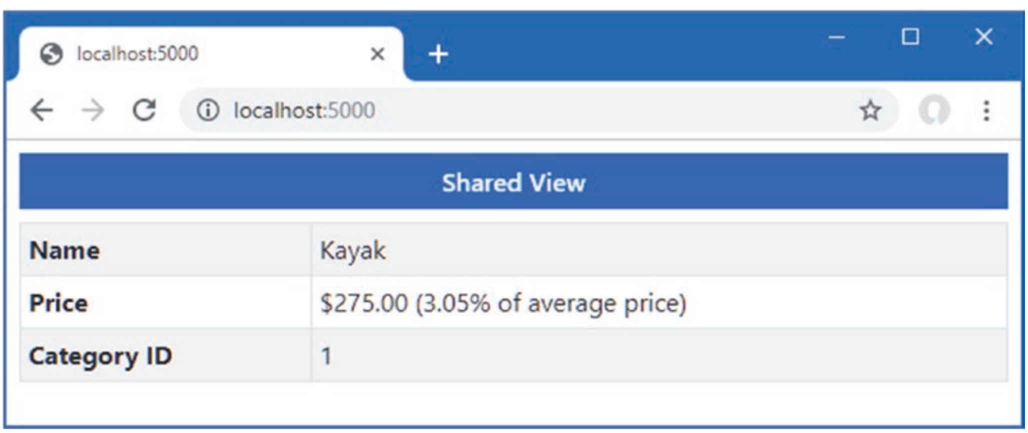

Figure 22-5. *Using a layout*

Configuring Layouts Using the View Bag

The view can provide the layout with data values, allowing the common content provided by the view to be customized. The view bag properties are defined in the code block that selects the layout, as shown in Listing 22-11.

Listing 22-11. Setting a View Bag Property in the Index.cshtml File in the Views/Home Folder

```
@model Product
@{
    Layout = "_Layout";
    ViewBag.Title = "Product Table";
}
```

```
<div class="m-2">
    <table class="table table-sm table-striped table-bordered">
        <tbody>
            <tr><th>Name</th><td>@Model.Name</td></tr>
            <tr>
                <th>Price</th>
                <td>
                    @Model.Price.ToString("c")
                    (@(((Model.Price / ViewBag.AveragePrice)
                        * 100).ToString("F2"))% of average price)
                </td>
            </tr>
            <tr><th>Category ID</th><td>@Model.CategoryId</td></tr>
        </tbody>
    </table>
</div>
```

The view sets a `Title` property, which can be used in the layout, as shown in Listing 22-12.

Listing 22-12. Using a View Bag Property in the _Layout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <h6 class="bg-primary text-white text-center m-2 p-2">
        @(ViewBag.Title ?? "Layout")
    </h6>
    @RenderBody()
</body>
</html>
```

The `Title` property is used to set the content of the `title` element and h6 element in the body section. Layouts cannot rely on view bag properties being defined, which is why the expression in the h6 element provides a fallback value if the view doesn't define a `Title` property. To see the effect of the view bag property, use a browser to request `http://localhost:5000`, which produces the response shown in Figure 22-6.

UNDERSTANDING VIEW BAG PRECEDENCE

The values defined by the view take precedence if the same view bag property is defined by the view and the action method. If you want to allow the action to override the value defined in the view, then use a statement like this in the view code block:

```
...
@{
    Layout = "_Layout";
    ViewBag.Title = ViewBag.Title ?? "Product Table";
}
...
```

This statement will set the value for the `Title` property only if it has not already been defined by the action method.

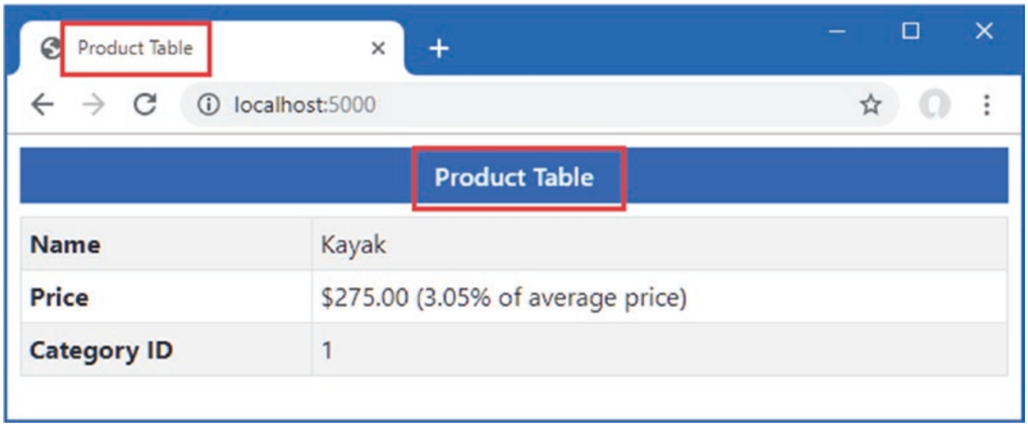

Figure 22-6. *Using a view bag property to configure a layout*

Using a View Start File

Instead of setting the `Layout` property in every view, you can add a *view start* file to the project that provides a default `Layout` value. If you are using Visual Studio, right-click the `Views` folder item in the Solution Explorer, select Add ➤ New Item, and locate the Razor View Start template, as shown in Figure 22-7. Make sure the name of the file is `_ViewStart.cshtml` and click the Add button to create the file, which will have the content shown in Listing 22-13.

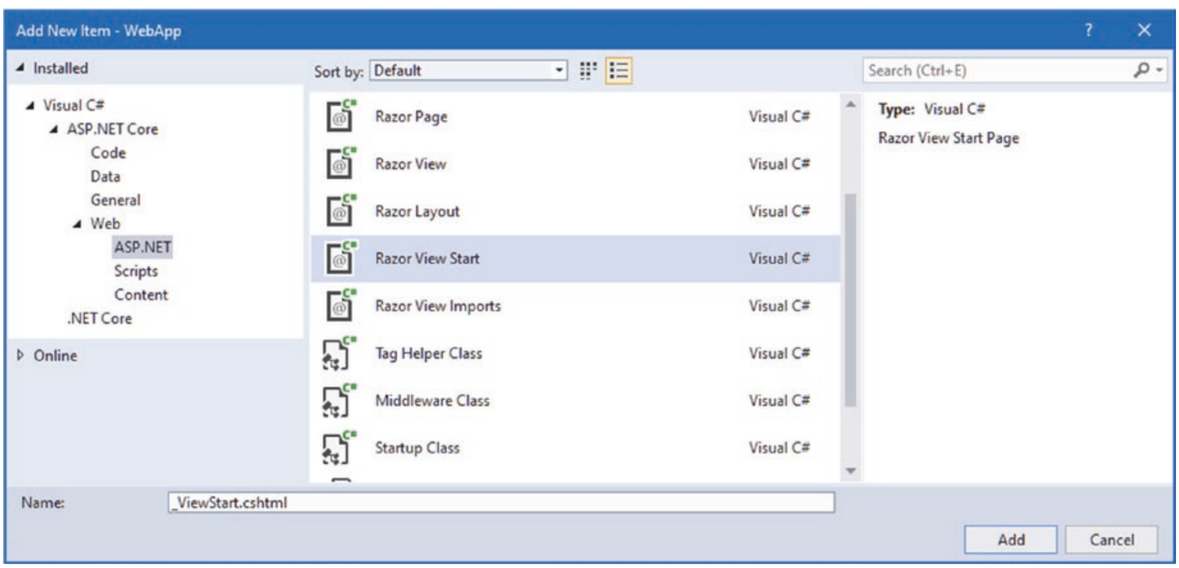

Figure 22-7. *Creating a view start file*

If you are using Visual Studio Code, then add a file named `_ViewStart.cshtml` to the `Views` folder and add the content shown in Listing 22-13.

Listing 22-13. The Contents of the _ViewStart.cshtml File in the Views Folder

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

The file contains sets the `Layout` property, and the value will be used as the default. Listing 22-14 removes the content from the `Common.cshtml` file that is contained in the layout.

Listing 22-14. Removing Content in the Common.cshtml File in the Views/Shared Folder

```
<h6 class="bg-secondary text-white text-center m-2 p-2">Shared View</h6>
```

The view doesn't define a view model type and doesn't need to set the `Layout` property because the project contains a view start file. The result is that the content in Listing 22-14 will be added to the body section of the HTML content of the response. Use a browser to navigate to `http://localhost:5000/second`, and you will see the response in Figure 22-8.

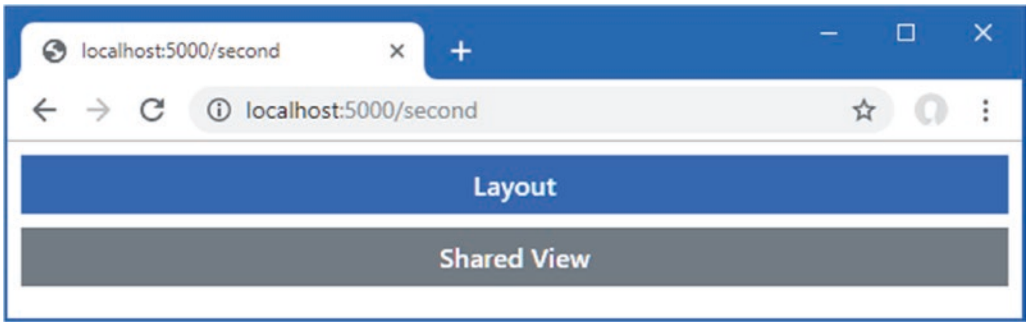

Figure 22-8. *Using a view start file*

Overriding the Default Layout

There are two situations where you may need to define a `Layout` property in a view even when there is a view start file in the project. In the first situation, a view requires a different layout from the one specified by the view start file. To demonstrate, add a Razor layout file named `_ImportantLayout.cshtml` to the `Views/Shared` folder with the content shown in Listing 22-15.

Listing 22-15. The Contents of the _ImportantLayout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <h3 class="bg-warning text-white text-center p-2 m-2">Important</h3>
    @RenderBody()
</body>
</html>
```

In addition to the HTML document structure, this file contains a header element that displays `Important` in large text. Views can select this layout by assigning its name to the `Layout` property, as shown in Listing 22-16.

■ **Tip** If you need to use a different layout for all the actions of a single controller, then add a view start file to the `Views/[controller]` folder that selects the view you require. The Razor engine will use the layout specified by the controller-specific view start file.

Listing 22-16. Using a Specific Layout in the Index.cshtml File in the Views/Home Folder

```
@model Product
@{
    Layout = "_ImportantLayout";
    ViewBag.Title = ViewBag.Title ?? "Product Table";
}
<div class="m-2">
    <table class="table table-sm table-striped table-bordered">
        <tbody>
            <tr><th>Name</th><td>@Model.Name</td></tr>
            <tr>
                <th>Price</th>
                <td>
                    @Model.Price.ToString("c")
                    (@(((Model.Price / ViewBag.AveragePrice)
                        * 100).ToString("F2"))% of average price)
                </td>
            </tr>
            <tr><th>Category ID</th><td>@Model.CategoryId</td></tr>
        </tbody>
    </table>
</div>
```

The Layout value in the view start file is overridden by the value in the view, allowing different layouts to be applied. Use a browse to request http://localhost:5000, and the response will be produced using the new layout, as shown in Figure 22-9.

SELECTING A LAYOUT PROGRAMMATICALLY

The value that a view assigns to the Layout property can be the result of an expression that allows layouts to be selected by the view, similar to the way that action methods can select views. Here is an example that selects the layout based on a property defined by the view model object:

```
...
@model Product
@{
    Layout = Model.Price > 100 ? "_ImportantLayout" : "_Layout";
    ViewBag.Title = ViewBag.Title ?? "Product Table";
}
...
```

The layout named _ImportantLayout is selected when the value of the view model object's Price property is greater than 100; otherwise, _Layout is used.

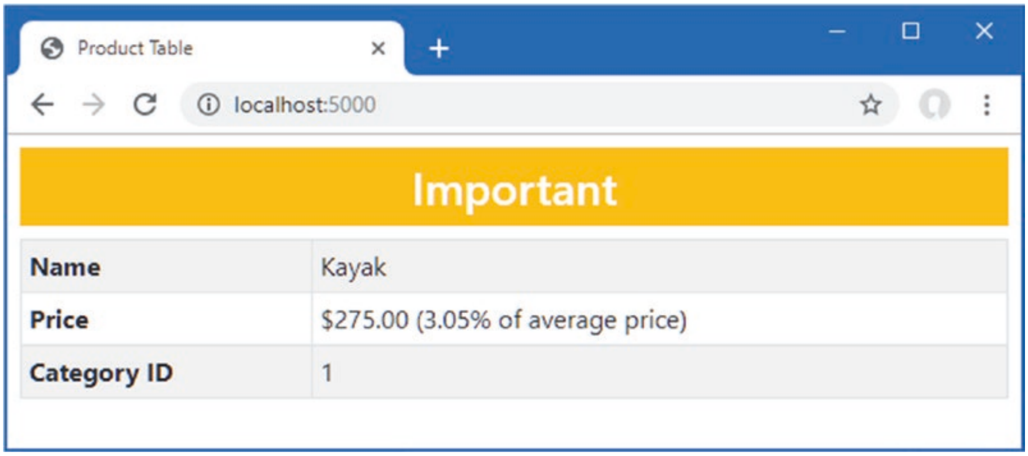

Figure 22-9. *Specifying a layout in a view*

The second situation where a Layout property can be needed is when a view contains a complete HTML document and doesn't require a layout at all. To see the problem, open a new PowerShell command prompt and run the command shown in Listing 22-17.

Listing 22-17. Sending an HTTP Request

```
Invoke-WebRequest http://localhost:5000/home/list | Select-Object  -expand Content
```

This command sends an HTTP GET request whose response will be produced using the `List.cshtml` file in the `Views/Home` folder. This view contains a complete HTML document, which is combined with the content in the view specified by the view start file, producing a malformed HTML document, like this:

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <h6 class="bg-primary text-white text-center m-2 p-2">
        Layout
    </h6>
    <!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <h6 class="bg-secondary text-white text-center m-2 p-2">Products</h6>
    <div class="m-2">
        <table class="table table-sm table-striped table-bordered">
            <thead>
                <tr><th>Name</th><th>Price</th></tr>
            </thead>
            <tbody>
                    <tr><td>Kayak</td><td>275.00</td></tr>
                    <tr><td>Lifejacket</td><td>48.95</td></tr>
                    <tr><td>Soccer Ball</td><td>19.50</td></tr>
                    <tr><td>Corner Flags</td><td>34.95</td></tr>
```

```
                <tr><td>Stadium</td><td>79500.00</td></tr>
                <tr><td>Thinking Cap</td><td>16.00</td></tr>
                <tr><td>Unsteady Chair</td><td>29.95</td></tr>
                <tr><td>Human Chess Board</td><td>75.00</td></tr>
                <tr><td>Bling-Bling King</td><td>1200.00</td></tr>
            </tbody>
        </table>
    </div>
</body>
</html>
</body>
</html>
```

The structural elements for the HTML document are duplicated, so there are two `html`, `head`, `body`, and `link` elements. Browsers are adept at handling malformed HTML but don't always cope with poorly structured content. Where a view contains a complete HTML document, the `Layout` property can be set to null, as shown in Listing 22-18.

Listing 22-18. Disabling Layouts in the List.cshtml File in the Views/Home Folder

```
@model IEnumerable<Product>
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <h6 class="bg-secondary text-white text-center m-2 p-2">Products</h6>
    <div class="m-2">
        <table class="table table-sm table-striped table-bordered">
            <thead>
                <tr><th>Name</th><th>Price</th></tr>
            </thead>
            <tbody>
                @foreach (Product p in Model) {
                    <tr><td>@p.Name</td><td>@p.Price</td></tr>
                }
            </tbody>
        </table>
    </div>
</body>
</html>
```

Save the view and run the command shown in Listing 22-17 again, and you will see that the response contains only the elements in the view and that the layout has been disabled.

```
<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <h6 class="bg-secondary text-white text-center m-2 p-2">Products</h6>
    <div class="m-2">
        <table class="table table-sm table-striped table-bordered">
```

```
            <thead>
                <tr><th>Name</th><th>Price</th></tr>
            </thead>
            <tbody>
                    <tr><td>Kayak</td><td>275.00</td></tr>
                    <tr><td>Lifejacket</td><td>48.95</td></tr>
                    <tr><td>Soccer Ball</td><td>19.50</td></tr>
                    <tr><td>Corner Flags</td><td>34.95</td></tr>
                    <tr><td>Stadium</td><td>79500.00</td></tr>
                    <tr><td>Thinking Cap</td><td>16.00</td></tr>
                    <tr><td>Unsteady Chair</td><td>29.95</td></tr>
                    <tr><td>Human Chess Board</td><td>75.00</td></tr>
                    <tr><td>Bling-Bling King</td><td>1200.00</td></tr>
            </tbody>
        </table>
    </div>
</body>
</html>
```

Using Layout Sections

The Razor view engine supports the concept of *sections*, which allow you to provide regions of content within a layout. Razor sections give greater control over which parts of the view are inserted into the layout and where they are placed. To demonstrate the sections feature, I have edited the `/Views/Home/Index.cshtml` file, as shown in Listing 22-19.

Listing 22-19. Defining Sections in the Index.cshtml File in the Views/Home Folder

```
@model Product
@{
    Layout = "_Layout";
    ViewBag.Title = ViewBag.Title ?? "Product Table";
}

@section Header {
    Product Information
}

<tr><th>Name</th><td>@Model.Name</td></tr>
<tr>
    <th>Price</th>
    <td>@Model.Price.ToString("c")</td>
</tr>
<tr><th>Category ID</th><td>@Model.CategoryId</td></tr>

@section Footer {
    @(((Model.Price / ViewBag.AveragePrice)
        * 100).ToString("F2"))% of average price
}
```

Sections are defined using the Razor `@section` expression followed by a name for the section. Listing 22-19 defines sections named `Header` and `Footer`, and sections can contain the same mix of HTML content and expressions, just like the main part of the view. Sections are applied in a layout with the `@RenderSection` expression, as shown in Listing 22-20.

Listing 22-20. Using Sections in the _Layout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="bg-info text-white m-2 p-1">
        This is part of the layout
    </div>

    <h6 class="bg-primary text-white text-center m-2 p-2">
        @RenderSection("Header")
    </h6>

    <div class="bg-info text-white m-2 p-1">
        This is part of the layout
    </div>

    <div class="m-2">
        <table class="table table-sm table-striped table-bordered">
            <tbody>
                @RenderBody()
            </tbody>
        </table>
    </div>

    <div class="bg-info text-white m-2 p-1">
        This is part of the layout
    </div>

    <h6 class="bg-primary text-white text-center m-2 p-2">
        @RenderSection("Footer")
    </h6>

    <div class="bg-info text-white m-2 p-1">
        This is part of the layout
    </div>
</body>
</html>
```

When the layout is applied, the `RenderSection` expression inserts the content of the specified section into the response. The regions of the view that are not contained within a section are inserted into the response by the `RenderBody` method. To see how the sections are applied, use a browser to request `http://localhost:5000`, which provides the response shown in Figure 22-10.

■ **Note** A view can define only the sections that are referred to in the layout. The view engine throws an exception if you define sections in the view for which there is no corresponding `@RenderSection` expression in the layout.

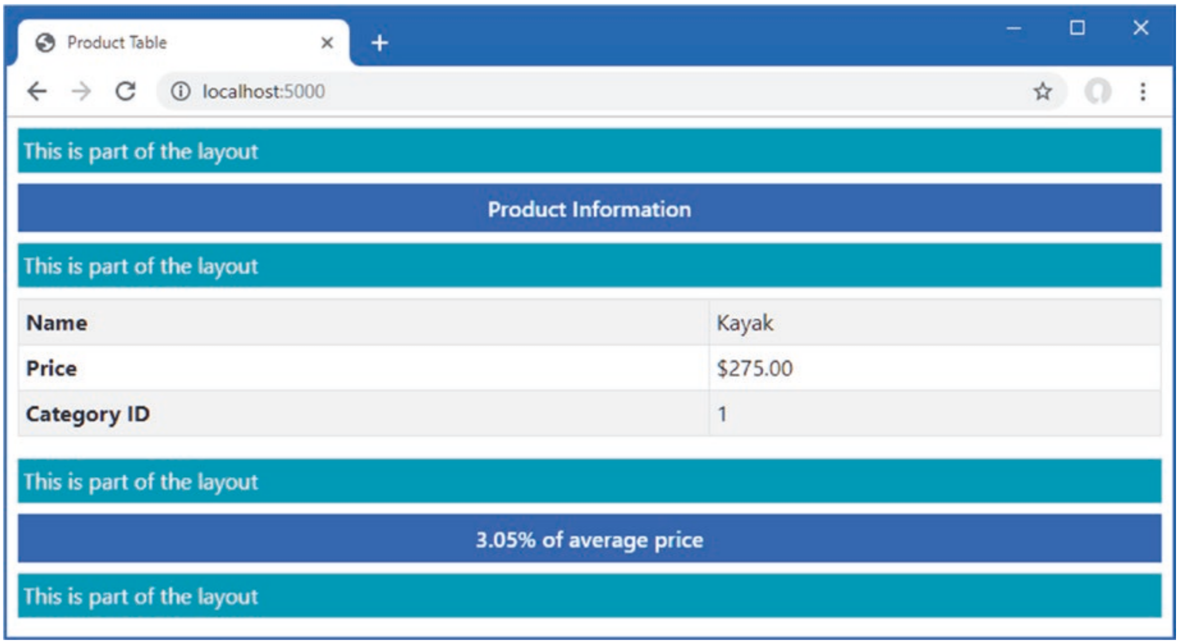

Figure 22-10. *Using sections in a layout*

Sections allow views to provide fragments of content to the layout without specifying how they are used. As an example, Listing 22-21 redefines the layout to consolidate the body and sections into a single HTML table.

Listing 22-21. Using a Table in the _Layout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="m-2">
        <table class="table table-sm table-striped table-bordered">
            <thead>
                <tr>
                    <th class="bg-primary text-white text-center" colspan="2">
                        @RenderSection("Header")
                    </th>
                </tr>
            </thead>
            <tbody>
                @RenderBody()
            </tbody>
            <tfoot>
                <tr>
                    <th class="bg-primary text-white text-center" colspan="2">
                        @RenderSection("Footer")
                    </th>
                </tr>
            </tfoot>
```

```
        </table>
    </div>
</body>
</html>
```

To see the effect of the change to the view, use a browser to request `http://localhost:5000`, which will produce the response shown in Figure 22-11.

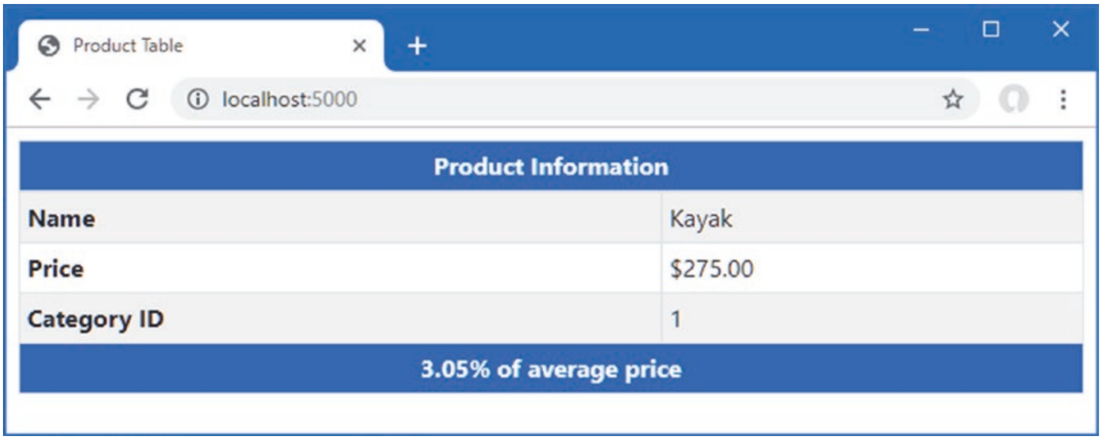

Figure 22-11. *Changing how sections are displayed in a layout*

Using Optional Layout Sections

By default, a view must contain all the sections for which there are `RenderSection` calls in the layout, and an exception will be thrown if the layout requires a section that the view hasn't defined. Listing 22-22 adds a call to the `RenderSection` method that requires a section named `Summary`.

Listing 22-22. Adding a Section in the _Layout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="m-2">
        <table class="table table-sm table-striped table-bordered">
            <thead>
                <tr>
                    <th class="bg-primary text-white text-center" colspan="2">
                        @RenderSection("Header")
                    </th>
                </tr>
            </thead>
            <tbody>
                @RenderBody()
            </tbody>
```

```
            <tfoot>
                <tr>
                    <th class="bg-primary text-white text-center" colspan="2">
                        @RenderSection("Footer")
                    </th>
                </tr>
            </tfoot>
        </table>
    </div>
    @RenderSection("Summary")
</body>
</html>
```

Use a browser to request `http://localhost:5000`, and you will see the exception shown in Figure 22-12.

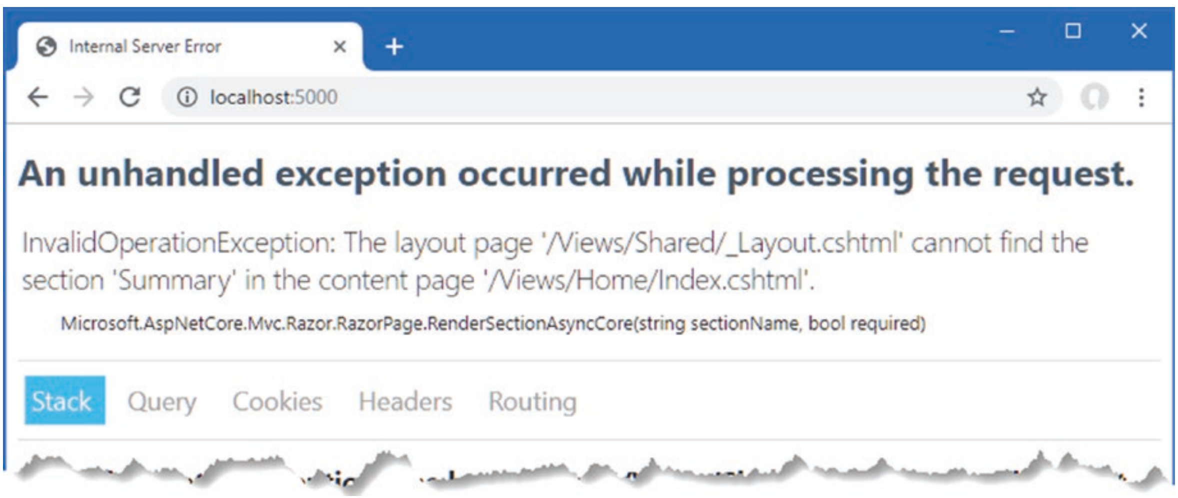

Figure 22-12. *Attempting to render a nonexistent view section*

There are two ways to solve this problem. The first is to create an optional section, which will be rendered only if it is defined by the view. Optional sections are created by passing a second argument to the `RenderSection` method, as shown in Listing 22-23.

Listing 22-23. Defining an Optional Section in the _Layout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="m-2">
        <table class="table table-sm table-striped table-bordered">
            <thead>
                <tr>
                    <th class="bg-primary text-white text-center" colspan="2">
                        @RenderSection("Header", false)
                    </th>
                </tr>
            </thead>
```

```
            <tbody>
                @RenderBody()
            </tbody>
            <tfoot>
                <tr>
                    <th class="bg-primary text-white text-center" colspan="2">
                        @RenderSection("Footer", false)
                    </th>
                </tr>
            </tfoot>
        </table>
    </div>
    @RenderSection("Summary", false)
</body>
</html>
```

The second argument specifies whether a section is required, and using `false` prevents an exception when the view doesn't define the section.

Testing for Layout Sections

The `IsSectionDefined` method is used to determine whether a view defines a specified section and can be used in an `if` expression to render fallback content, as shown in Listing 22-24.

Listing 22-24. Checking for a Section in the _Layout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="m-2">
        <table class="table table-sm table-striped table-bordered">
            <thead>
                <tr>
                    <th class="bg-primary text-white text-center" colspan="2">
                        @RenderSection("Header", false)
                    </th>
                </tr>
            </thead>
            <tbody>
                @RenderBody()
            </tbody>
            <tfoot>
                <tr>
                    <th class="bg-primary text-white text-center" colspan="2">
                        @RenderSection("Footer", false)
                    </th>
                </tr>
            </tfoot>
        </table>
    </div>
```

```
    @if (IsSectionDefined("Summary")) {
        @RenderSection("Summary", false)
    } else {
        <div class="bg-info text-center text-white m-2 p-2">
            This is the default summary
        </div>
    }
</body>
</html>
```

The IsSectionDefined method is invoked with the name of the section you want to check and returns true if the view defines that section. In the example, I used this helper to render fallback content when the view does not define the Summary section. To see the fallback content, use a browser to request http://localhost:5000, which produces the response shown in Figure 22-13.

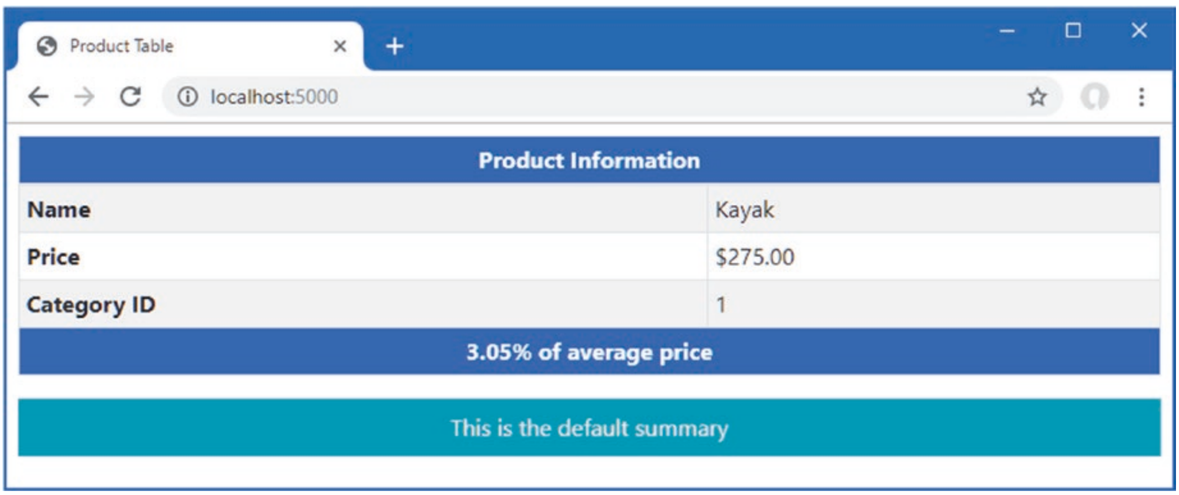

Figure 22-13. *Displaying fallback content for a view section*

Using Partial Views

You will often need to use the same set of HTML elements and expressions in several different places. *Partial views* are views that contain fragments of content that will be included in other views to produce complex responses without duplication.

Enabling Partial Views

Partial views are applied using a feature called *tag helpers*, which are described in detail in Chapter 25; tag helpers are configured in the view imports file, which was added to the project in Chapter 21. To enable the feature required for partial views, add the statement shown in Listing 22-25 to the _ViewImports.cshtml file.

Listing 22-25. Enabling Tag Helpers in the _ViewImports.cshtml File in the Views Folder

```
@using WebApp.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Creating a Partial View

Partial views are just regular CSHTML files, and it is only the way they are used that differentiates them from standard views. If you are using Visual Studio, right-click the Views/Home folder, select Add ➤ New Item, and use the Razor View template to create a file named _RowPartial.cshtml. Once the file has been created, replace the contents with those shown in Listing 22-26. If you are using Visual Studio Code, add a file named _RowPartial.cshtml to the Views/Home folder and add to it the content shown in Listing 22-26.

■ **Tip** Visual Studio provides some tooling support for creating prepopulated partial views, but the simplest way to create a partial view is to create a regular view using the Razor View item template.

Listing 22-26. The Contents of the _RowPartial.cshtml File in the Views/Home Folder

```
@model Product

<tr>
    <td>@Model.Name</td>
    <td>@Model.Price</td>
</tr>
```

The `model` expression is used to define the view model type for the partial view, which contains the same mix of expressions and HTML elements as regular views. The content of this partial view creates a table row, using the `Name` and `Price` properties of a `Product` object to populate the table cells.

Applying a Partial View

Partial views are applied by adding a `partial` element in another view or layout. In Listing 22-27, I have added the element to the `List.cshtml` file so the partial view is used to generate the rows in the table.

Listing 22-27. Using a Partial View in the List.cshtml File in the Views/Home Folder

```
@model IEnumerable<Product>
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <h6 class="bg-secondary text-white text-center m-2 p-2">Products</h6>
    <div class="m-2">
        <table class="table table-sm table-striped table-bordered">
            <thead>
                <tr><th>Name</th><th>Price</th></tr>
            </thead>
            <tbody>
                @foreach (Product p in Model) {
                    <partial name="_RowPartial" model="p" />
                }
            </tbody>
        </table>
    </div>
</body>
</html>
```

The attributes applied to the `partial` element control the selection and configuration of the partial view, as described in Table 22-2.

Table 22-2. The partial Element Attributes

Name	Description
`name`	This property specifies the name of the partial view, which is located using the same search process as regular views.
`model`	This property specifies the value that will be used as the view model object for the partial view.
`for`	This property is used to define an expression that selects the view model object for the partial view, as explained next.
`view-data`	This property is used to provide the partial view with additional data.

The `partial` element in Listing 22-27 uses the `name` attribute to select the `_RowPartial` view and the `model` attribute to select the `Product` object that will be used as the view model object. The `partial` element is applied within the `@foreach` expression, which means that it will be used to generate each row in the table, which you can see by using a browser to request `http://localhost:5000/home/list` to produce the response shown in Figure 22-14.

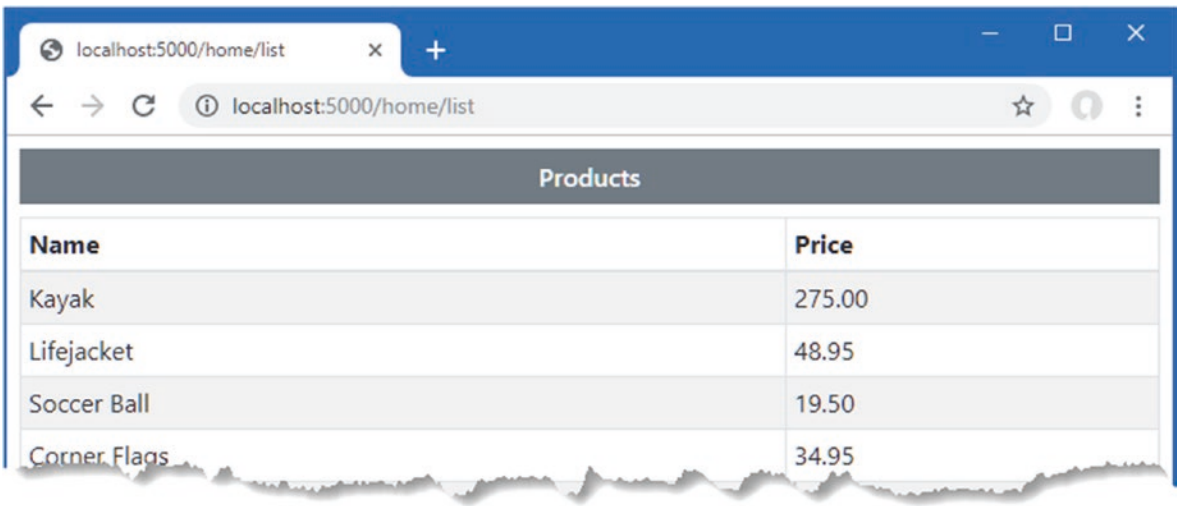

***Figure 22-14.** Using a partial view*

USING THE HTML HELPER TO APPLY PARTIAL VIEWS

In earlier versions of ASP.NET Core, partial views were applied using the `Html` property that is added to the C# class generated from the view, as explained in Chapter 21. The object returned by the `Html` property implements the `IHtmlHelper` interface, through which views can be applied, like this:

```
...
@Html.Partial("_RowPartial")
...
```

This type of expression works and is still supported, but the `partial` element provides a more elegant approach that is consistent with the rest of the HTML elements in the view.

Selecting the Partial View Model Using an Expression

The `for` attribute is used to set the partial view's model using an expression that is applied to the view's model, which is a feature more easily demonstrated than described. Add a partial view named `_CellPartial.cshtml` to the `Views/Home` folder with the content shown in Listing 22-28.

Listing 22-28. The Contents of the _CellPartial.cshtml File in the Views/Home Folder

```
@model string

<td class="bg-info text-white">@Model</td>
```

This partial view has a string view model object, which it uses as the contents of a table cell element; the table cell element is styled using the Bootstrap CSS framework. In Listing 22-29, I have added a `partial` element to the `_RowPartial.cshtml` file that uses the `_CellPartial` partial view to display the table cell for the name of the `Product` object.

Listing 22-29. Using a Partial View in the _RowPartial.cshtml File in the Views/Home Folder

```
@model Product

<tr>
    <partial name="_CellPartial" for="Name" />
    <td>@Model.Price</td>
</tr>
```

The `for` attribute selects the `Name` property as the model for the `_CellPartial` partial view. To see the effect, use a browser to request `http://localhost:5000/home/list`, which will produce the response shown in Figure 22-15.

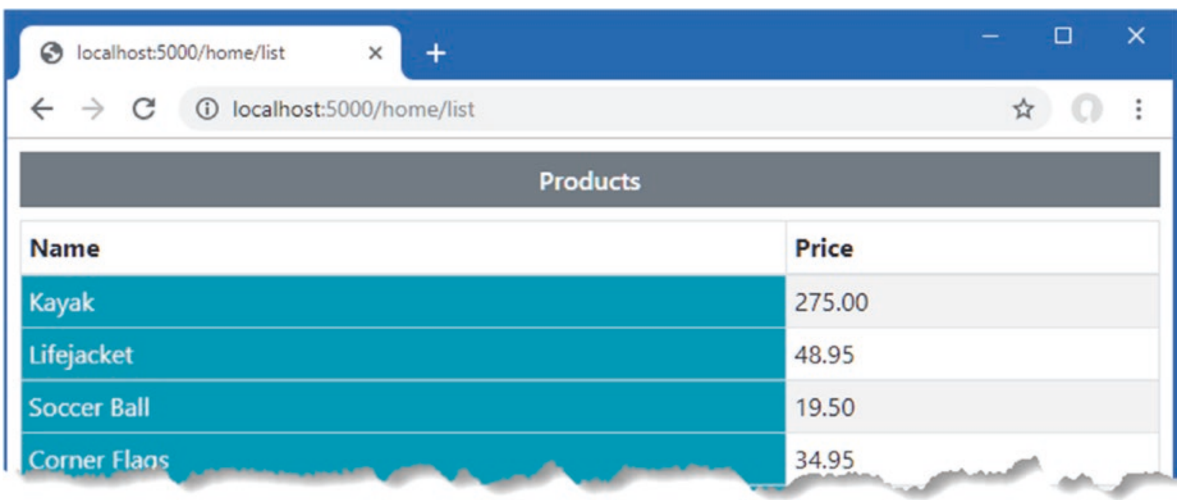

Figure 22-15. *Selecting a model property for use in a partial view*

USING TEMPLATED DELEGATES

Templated delegates are an alternative way of avoiding duplication in a view. Templated delegates are defined in a code block, like this:

```
...
@{
    Func<Product, object> row
        = @<tr><td>@item.Name</td><td>@item.Price</td></tr>;
}
...
```

The template is a function that accepts a `Product` input object and returns a dynamic result. Within the template expression, the input object is referred to as `item` in expressions. The templated delegate is invoked as a method expression to generate content.

```
...
<tbody>
    @foreach (Product p in Model) {
        @row(p)
    }
</tbody>
...
```

I find this feature awkward and prefer using partial views, although this is a matter of preference and habit rather than any objective problems with the way that templated delegates work.

Understanding Content-Encoding

Razor views provide two useful features for encoding content. The HTML content-encoding feature ensures that expression responses don't change the structure of the response sent to the browser, which is an important security feature. The JSON encoding feature encodes an object as JSON and inserts it into the response, which can be a useful debugging feature and can also be useful when providing data to JavaScript applications. Both encoding features are described in the following sections.

Understanding HTML Encoding

The Razor view engine encodes expression results to make them safe to include in an HTML document without changing its structure. This is an important feature when dealing with content that is provided by users, who may try to subvert the application or accidentally enter dangerous content. Listing 22-30 adds an action method to the Home controller that passes a fragment of HTML to the View method.

Listing 22-30. Adding an Action in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;
using WebApp.Models;
using Microsoft.EntityFrameworkCore;

namespace WebApp.Controllers {

    public class HomeController: Controller {
        private DataContext context;

        public HomeController(DataContext ctx) {
            context = ctx;
        }

        public async Task<IActionResult> Index(long id = 1) {
            ViewBag.AveragePrice = await context.Products.AverageAsync(p => p.Price);
            return View(await context.Products.FindAsync(id));
        }

        public IActionResult List() {
            return View(context.Products);
        }

        public IActionResult Html() {
            return View((object)"This is a <h3><i>string</i></h3>");
        }
    }
}
```

The new action passes a string that contains HTML elements. To create the view for the new action method, add a Razor view file named `Html.cshtml` to the `Views/Home` folder with the content shown in Listing 22-31.

■ **Tip** Notice that I cast the string passed to the `View` method as an object, without which the string is assumed to be the name of a view and not the view model object.

Listing 22-31. The Contents of the Html.cshtml File in the Views/Home Folder

```
@model string
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="bg-secondary text-white text-center m-2 p-2">@Model</div>
</body>
</html>
```

Restart ASP.NET Core and use a browser to request `http://localhost:5000/home/html`. The response, which is shown on the left of Figure 22-17, shows how the potentially dangerous characters in the view model string have been escaped.

To include the result of an expression without safe encoding, you can invoke the `Html.Raw` method. The `Html` property is one of the properties added to the generated view class, described in Chapter 21, which returns an object that implements the `IHtmlHelper` interface, as shown in Listing 22-32.

Listing 22-32. Disabling Encoding in the Html.cshtml File in the Views/Home Folder

```
@model string
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="bg-secondary text-white text-center m-2 p-2">@Html.Raw(Model)</div>
</body>
</html>
```

Request the `http://localhost:5000/home/html` URL again, and you will see that the view model string is passed on without being encoded and is then interpreted by the browser as part of the HTML document, as shown on the right of Figure 22-16.

■ **Caution** Do not disable safe encoding unless you are entirely confident that no malicious content will be passed to the view. Careless use of this feature presents a security risk to your application and your users.

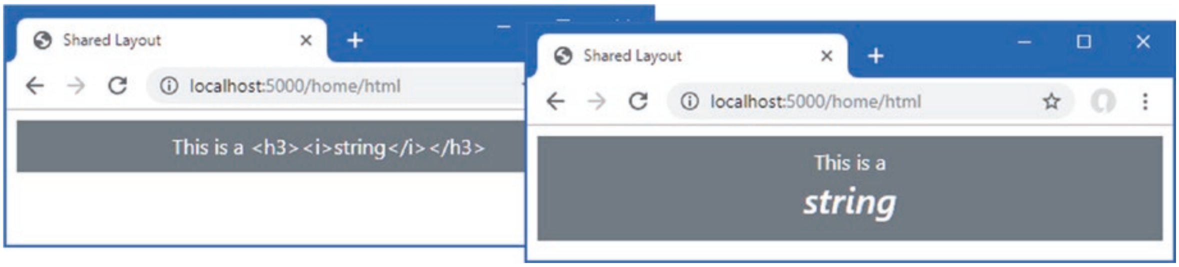

Figure 22-16. *HTML result encoding*

Understanding JSON Encoding

The Json property, which is added to the class generated from the view, as described in Chapter 21, can be used to encode an object as JSON. The most common use for JSON data is in RESTful web services, as described in earlier chapters, but I find the Razor JSON encoding feature useful as a debugging aid when I don't get the output I expect from a view. Listing 22-33 adds a JSON representation of the view model object to the output produced by the Index view.

Listing 22-33. Using JSON Encoding in the Index.cshtml File in the Views/Home Folder

```
@model Product
@{
    Layout = "_Layout";
    ViewBag.Title = ViewBag.Title ?? "Product Table";
}

@section Header {
    Product Information
}

<tr><th>Name</th><td>@Model.Name</td></tr>
<tr>
    <th>Price</th>
    <td>@Model.Price.ToString("c")</td>
</tr>
<tr><th>Category ID</th><td>@Model.CategoryId</td></tr>

@section Footer {
    @(((Model.Price / ViewBag.AveragePrice)
        * 100).ToString("F2"))% of average price
}

@section Summary {
    <div class="bg-info text-white m-2 p-2">
        @Json.Serialize(Model)
    </div>
}
```

The Json property returns an implementation of the IJsonHelper interface, whose Serialize method produces a JSON representation of an object. Use a browser to request http://localhost:5000, and you will see the response shown in Figure 22-17, which includes JSON in the Summary section of the view.

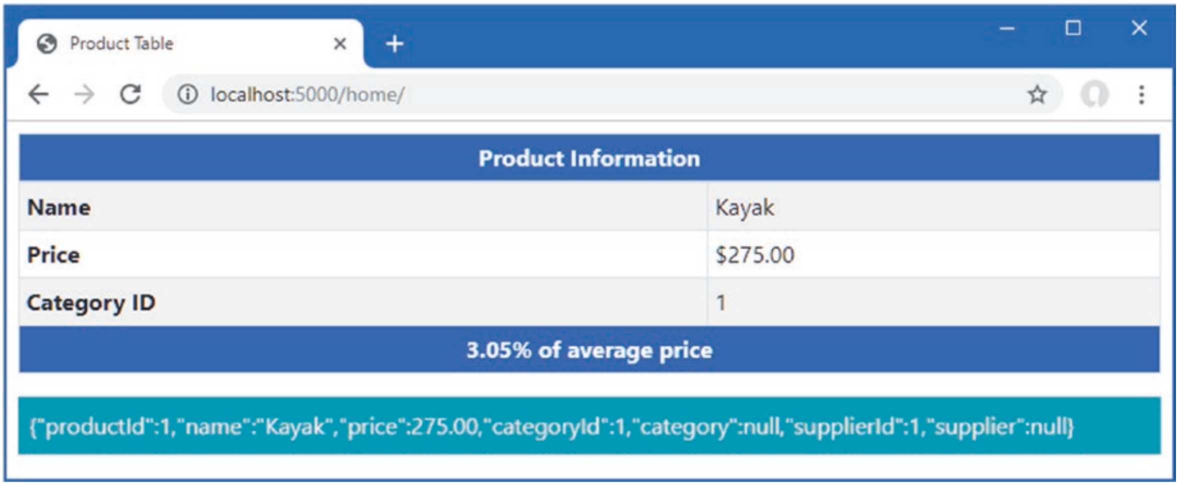

Figure 22-17. *Encoding an expression result as JSON*

Summary

In this chapter, I continued to describe the features available in Razor views. I showed you how to use the view bag, how to use layouts and partial views to deal with common content, and how to manage the encoding process for expression results. In the next chapter, I introduce Razor Pages, which provides an alternative way to generate HTML responses.

CHAPTER 23

Using Razor Pages

In this chapter, I introduce Razor Pages, which is a simpler approach to generating HTML content, intended to capture some of the enthusiasm for the legacy ASP.NET Web Pages framework. I explain how Razor Pages work, explain how they differ from the controllers and views approach taken by the MVC Framework, and show you how they fit into the wider ASP.NET Core platform.

The process of explaining how Razor Pages work can minimize the differences from the controllers and views described in earlier chapters. You might form the impression that Razor Pages are just MVC-lite and dismiss them, which would be a shame. Razor Pages are interesting because of the developer experience and not the way they are implemented.

My advice is to give Razor Pages a chance, especially if you are an experienced MVC developer. Although the technology used will be familiar, the process of creating application features is different and is well-suited to small and tightly focused features that don't require the scale and complexity of controllers and views. I have been using the MVC Framework since it was first introduced, and I admit to ignoring the early releases of Razor Pages. Now, however, I find myself mixing Razor Pages and the MVC Framework in most projects, much as I did in the SportsStore example in Part 1. Table 23-1 puts Razor Pages in context.

Table 23-1. *Putting Razor Pages in Context*

Question	Answer
What are they?	Razor Pages are a simplified way of generating HTML responses.
Why are they useful?	The simplicity of Razor Pages means you can start getting results sooner than with the MVC Framework, which can require a relatively complex preparation process. Razor Pages are also easier for less experienced web developers to understand because the relationship between the code and content is more obvious.
How are they used?	Razor Pages associate a single view with the class that provides it with features and uses a file-based routing system to match URLs.
Are there any pitfalls or limitations?	Razor Pages are less flexible than the MVC Framework, which makes them unsuitable for complex applications. Razor Pages can be used only to generate HTML responses and cannot be used to create RESTful web services.
Are there any alternatives?	The MVC Framework's approach of controllers and views can be used instead of Razor Pages.

Table 23-2 summarizes the chapter.

A. Freeman, *Pro ASP.NET Core 3*, https://doi.org/10.1007/978-1-4842-5440-0_23

Table 23-2. *Chapter Summary*

Problem	Solution	Listing
Enabling Razor Pages	Use `AddRazorPages` and `MapRazorPages` to set up the required services and middleware	3
Creating a self-contained endpoint	Create a Razor Page	4, 26, 27
Routing requests to a Razor Page	Use the name of the page or specify a route using the `@page` directive	5–8
Providing logic to support the view section of a Razor Page	Use a page model class	9–12
Creating results that are not rendered using the view section of a Razor Page	Define a handler method that returns an action result	13–15
Handling multiple HTTP methods	Define handlers in the page model class	16–18
Avoiding duplication of content	Use a layout or a partial view	19–25

Preparing for This Chapter

This chapter uses the `WebApp` project from Chapter 22. Open a new PowerShell command prompt, navigate to the folder that contains the `WebApp.csproj` file, and run the command shown in Listing 23-1 to drop the database.

■ **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-3`. See Chapter 1 for how to get help if you have problems running the examples.

Listing 23-1. Dropping the Database

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

Running the Example Application

Once the database has been dropped, select Start Without Debugging or Run Without Debugging from the Debug menu or use the PowerShell command prompt to run the command shown in Listing 23-2.

Listing 23-2. Running the Example Application

```
dotnet run
```

The database will be seeded as part of the application startup. Once ASP.NET Core is running, use a web browser to request `http://localhost:5000`, which will produce the response shown in Figure 23-1.

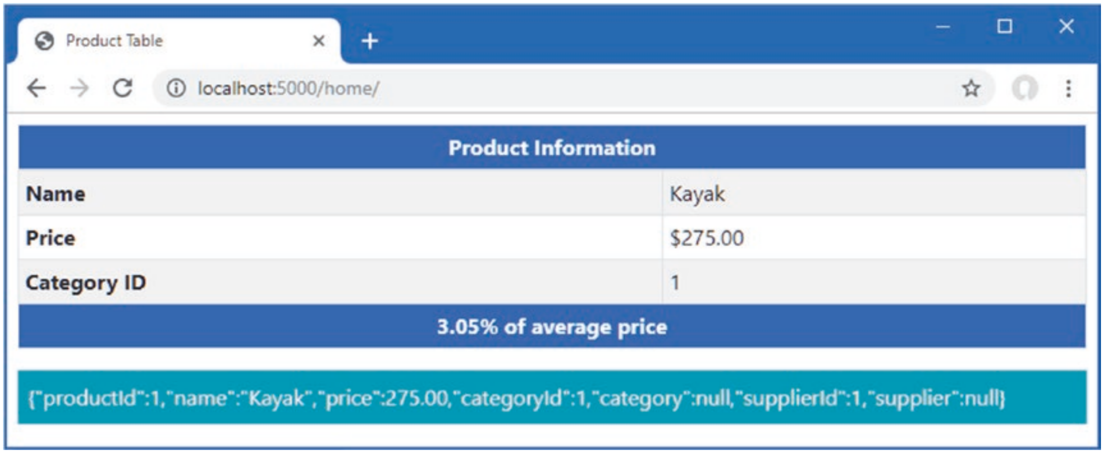

Figure 23-1. *Running the example application*

Understanding Razor Pages

As you learn how Razor Pages work, you will see they share common functionality with the MVC Framework. In fact, Razor Pages are typically described as a simplification of the MVC Framework—which is true—but that doesn't give any sense of why Razor Pages can be useful.

The MVC Framework solves every problem in the same way: a controller defines action methods that select views to produce responses. It is a solution that works because it is so flexible: the controller can define multiple action methods that respond to different requests, the action method can decide which view will be used as the request is being processed, and the view can depend on private or shared partial views to produce its response.

Not every feature in web applications needs the flexibility of the MVC Framework. For many features, a single action method will be used to handle a wide range of requests, all of which are dealt with using the same view. Razor Pages offer a more focused approach that ties together markup and C# code, sacrificing flexibility for focus.

But Razor Pages have limitations. Razor Pages tend to start out focusing on a single feature but slowly grow out of control as enhancements are made. And, unlike MVC controllers, Razor Pages cannot be used to create web services.

You don't have to choose just one model because the MVC Framework and Razor Pages coexist, as demonstrated in this chapter. This means that self-contained features can be easily developed with Razor Pages, leaving the more complex aspects of an application to be implemented using the MVC controllers and actions.

In the sections that follow, I show you how to configure and use Razor pages, and then I explain how they work and demonstrate the common foundation they share with MVC controllers and actions.

Configuring Razor Pages

To prepare the application for Razor Pages, statements must be added to the `Startup` class to set up services and configure the endpoint routing system, as shown in Listing 23-3.

Listing 23-3. Configuring the Application in the Startup.cs File in the WebApp Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using WebApp.Models;
```

```
namespace WebApp {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        public IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<DataContext>(opts => {
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:ProductConnection"]);
                opts.EnableSensitiveDataLogging(true);
            });
            services.AddControllersWithViews().AddRazorRuntimeCompilation();
            services.AddRazorPages().AddRazorRuntimeCompilation();

            services.AddDistributedMemoryCache();
            services.AddSession(options => {
                options.Cookie.IsEssential = true;
            });
        }

        public void Configure(IApplicationBuilder app, DataContext context) {
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseSession();
            app.UseRouting();
            app.UseEndpoints(endpoints => {
                endpoints.MapControllers();
                endpoints.MapDefaultControllerRoute();
                endpoints.MapRazorPages();
            });
            SeedData.SeedDatabase(context);
        }
    }
}
```

The AddRazorPages method sets up the service that is required to use Razor Pages, while the optional AddRazorRuntimeCompilation method enables runtime recompilation, using the package added to the project in Chapter 21. The MapRazorPages method creates the routing configuration that matches URLs to pages, which is explained later in the chapter.

Creating a Razor Page

Razor Pages are defined in the Pages folder. If you are using Visual Studio, create the WebApp/Pages folder, right-click it in the Solution Explorer, select Add ➤ New Item from the popup menu, and select the Razor Page template, as shown in Figure 23-2. Set the Name field to Index.cshtml and click the Add button to create the file and replace the contents of the file with those shown in Listing 23-4.

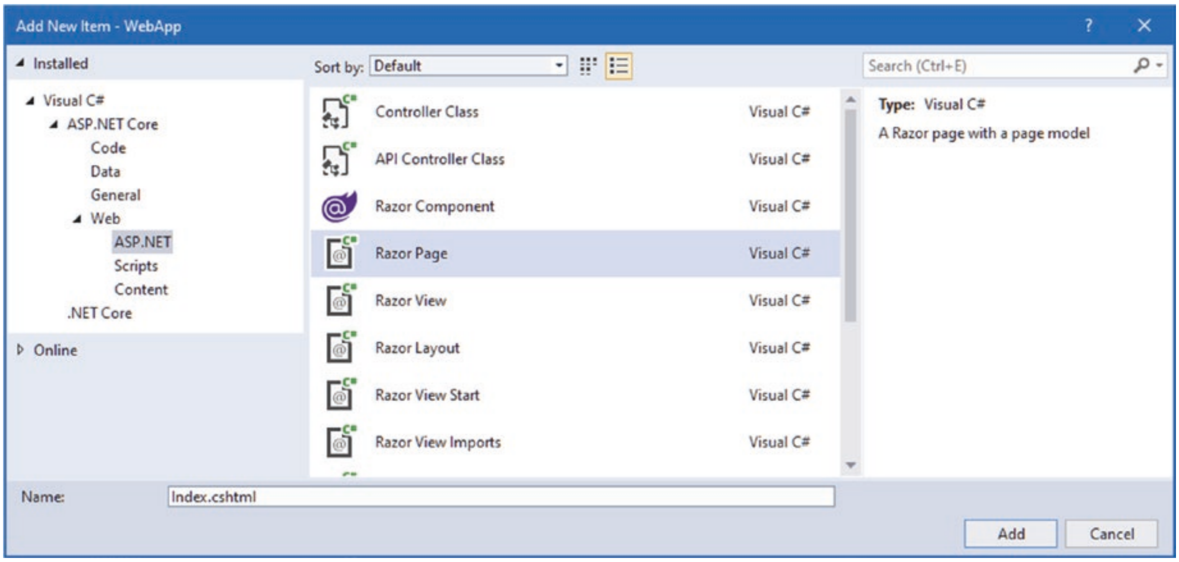

Figure 23-2. *Creating a Razor Page*

If you are using Visual Studio Code, create the `WebApp/Pages` folder and add to it a new file named `Index.cshtml` with the content shown in Listing 23-4.

Listing 23-4. The Contents of the Index.cshtml File in the Pages Folder

```
@page
@model IndexModel
@using Microsoft.AspNetCore.Mvc.RazorPages
@using WebApp.Models;

<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="bg-primary text-white text-center m-2 p-2">@Model.Product.Name</div>
</body>
</html>

@functions {

    public class IndexModel: PageModel {
        private DataContext context;

        public Product Product { get; set; }

        public IndexModel(DataContext ctx) {
            context = ctx;
        }

        public async Task OnGetAsync(long id = 1) {
            Product = await context.Products.FindAsync(id);
        }
    }
}
```

Razor Pages use the Razor syntax that I described in Chapters 21 and 22, and Razor Pages even use the same CSHTML file extension. But there are some important differences.

The `@page` directive must be the first thing in a Razor Page, which ensures that the file is not mistaken for a view associated with a controller. But the most important difference is that the `@functions` directive is used to define the C# code that supports the Razor content in the same file. I explain how Razor Pages work shortly, but to see the output generated by the Razor Page, restart ASP.NET Core and use a browser to request `http://localhost:5000/index`, which produces the response shown in Figure 23-3.

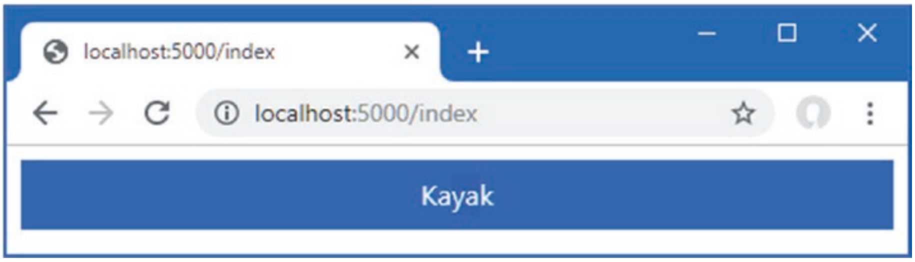

Figure 23-3. *Using a Razor Page*

Understanding the URL Routing Convention

URL routing for Razor Pages is based on the file name and location, relative to the `Pages` folder. The Razor Page in Listing 23-4 is in a file named `Index.cshtml`, in the `Pages` folder, which means that it will handle requests for the `/index`. The routing convention can be overridden, as described in the "Understanding Razor Pages Routing" section, but, by default, it is the location of the Razor Page file that determines the URLs that it responds to.

Understanding the Page Model

In a Razor Page, the `@model` directive is used to select a *page model* class, rather than identifying the type of the object provided by an action method. The `@model` directive in Listing 23-4 selects the `IndexModel` class.

```
...
@model IndexModel
...
```

The page model is defined within the `@functions` directive and is derived from the `PageModel` class, like this:

```
...
@functions {
    public class IndexModel: PageModel {
...
```

When the Razor Page is selected to handle an HTTP request, a new instance of the page model class is created, and dependency injection is used to resolve any dependencies that have been declared using constructor parameters, using the features described in Chapter 14. The `IndexModel` class declares a dependency on the `DataContext` service created in Chapter 18, which allows it to access the data in the database.

```
...
public IndexModel(DataContext ctx) {
    context = ctx;
}
...
```

After the page model object has been created, a handler method is invoked. The name of the handler method is On, followed by the HTTP method for the request so that the OnGet method is invoked when the Razor Page is selected to handle an HTTP GET request. Handler methods can be asynchronous, in which case a GET request will invoke the OnGetAsync method, which is the method implemented by the IndexModel class.

```
...
public async Task OnGetAsync(long id = 1) {
    Product = await context.Products.FindAsync(id);
}
...
```

Values for the handler method parameters are obtained from the HTTP request using the model binding process, which is described in detail in Chapter 28. The OnGetAsync method receives the value for its id parameters from the model binder, which it uses to query the database and assign the result to its Product property.

Understanding the Page View

Razor Pages use the same mix of HTML fragments and code expressions to generate content, which defines the view presented to the user. The page model's methods and properties are accessible in the Razor Page through the @Model expression. The Product property defined by the IndexModel class is used to set the content of an HTML element, like this:

```
...
<div class="bg-primary text-white text-center m-2 p-2">@Model.Product.Name</div>
...
```

The @Model expression returns an IndexModel object, and this expression reads the Name property of the object returned by the Product property.

Understanding the Generated C# Class

Behind the scenes, Razor Pages are transformed into C# classes, just like regular Razor views. Here is a simplified version of the C# class that is produced from the Razor Page in Listing 23-4:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Microsoft.AspNetCore.Mvc.Razor.TagHelpers;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Razor.Runtime.TagHelpers;
using Microsoft.AspNetCore.Razor.TagHelpers;
using WebApp.Models;

namespace AspNetCore {

    public class Pages_Index : Page {
        public <IndexModel> ViewData => (<IndexModel>)PageContext?.ViewData;
        public IndexModel Model => ViewData.Model;

        public async override Task ExecuteAsync() {
            WriteLiteral("\r\n<!DOCTYPE html>\r\n<html>\r\n");
            WriteLiteral("<head>");
```

```
            WriteLiteral("@<link
                href=\"/lib/twitter-bootstrap/css/bootstrap.min.css\"
                rel=\"stylesheet\" />");
            WriteLiteral("</head>");
            WriteLiteral("<body>");
            WriteLiteral("<div class=\"bg-primary text-white text-center m-2 p-2\">")
            Write(Model.Product.Name);
            WriteLiteral("</div>");
            WriteLiteral("</body></html>\r\n\r\n");
        }

        public class IndexModel: PageModel {
            private DataContext context;
            public Product Product { get; set; }

            public IndexModel(DataContext ctx) {
                context = ctx;
            }

            public async Task OnGetAsync(long id = 1) {
                Product = await context.Products.FindAsync(id);
            }
        }

        public IUrlHelper Url { get; private set; }
        public IViewComponentHelper Component { get; private set; }
        public IJsonHelper Json { get; private set; }
        public IHtmlHelper<IndexModel> Html { get; private set; }
        public IModelExpressionProvider ModelExpressionProvider { get; private set; }
    }
}
```

If you compare this code with the equivalent shown in Chapter 21, you can see how Razor Pages rely on the same features used by the MVC Framework. The HTML fragments and view expressions are transformed into calls to the `WriteLiteral` and `Write` methods.

■ **Tip** You can see the generated classes by examining the contents of the `obj/Debug/netcoreapp3.0/Razor/Pages` folder with the Windows File Explorer.

Understanding Razor Pages Routing

Razor Pages rely on the location of the CSHTML file for routing so that a request for `http://localhost:5000/index` is handled by the `Pages/Index.cshtml` file. Adding a more complex URL structure for an application is done by adding folders whose names represent the segments in the URL you want to support. As an example, create the `WebApp/Pages/Suppliers` folder and add to it a Razor Page named `List.cshtml` with the contents shown in Listing 23-5.

Listing 23-5. The Contents of the List.cshtml File in the Pages/Suppliers Folder

```
@page
@model ListModel
@using Microsoft.AspNetCore.Mvc.RazorPages
@using WebApp.Models;
```

```
<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <h5 class="bg-primary text-white text-center m-2 p-2">Suppliers</h5>
    <ul class="list-group m-2">
        @foreach (string s in Model.Suppliers) {
            <li class="list-group-item">@s</li>
        }
    </ul>
</body>
</html>

@functions {

    public class ListModel : PageModel {
        private DataContext context;

        public IEnumerable<string> Suppliers { get; set; }

        public ListModel(DataContext ctx) {
            context = ctx;
        }

        public void OnGet() {
            Suppliers = context.Suppliers.Select(s => s.Name);
        }
    }
}
```

The new page model class defines a `Suppliers` property that is set to the sequence of `Name` values for the `Supplier` objects in the database. The database operation in this example is synchronous, so the page model class defined the `OnGet` method, rather than `OnGetAsync`. The supplier names are displayed in a list using an `@foreach` expression. To use the new Razor Page, use a browser to request `http://localhost:5000/suppliers/list`, which produces the response shown in Figure 23-4. The path segments of the request URL correspond to the folder and file name of the `List.cshtml` Razor Page.

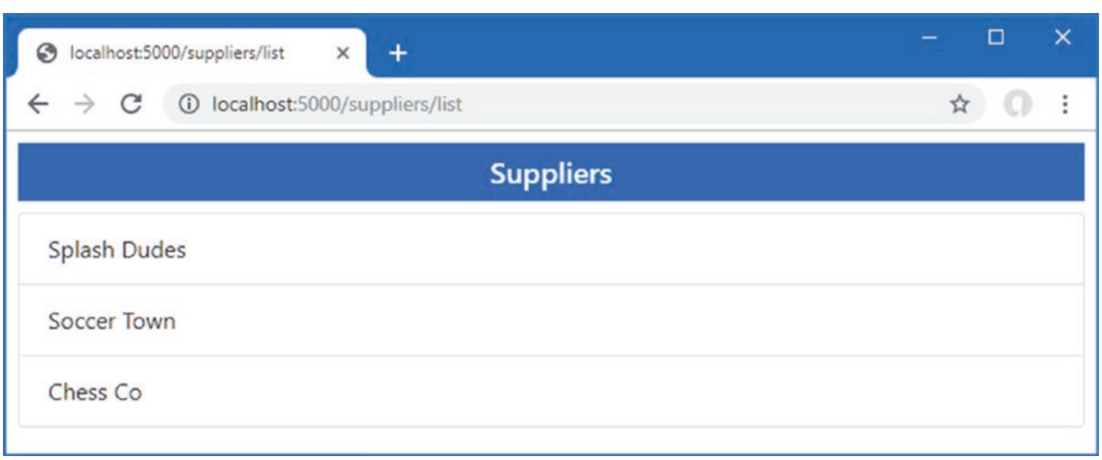

Figure 23-4. *Using a folder structure to route requests*

UNDERSTANDING THE DEFAULT URL HANDLING

The `MapRazorPages` method sets up a route for the default URL for the `Index.cshtml` Razor Page, following a similar convention used by the MVC Framework. It is for this reason that the first Razor Page added to a project is usually called `Index.cshtml`. However, when the application mixes Razor Pages and the MVC Framework together, the default route is set up by whichever is configured first, which is why requests for `http://localhost:5000` for the example application are handled by the `Index` action of the `Home` MVC controller. If you want the `Index.cshtml` file to handle the default URL, then you can change the order of the endpoint routing statements so that Razor Pages is set up first, like this:

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

In my own projects, where I mix Razor Pages and MVC controllers, I tend to rely on the MVC Framework to handle the default URL, and I avoid creating the `Index.cshtml` Razor Page to avoid confusion.

Specifying a Routing Pattern in a Razor Page

Using the folder and file structure to perform routing means there are no segment variables for the model binding process to use. Instead, values for the request handler methods are obtained from the URL query string, which you can see by using a browser to request `http://localhost:5000/index?id=2`, which produces the response shown in Figure 23-5.

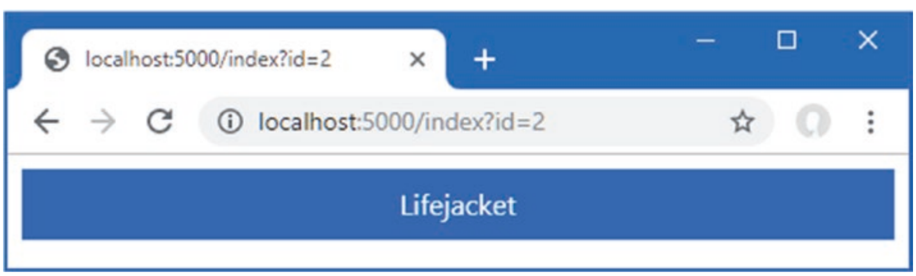

Figure 23-5. *Using a query string parameter*

The query string provides a parameter named `id`, which the model binding process uses to satisfy the `id` parameter defined by the `OnGetAsync` method in the `Index` Razor Page.

```
...
public async Task OnGetAsync(long id = 1) {
...
```

I explain how model binding works in detail in Chapter 28, but for now, it is enough to know that the query string parameter in the request URL is used to provide the `id` argument when the `OnGetAsync` method is invoked, which is used to query the database for a product.

The `@page` directive can be used with a routing pattern, which allows segment variables to be defined, as shown in Listing 23-6.

Listing 23-6. Defining a Segment Variable in the Index.cshtml File in the Pages Folder

```
@page "{id:long?}"
@model IndexModel
@using Microsoft.AspNetCore.Mvc.RazorPages
@using WebApp.Models;

<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="bg-primary text-white text-center m-2 p-2">@Model.Product.Name</div>
</body>
</html>

@functions {

    // ...statements omitted for brevity...
}
```

All the URL pattern features that are described in Chapter 13 can be used with the `@page` directive. The route pattern used in Listing 23-6 adds an optional segment variable named `id`, which is constrained so that it will match only those segments that can be parsed to a `long` value. To see the change, restart ASP.NET Core (automatic recompilation doesn't detect routing changes) and use a browser to request `http://localhost:5000/index/4`, which produces the response shown on the left of Figure 23-6.

The `@page` directive can also be used to override the file-based routing convention for a Razor Page, as shown in Listing 23-7.

Listing 23-7. Changing the Route in the List.cshtml File in the Pages/Suppliers Folder

```
@page "/lists/suppliers"
@model ListModel
@using Microsoft.AspNetCore.Mvc.RazorPages
@using WebApp.Models;

<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <h5 class="bg-primary text-white text-center m-2 p-2">Suppliers</h5>
    <ul class="list-group m-2">
        @foreach (string s in Model.Suppliers) {
            <li class="list-group-item">@s</li>
        }
    </ul>
</body>
</html>

@functions {

    // ...statements omitted for brevity...
}
```

The directive changes the route for the `List` page so that it matches URLs whose path is `/lists/suppliers`. To see the effect of the change, restart ASP.NET Core and request `http://localhost:5000/lists/suppliers`, which produces the response shown on the right of Figure 23-6.

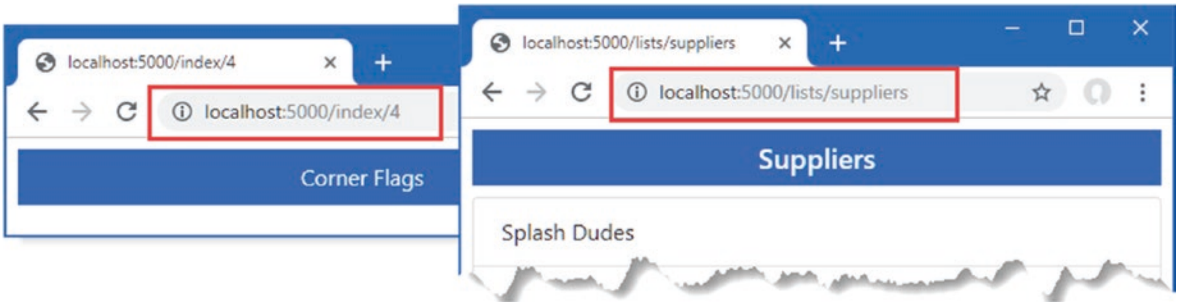

Figure 23-6. *Changing routes using the @page directive*

Adding Routes for a Razor Page

Using the @page directive replaces the default file-based route for a Razor Page. If you want to define multiple routes for a page, then configuration statements can be added to the `Startup` class, as shown in Listing 23-8.

Listing 23-8. Adding Razor Page Routes in the Startup.cs File in the WebApp Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using WebApp.Models;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace WebApp {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        public IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<DataContext>(opts => {
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:ProductConnection"]);
                opts.EnableSensitiveDataLogging(true);
            });
            services.AddControllersWithViews().AddRazorRuntimeCompilation();
            services.AddRazorPages().AddRazorRuntimeCompilation();

            services.AddDistributedMemoryCache();
            services.AddSession(options => {
                options.Cookie.IsEssential = true;
            });

            services.Configure<RazorPagesOptions>(opts => {
                opts.Conventions.AddPageRoute("/Index", "/extra/page/{id:long?}");
            });
        }
```

```
        public void Configure(IApplicationBuilder app, DataContext context) {
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseSession();
            app.UseRouting();
            app.UseEndpoints(endpoints => {
                endpoints.MapControllers();
                endpoints.MapDefaultControllerRoute();
                endpoints.MapRazorPages();
            });
            SeedData.SeedDatabase(context);
        }
    }
}
```

The options pattern is used to add additional routes for a Razor Page using the RazorPageOptions class. The AddPageRoute extension method is called on the Conventions property to add a route for a page. The first argument is the path to the page, without the file extension and relative to the Pages folder. The second argument is the URL pattern to add to the routing configuration. To test the new route, restart ASP.NET Core and use a browser to request http://localhost:5000/extra/page/2, which is matched by the URL pattern added in Listing 23-8 and produces the response shown on the left of Figure 23-7. The route added in Listing 23-8 supplements the route defined by the @page attribute, which you can test by requesting http://localhost:5000/index/2, which will produce the response shown on the right of Figure 23-7.

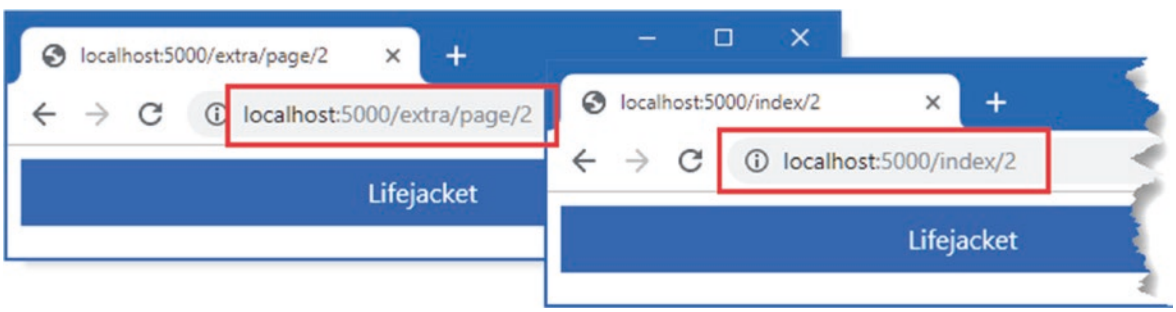

Figure 23-7. *Adding a route for a Razor Page*

Understanding the Page Model Class

Page models are derived from the PageModel class, which provides the link between the rest of ASP.NET Core and the view part of the Razor Page. The PageModel class provides methods for managing how requests are handled and properties that provide context data, the most useful of which are described in Table 23-3. I have listed these properties for completeness, but they are not often required in Razor Page development, which focuses more on selecting the data that is required to render the view part of the page.

Table 23-3. *Selected PageModel Properties for Context Data*

Name	Description
`HttpContext`	This property returns an `HttpContext` object, described in Chapter 12.
`ModelState`	This property provides access to the model binding and validation features described in Chapters 28 and 29.
`PageContext`	This property returns a `PageContext` object that provides access to many of the same properties defined by the `PageModel` class, along with additional information about the current page selection.
`Request`	This property returns an `HttpRequest` object that describes the current HTTP request, as described in Chapter 12.
`Response`	This property returns an `HttpResponse` object that represents the current response, as described in Chapter 12.
`RouteData`	This property provides access to the data matched by the routing system, as described in Chapter 13.
`TempData`	This property provides access to the temp data feature, which is used to store data until it can be read by a subsequent request. See Chapter 22 for details.
`User`	This property returns an object that describes the user associated with the request, as described in Chapter 38.

Using a Code-Behind Class File

The `@function` directive allows the page-behind class and the Razor content to be defined in the same file, which is a development approach used by popular client-side frameworks, such as React or Vue.js.

Defining code and markup in the same file is convenient but can become difficult to manage for more complex applications. Razor Pages can also be split into separate view and code files, which is similar to the MVC examples in previous chapters and is reminiscent of ASP.NET Web Pages, which defined C# classes in files known as *code-behind files*. The first step is to remove the page model class from the CSHTML file, as shown in Listing 23-9.

Listing 23-9. Removing the Page Model Class in the Index.cshtml File in the Pages Folder

```
@page "{id:long?}"
@model WebApp.Pages.IndexModel

<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="bg-primary text-white text-center m-2 p-2">@Model.Product.Name</div>
</body>
</html>
```

The convention for naming Razor Pages code-behind files is to append the `.cs` file extension to the name of the view file. If you are using Visual Studio, the code-behind file was created by the Razor Page template when the `Index.cshtml` file was added to the project. Expand the `Index.cshtml` item in the Solution Explorer and you will see the code-behind file, as shown in Figure 23-8. Open the file for editing and replace the contents with the statements shown in Listing 23-10.

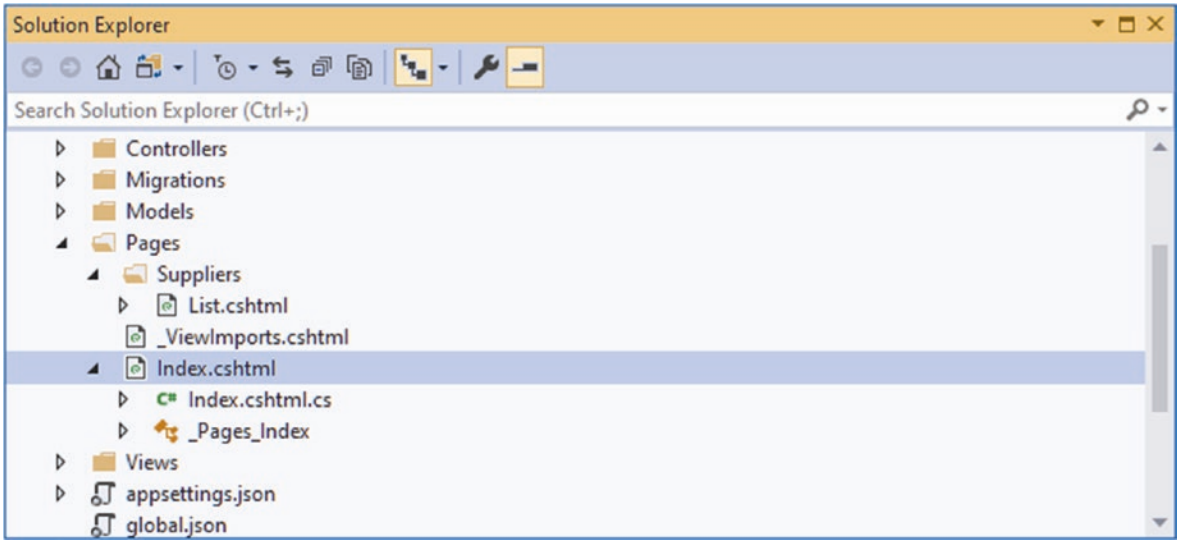

Figure 23-8. *Revealing the code-behind file in the Visual Studio Solution Explorer*

If you are using Visual Studio Code, add a file named `Index.cshtml.cs` to the `WebApp/Pages` folder with the content shown in Listing 23-10.

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

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.RazorPages;
using WebApp.Models;

namespace WebApp.Pages {

    public class IndexModel: PageModel {
        private DataContext context;

        public Product Product { get; set; }

        public IndexModel(DataContext ctx) {
            context = ctx;
        }

        public async Task OnGetAsync(long id = 1) {
            Product = await context.Products.FindAsync(id);
        }
    }
}
```

When defining the separate page model class, I defined the class in the `WebApp.Pages` namespace. This isn't a requirement, but it makes the C# class consistent with the rest of the application.

One drawback of using a code-behind file is that automatic recompilation applies only to CSHTML files, which means that changes to the class file are not applied until the application has been restarted. Restart ASP.NET Core and request `http://localhost:5000/index` to ensure the code-behind file is used, producing the response shown in Figure 23-9.

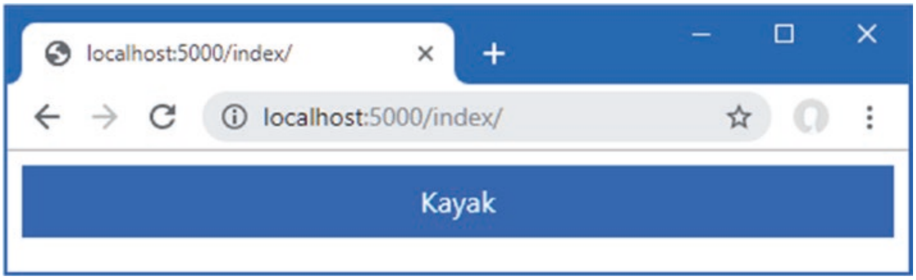

Figure 23-9. *Using a code-behind file*

Adding a View Imports File

A view imports file can be used to avoid using the fully qualified name for the page model class in the view file, performing the same role as the one I used in Chapter 22 for the MVC Framework. If you are using Visual Studio, use the Razor View Imports template to add a file named `_ViewImports.cshtml` to the `WebApp/Pages` folder, with the content shown in Listing 23-11. If you are using Visual Studio Code, add the file directly.

Listing 23-11. The Contents of the _ViewImports.cshtml File in the WebApp/Pages Folder

```
@namespace WebApp.Pages
@using WebApp.Models
```

The `@namespace` directive sets the namespace for the C# class that is generated by a view, and using the directive in the view imports file sets the default namespace for all the Razor Pages in the application, with the effect that the view and its page model class are in the same namespace and the `@model` directive does not require a fully qualified type, as shown in Listing 23-12.

Listing 23-12. Removing the Page Model Namespace in the Index.cshtml File in the Pages Folder

```
@page "{id:long?}"
@model IndexModel

<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="bg-primary text-white text-center m-2 p-2">@Model.Product.Name</div>
</body>
</html>
```

Use the browser to request `http://localhost:5000/index`, which will trigger the recompilation of the views. There is no difference in the response produced by the Razor Page, which is shown in Figure 23-9.

Understanding Action Results in Razor Pages

Although it is not obvious, Razor Page handler methods use the same `IActionResult` interface to control the responses they generate. To make page model classes easier to develop, handler methods have an implied result that displays the view part of the page. Listing 23-13 makes the result explicit.

Listing 23-13. Using an Explicit Result in the Index.cshtml.cs File in the Pages Folder

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.RazorPages;
using WebApp.Models;
using Microsoft.AspNetCore.Mvc;
```

```
namespace WebApp.Pages {

    public class IndexModel : PageModel {
        private DataContext context;

        public Product Product { get; set; }

        public IndexModel(DataContext ctx) {
            context = ctx;
        }

        public async Task<IActionResult> OnGetAsync(long id = 1) {
            Product = await context.Products.FindAsync(id);
            return Page();
        }
    }
}
```

The Page method is inherited from the PageModel class and creates a PageResult object, which tells the framework to render the view part of the page. Unlike the View method used in MVC action methods, the Razor Pages Page method doesn't accept arguments and always renders the view part of the page that has been selected to handle the request.

The PageModel class provides other methods that create different action results to produce different outcomes, as described in Table 23-4.

Table 23-4. *The PageModel Action Result Methods*

Name	Description
`Page()`	This `IActionResult` returned by this method produces a 200 OK status code and renders the view part of the Razor Page.
`NotFound()`	The `IActionResult` returned by this method produces a 404 NOT FOUND status code.
`BadRequest(state)`	The `IActionResult` returned by this method produces a 400 BAD REQUEST status code. The method accepts an optional model state object that describes the problem to the client, as demonstrated in Chapter 19.
`File(name, type)`	The `IActionResult` returned by this method produces a 200 OK response, sets the `Content-Type` header to the specified type, and sends the specified file to the client.
`Redirect(path)` `RedirectPermanent(path)`	The `IActionResult` returned by these methods produces 302 FOUND and 301 MOVED PERMANENTLY responses, which redirect the client to the specified URL.
`RedirectToAction(name)RedirectToActionPermanent(name)`	The `IActionResult` returned by these methods produces 302 FOUND and 301 MOVED PERMANENTLY responses, which redirect the client to the specified action method. The URL used to redirect the client is produced using the routing features described in Chapter 13.
`RedirectToPage(name)` `RedirectToPagePermanent(name)`	The `IActionResult` returned by these methods produce 302 FOUND and 301 MOVED PERMANENTLY responses that redirect the client to another Razor Page. If no name is supplied, the client is redirected to the current page.
`StatusCode(code)`	The `IActionResult` returned by this method produces a response with the specific status code.

Using an Action Result

Except for the Page method, the methods in Table 23-4 are the same as those available in action methods. However, care must be taken with these methods because sending a status code response is unhelpful in Razor Pages because they are used only when a client expects the content of the view.

Instead of using the NotFound method when requested data cannot be found, for example, a better approach is to redirect the client to another URL that can display an HTML message for the user. The redirection can be to a static HTML file, to another Razor Page, or to an action defined by a controller. Add a Razor Page named NotFound.cshtml to the Pages folder and add the content shown in Listing 23-14.

Listing 23-14. The Contents of the NotFound.cshtml File in the Pages Folder

```
@page "/noid"
@model NotFoundModel
@using Microsoft.AspNetCore.Mvc.RazorPages
@using WebApp.Models;

<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
    <title>Not Found</title>
</head>
<body>
    <div class="bg-primary text-white text-center m-2 p-2">No Matching ID</div>
    <ul class="list-group m-2">
        @foreach (Product p in Model.Products) {
            <li class="list-group-item">@p.Name (ID: @p.ProductId)</li>
        }
    </ul>
</body>
</html>

@functions {

    public class NotFoundModel: PageModel {
        private DataContext context;

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

        public NotFoundModel(DataContext ctx) {
            context = ctx;
        }

        public void OnGetAsync(long id = 1) {
            Products = context.Products;
        }
    }
}
```

The @page directive overrides the route convention so that this Razor Page will handle the /noid URL path. The page model class uses an Entity Framework Core context object to query the database and displays a list of the product names and key values that are in the database.

In Listing 23-15, I have updated the handle method of the IndexModel class to redirect the user to the NotFound page when a request is received that doesn't match a Product object in the database.

Listing 23-15. Using a Redirection in the Index.cshtml.cs File in the Pages Folder

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.RazorPages;
using WebApp.Models;
using Microsoft.AspNetCore.Mvc;
```

```
namespace WebApp.Pages {

    public class IndexModel : PageModel {
        private DataContext context;

        public Product Product { get; set; }

        public IndexModel(DataContext ctx) {
            context = ctx;
        }

        public async Task<IActionResult> OnGetAsync(long id = 1) {
            Product = await context.Products.FindAsync(id);
            if (Product == null) {
                return RedirectToPage("NotFound");
            }
            return Page();
        }
    }
}
```

The `RedirectToPage` method produces an action result that redirects the client to a different Razor Page. The name of the target page is specified without the file extension, and any folder structure is specified relative to the `Pages` folder. To test the redirection, restart ASP.NET Core and request `http://localhost:5000/index/500`, which provides a value of 500 for the `id` segment variable and does not match anything in the database. The browser will be redirected and produce the result shown in Figure 23-10.

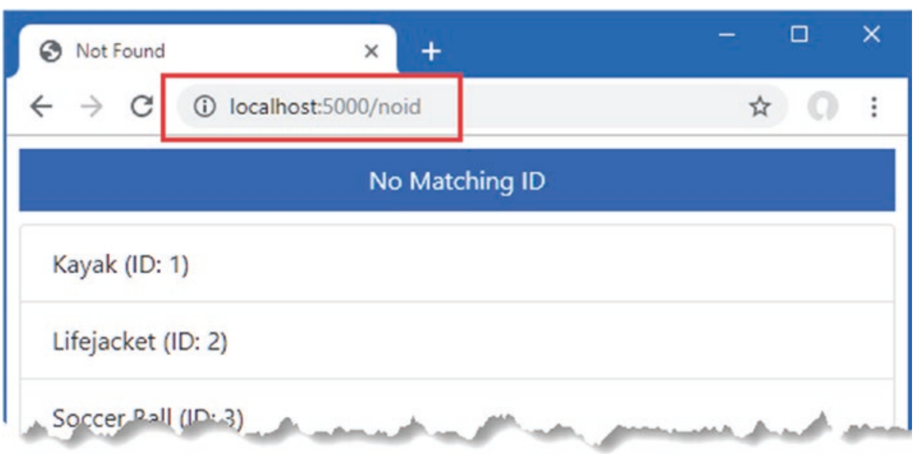

Figure 23-10. *Redirecting to a different Razor Page*

Notice that the routing system is used to produce the URL to which the client is redirected, which uses the routing pattern specified with the `@page` directive. In this example, the argument to the `RedirectToPage` method was `NotFound`, but this has been translated into a redirection to the `/noid` path specified by the `@page` directive in Listing 23-14.

Handling Multiple HTTP Methods

Razor Pages can define handler methods that respond to different HTTP methods. The most common combination is to support the GET and POST methods that allow users to view and edit data. To demonstrate, add a Razor Page called `Editor.cshtml` to the `Pages` folder and add the content shown in Listing 23-16.

■ **Note** I have kept this example as simple as possible, but there are excellent ASP.NET Core features for creating HTML forms and for receiving data when it is submitted, as described in Chapter 31.

Listing 23-16. The Contents of the Editor.cshtml File in the WebApps/Pages Folder

```
@page "{id:long}"
@model EditorModel

<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="bg-primary text-white text-center m-2 p-2">Editor</div>
    <div class="m-2">
        <table class="table table-sm table-striped table-bordered">
            <tbody>
                <tr><th>Name</th><td>@Model.Product.Name</td></tr>
                <tr><th>Price</th><td>@Model.Product.Price</td></tr>
            </tbody>
        </table>
        <form method="post">
            @Html.AntiForgeryToken()
            <div class="form-group">
                <label>Price</label>
                <input name="price" class="form-control"
                       value="@Model.Product.Price" />
            </div>
            <button class="btn btn-primary" type="submit">Submit</button>
        </form>
    </div>
</body>
</html>
```

The elements in the Razor Page view create a simple HTML form that presents the user with an input element containing the value of the `Price` property for a `Product` object. The `form` element is defined without an action attribute, which means the browser will send a POST request to the Razor Page's URL when the user clicks the Submit button.

■ **Note** The `@Html.AntiForgeryToken()` expression in Listing 23-16 adds a hidden form field to the HTML form that ASP.NET Core uses to guard against cross-site request forgery (CSRF) attacks. I explain how this feature works in Chapter 27, but for this chapter, it is enough to know that POST requests that do not contain this form field will be rejected.

If you are using Visual Studio, expand the `Editor.cshtml` item in the Solution Explorer to reveal the `Editor.cshtml.cs` class file and replace its contents with the code shown in Listing 23-17. If you are using Visual Studio Code, add a file named `Editor.cshtml.cs` to the `WebApp/Pages` folder and use it to define the class shown in Listing 23-17.

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

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using WebApp.Models;
```

```
namespace WebApp.Pages {
    public class EditorModel : PageModel {
        private DataContext context;

        public Product Product { get; set; }

        public EditorModel(DataContext ctx) {
            context = ctx;
        }

        public async Task OnGetAsync(long id) {
            Product = await context.Products.FindAsync(id);
        }

        public async Task<IActionResult> OnPostAsync(long id, decimal price) {
            Product p = await context.Products.FindAsync(id);
            p.Price = price;
            await context.SaveChangesAsync();
            return RedirectToPage();
        }
    }
}
```

The page model class defines two handler methods, and the name of the method tells the Razor Pages framework which HTTP method each handles. The `OnGetAsync` method is used to handle GET requests, which it does by locating a `Product`, whose details are displayed by the view.

The `OnPostAsync` method is used to handle POST requests, which will be sent by the browser when the user submits the HTML form. The parameters for the `OnPostAsync` method are obtained from the request so that the `id` value is obtained from the URL route and the `price` value is obtained from the form. (The model binding feature that extracts data from forms is described in Chapter 28.)

UNDERSTANDING THE POST REDIRECTION

Notice that the last statement in the `OnPostAsync` method invokes the `RedirectToPage` method without an argument, which redirects the client to the URL for the Razor Page. This may seem odd, but the effect is to tell the browser to send a GET request to the URL it used for the POST request. This type of redirection means that the browser won't resubmit the POST request if the user reloads the browser, preventing the same action from being accidentally performed more than once.

To see how the page model class handles different HTTP methods, restart ASP.NET Core and use a browser to navigate to `http://localhost:5000/editor/1`. Edit the field to set the price to 100 and click the Submit button. The browser will send a POST request that is handled by the `OnPostAsync` method. The database will be updated, and the browser will be redirected so that the updated data is displayed, as shown in Figure 23-11.

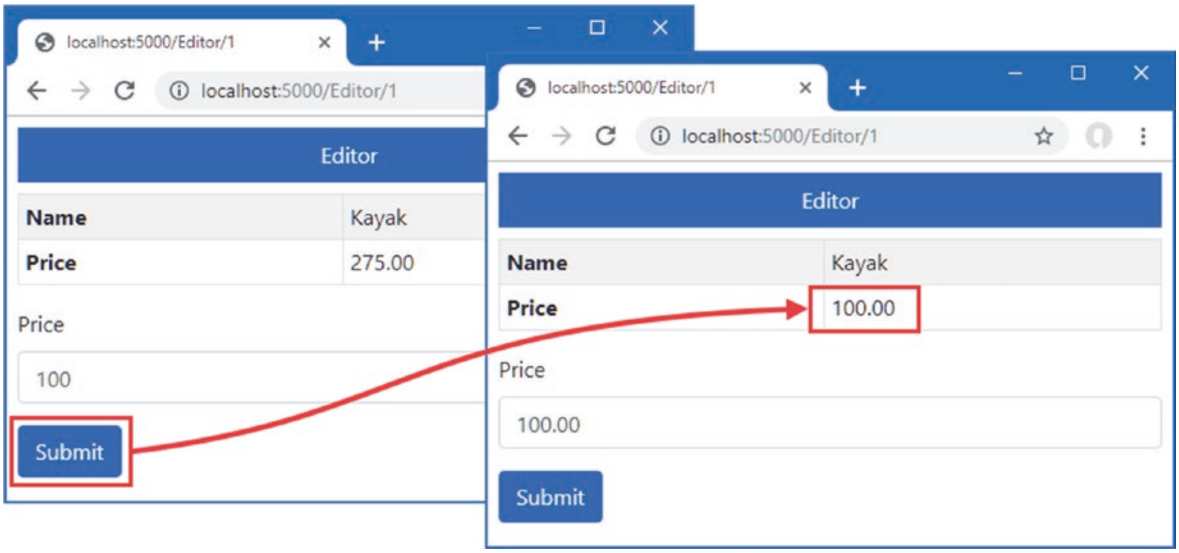

Figure 23-11. *Handling multiple HTTP methods*

Selecting a Handler Method

The page model class can define multiple handler methods, allowing the request to select a method using a `handler` query string parameter or routing segment variable. To demonstrate this feature, add a Razor Page file named `HandlerSelector.cshtml` to the Pages folder with the content shown in Listing 23-18.

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

```
@page
@model HandlerSelectorModel
@using Microsoft.AspNetCore.Mvc.RazorPages
@using Microsoft.EntityFrameworkCore

<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="bg-primary text-white text-center m-2 p-2">Selector</div>
    <div class="m-2">
        <table class="table table-sm table-striped table-bordered">
            <tbody>
                <tr><th>Name</th><td>@Model.Product.Name</td></tr>
                <tr><th>Price</th><td>@Model.Product.Name</td></tr>
                <tr><th>Category</th><td>@Model.Product.Category?.Name</td></tr>
                <tr><th>Supplier</th><td>@Model.Product.Supplier?.Name</td></tr>
            </tbody>
        </table>
        <a href="/handlerselector" class="btn btn-primary">Standard</a>
        <a href="/handlerselector?handler=related" class="btn btn-primary">
            Related
        </a>
    </div>
</body>
</html>
```

```
@functions{

    public class HandlerSelectorModel: PageModel {
        private DataContext context;

        public Product Product { get; set; }

        public HandlerSelectorModel(DataContext ctx) {
            context = ctx;
        }

        public async Task OnGetAsync(long id = 1) {
            Product = await context.Products.FindAsync(id);
        }

        public async Task OnGetRelatedAsync(long id = 1) {
            Product = await context.Products
                .Include(p => p.Supplier)
                .Include(p => p.Category)
                .FirstOrDefaultAsync(p => p.ProductId == id);
            Product.Supplier.Products = null;
            Product.Category.Products = null;
        }
    }
}
```

The page model class in this example defines two handler methods: `OnGetAsync` and `OnGetRelatedAsync`. The `OnGetAsync` method is used by default, which you can see by using a browser to request `http://localhost:5000/handlerselector`. The handler method queries the database and presents the result to the user, as shown on the left of Figure 23-12.

One of the anchor elements rendered by the page targets a URL with a handler query string parameter, like this:

```
...
<a href="/handlerselector?handler=related" class="btn btn-primary">Related</a>
...
```

The name of the handler method is specified without the `On[method]` prefix and without the `Async` suffix so that the `OnGetRelatedAsync` method is selected using a handler value of `related`. This alternative handler method includes related data in its query and presents additional data to the user, as shown on the right of Figure 23-12.

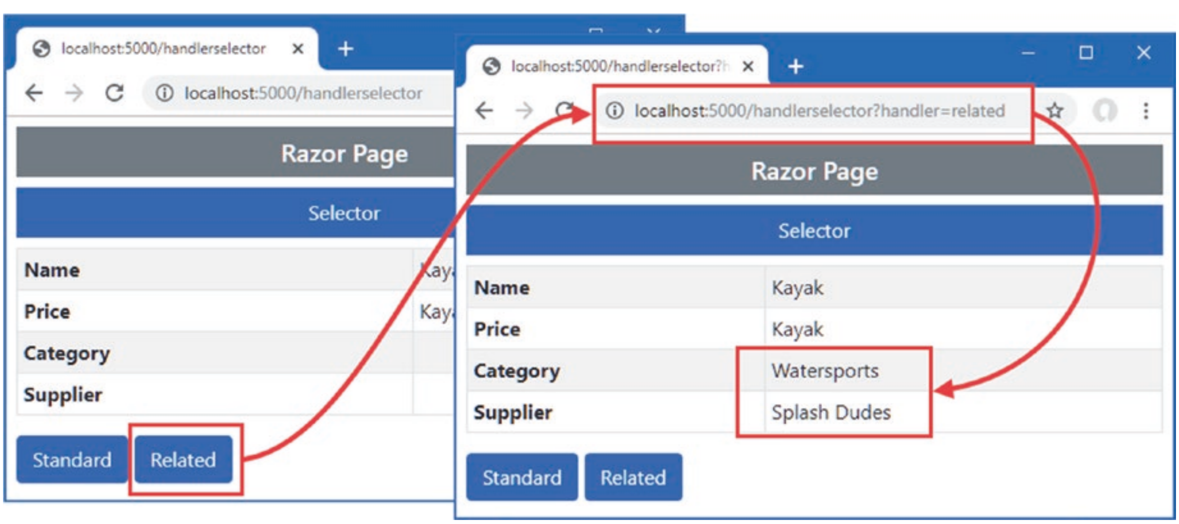

Figure 23-12. Selecting handler methods

Understanding the Razor Page View

The view part of a Razor Page uses the same syntax and has the same features as the views used with controllers. Razor Pages can use the full range of expressions and features such as sessions, temp data, and layouts. Aside from the use of the @page directive and the page model classes, the only differences are a certain amount of duplication to configure features such as layouts and partial views, as described in the sections that follow.

Creating a Layout for Razor Pages

Layouts for Razor Pages are created in the same way as for controller views but in the Pages/Shared folder. If you are using Visual Studio, create the Pages/Shared folder and add to it a file named _Layout.cshtml using the Razor Layout template with the contents shown in Listing 23-19. If you are using Visual Studio Code, create the Pages/Shared folder, create the _Layout.cshtml file in the new folder, and add the content shown in Listing 23-19.

■ **Note** Layouts can be created in the same folder as the Razor Pages that use them, in which case they will be used in preference to the files in the Shared folder.

Listing 23-19. The Contents of the _Layout.cshtml File in the Pages/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
    <title>@ViewBag.Title</title>
</head>
<body>
    <h5 class="bg-secondary text-white text-center m-2 p-2">
        Razor Page
    </h5>
    @RenderBody()
</body>
</html>
```

The layout doesn't use any features that are specific to Razor Pages and contains the same elements and expressions used in Chapter 22 when I created a layout for the controller views.

Next, use the Razor View Start template to add a file named _ViewStart.cshtml to the Pages folder. Visual Studio will create the file with the content shown in Listing 23-20. If you are using Visual Studio Code, create the _ViewStart.cshtml file and add the content shown in Listing 23-20.

Listing 23-20. The Contents of the _ViewStart.cshtml File in the Pages Folder

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

The C# classes generated from Razor Pages are derived from the Page class, which provides the Layout property used by the view start file, which has the same purpose as the one used by controller views. In Listing 23-21, I have updated the Index page to remove the elements that will be provided by the layout.

Listing 23-21. Removing Elements in the Index.cshtml File in the Pages Folder

```
@page "{id:long?}"
@model IndexModel

<div class="bg-primary text-white text-center m-2 p-2">@Model.Product.Name</div>
```

Using a view start file applies the layout to all pages that don't override the value assigned to the `Layout` property. In Listing 23-22, I have added a code block to the `Editor` page so that it doesn't use a layout.

Listing 23-22. Disabling Layouts in the Editor.cshtml File in the Pages Folder

```
@page "{id:long}"
@model EditorModel
@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>

    <! ...elements omitted for brevity ... />

</body>
</html>
```

Use a browser to request `http://localhost:5000/index`, and you will see the effect of the new layout, which is shown on the left of Figure 23-13. Use the browser to request `http://localhost:5000/editor/1`, and you will receive content that is generated without the layout, as shown on the right of Figure 23-13.

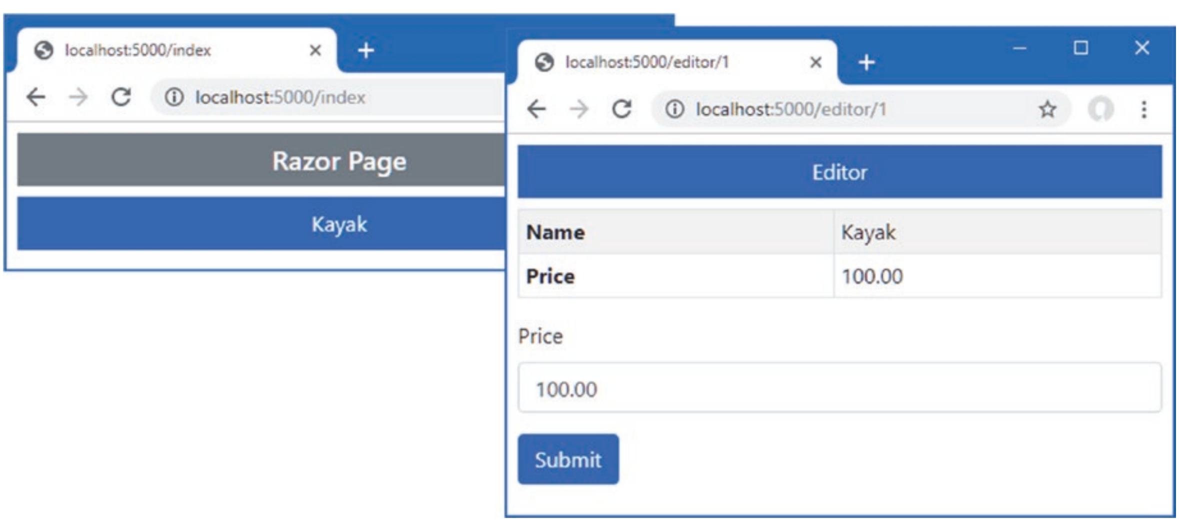

Figure 23-13. *Using a layout in Razor Pages*

Using Partial Views in Razor Pages

Razor Pages can use partial views so that common content isn't duplicated. The example in this section relies on the tag helpers feature, which I describe in detail in Chapter 25. For this chapter, add the directive shown in Listing 23-23 to the view imports file, which enables the custom HTML element used to apply partial views.

Listing 23-23. Enabling Tag Helpers in the _ViewImports.cshtml File in the Pages Folder

```
@namespace WebApp.Pages
@using WebApp.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Next, add a Razor view named `_ProductPartial.cshtml` in the `Pages/Shared` folder and add the content shown in Listing 23-24.

Listing 23-24. The Contents of the _ProductPartial.cshtml File in the Pages/Shared Folder

```
@model Product

<div class="m-2">
    <table class="table table-sm table-striped table-bordered">
        <tbody>
            <tr><th>Name</th><td>@Model.Name</td></tr>
            <tr><th>Price</th><td>@Model.Price</td></tr>
        </tbody>
    </table>
</div>
```

Notice there is nothing specific to Razor Pages in the partial view. Partial views use the `@model` directive to receive a view model object and do not use the `@page` directive or have page models, both of which are specific to Razor Pages. This allows Razor Pages to share partial views with MVC controllers, as described in the sidebar.

UNDERSTANDING THE PARTIAL METHOD SEARCH PATH

The Razor view engine starts looking for a partial view in the same folder as the Razor Page that uses it. If there is no matching file, then the search continues in each parent directory until the `Pages` folder is reached. For a partial view used by a Razor Page defined in the `Pages/App/Data` folder, for example, the view engine looks in the `Pages/App/Data` folder, the `Page/App` folder, and then the `Pages` folder. If no file is found, the search continues to the `Pages/Shared` folder and, finally, to the `Views/Shared` folder.

The last search location allows partial views defined for use with controllers to be used by Razor Pages, which is a useful feature for avoiding duplicate content in applications where MVC controllers and Razor Pages are both used.

Partial views are applied using `partial` element, as shown in Listing 23-25, with the `name` attribute specifying the name of the view and the `model` attribute providing the view model.

■ **Caution** Partial views receive a view model through their `@model` directive and not a page model. It is for this reason that the value of the model attribute is `Model.Product` and not just `Model`.

Listing 23-25. Using a Partial View in the Index.cshtml File in the Pages Folder

```
@page "{id:long?}"
@model IndexModel

<div class="bg-primary text-white text-center m-2 p-2">@Model.Product.Name</div>
<partial name="_ProductPartial" model="Model.Product" />
```

When the Razor Page is used to handle a response, the contents of the partial view are incorporated into the response. Use a browser to request `http://localhost:5000/index`, and the response includes the table defined in the partial view, as shown in Figure 23-14.

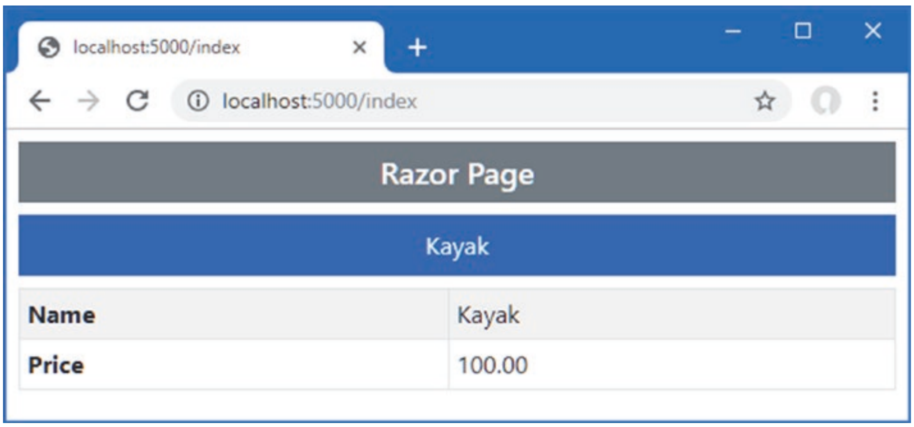

Figure 23-14. *Using a partial view*

Creating Razor Pages Without Page Models

If a Razor Page is simply presenting data to the user, the result can be a page model class that simply declares a constructor dependency to set a property that is consumed in the view. To understand this pattern, add a Razor Page named `Data.cshtml` to the `WebApp/Pages` folder with the content shown in Listing 23-26.

Listing 23-26. The Contents of the Data.cshtml File in the Pages Folder

```
@page
@model DataPageModel
@using Microsoft.AspNetCore.Mvc.RazorPages

<h5 class="bg-primary text-white text-center m-2 p-2">Categories</h5>
<ul class="list-group m-2">
    @foreach (Category c in Model.Categories) {
        <li class="list-group-item">@c.Name</li>
    }
</ul>

@functions {

    public class DataPageModel : PageModel {
        private DataContext context;

        public IEnumerable<Category> Categories { get; set; }

        public DataPageModel(DataContext ctx) {
            context = ctx;
        }
```

```
        public void OnGet() {
            Categories = context.Categories;
        }
    }
}
```

The page model in this example doesn't transform data, perform calculations, or do anything other than giving the view access to the data through dependency injection. To avoid this pattern, where a page model class is used only to access a service, the `@inject` directive can be used to obtain the service in the view, without the need for a page model, as shown in Listing 23-27.

■ **Caution** The `@inject` directive should be used sparingly and only when the page model class adds no value other than to provide access to services. In all other situations, using a page model class is easier to manage and maintain.

Listing 23-27. Accessing a Service in the Data.cshtml File in the Pages Folder

```
@page
@inject DataContext context;

<h5 class="bg-primary text-white text-center m-2 p-2">Categories</h5>
<ul class="list-group m-2">
    @foreach (Category c in context.Categories) {
        <li class="list-group-item">@c.Name</li>
    }
</ul>
```

The `@inject` expression specifies the service type and the name by which the service is accessed. In this example, the service type is `DataContext`, and the name by which it is accessed is `context`. Within the view, the `@foreach` expression generates elements for each object returned by the `DataContext.Categories` properties. Since there is no page model in this example, I have removed the `@page` and `@using` directives. Use a browser to navigate to `http://localhost:5000/data`, and you will see the response shown in Figure 23-15.

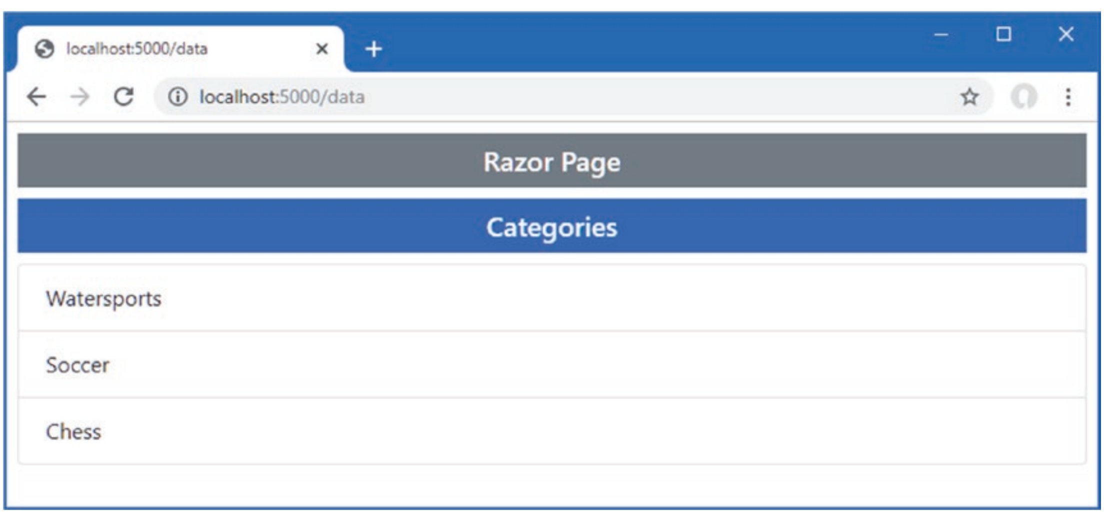

Figure 23-15. *Using a Razor Page without a page model*

Summary

In this chapter, I introduced Razor Pages and explained how they differ from the controllers and views. I showed you how to define content and code in the same file, how to use a code-behind file, and how page models provide the underpinnings for the most important Razor Pages features. In the next chapter, I describe the view components feature.

CHAPTER 24

Using View Components

I describe *view components* in this chapter, which are classes that provide action-style logic to support partial views; this means view components provide complex content to be embedded in views while allowing the C# code that supports it to be easily maintained. Table 24-1 puts view components in context.

Table 24-1. *Putting View Components in Context*

Question	Answer
What are they?	View components are classes that provide application logic to support partial views or to inject small fragments of HTML or JSON data into a parent view.
Why are they useful?	Without view components, it is hard to create embedded functionality such as shopping baskets or login panels in a way that is easy to maintain.
How are they used?	View components are typically derived from the `ViewComponent` class and are applied in a parent view using the custom `vc` HTML element or the `@await Component.InvokeAsync` expression.
Are there any pitfalls or limitations?	View components are a simple and predictable feature. The main pitfall is not using them and trying to include application logic within views where it is difficult to test and maintain.
Are there any alternatives?	You could put the data access and processing logic directly in a partial view, but the result is difficult to work with and hard to maintain.

Table 24-2 summarizes the chapter.

Table 24-2. *Chapter Summary*

Problem	Solution	Listing
Creating a reusable unit of code and content	Define a view component	7–13
Creating a response from a view component	Use one of the `IViewComponentResult` implementation classes	14–18
Getting context data	Use the properties inherited from the base class or use the parameters of the `Invoke` or `InvokeAsync` method	19–23
Generating view component responses asynchronously	Override the `InvokeAsync` method	24–26
Integrating a view component into another endpoint	Create a hybrid controller or Razor Page	27–34

Preparing for This Chapter

This chapter uses the WebApp project from Chapter 23. To prepare for this chapter, add a class file named `City.cs` to the `WebApp/Models` folder with the content shown in Listing 24-1.

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-3`. See Chapter 1 for how to get help if you have problems running the examples.

A. Freeman, *Pro ASP.NET Core 3*, https://doi.org/10.1007/978-1-4842-5440-0_24

Listing 24-1. The Contents of the City.cs File in the Models Folder

```
namespace WebApp.Models {

    public class City {
        public string Name { get; set; }
        public string Country { get; set; }
        public int Population { get; set; }
    }
}
```

Add a class named `CitiesData.cs` to the `WebApp/Models` folder with the content shown in Listing 24-2.

Listing 24-2. The Contents of the CitiesData.cs File in the WebApp/Models Folder

```
using System.Collections.Generic;

namespace WebApp.Models {

    public class CitiesData {

        private List<City> cities = new List<City> {
            new City { Name = "London", Country = "UK", Population = 8539000},
            new City { Name = "New York", Country = "USA", Population = 8406000 },
            new City { Name = "San Jose", Country = "USA", Population = 998537 },
            new City { Name = "Paris", Country = "France", Population = 2244000 }
        };

        public IEnumerable<City> Cities => cities;

        public void AddCity(City newCity) {
            cities.Add(newCity);
        }
    }
}
```

The `CitiesData` class provides access to a collection of `City` objects and provides an `AddCity` method that adds a new object to the collection. Add the statement shown in Listing 24-3 to the `ConfigureServices` method of the `Startup` class to create a service for the `CitiesData` class.

Listing 24-3. Defining a Service in the Startup.cs File in the WebApp Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using WebApp.Models;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace WebApp {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }
```

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

        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<DataContext>(opts => {
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:ProductConnection"]);
                opts.EnableSensitiveDataLogging(true);
            });
            services.AddControllersWithViews().AddRazorRuntimeCompilation();
            services.AddRazorPages().AddRazorRuntimeCompilation();

            services.AddDistributedMemoryCache();
            services.AddSession(options => {
                options.Cookie.IsEssential = true;
            });

            services.Configure<RazorPagesOptions>(opts => {
                opts.Conventions.AddPageRoute("/Index", "/extra/page/{id:long?}");
            });

            services.AddSingleton<CitiesData>();
        }

        public void Configure(IApplicationBuilder app, DataContext context) {
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseSession();
            app.UseRouting();
            app.UseEndpoints(endpoints => {
                endpoints.MapControllers();
                endpoints.MapDefaultControllerRoute();
                endpoints.MapRazorPages();
            });
            SeedData.SeedDatabase(context);
        }
    }
}
```

The new statement uses the AddSingleton method to create a CitiesData service. There is no interface/implementation separation in this service, which I have created to easily distribute a shared CitiesData object. Add a Razor Page named Cities.cshtml to the WebApp/Pages folder and add the content shown in Listing 24-4.

Listing 24-4. The Contents of the Cities.cshtml File in the Pages Folder

```
@page
@inject CitiesData Data

<div class="m-2">
    <table class="table table-sm table-striped table-bordered">
        <tbody>
            @foreach (City c in Data.Cities) {
                <tr>
                    <td>@c.Name</td>
                    <td>@c.Country</td>
                    <td>@c.Population</td>
                </tr>
            }
```

```
        </tbody>
    </table>
</div>
```

Dropping the Database

Open a new PowerShell command prompt, navigate to the folder that contains the `WebApp.csproj` file, and run the command shown in Listing 24-5 to drop the database.

Listing 24-5. Dropping the Database

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

Running the Example Application

Select Start Without Debugging or Run Without Debugging from the Debug menu or use the PowerShell command prompt to run the command shown in Listing 24-6.

Listing 24-6. Running the Example Application

```
dotnet run
```

The database will be seeded as part of the application startup. Once ASP.NET Core is running, use a web browser to request `http://localhost:5000/cities`, which will produce the response shown in Figure 24-1.

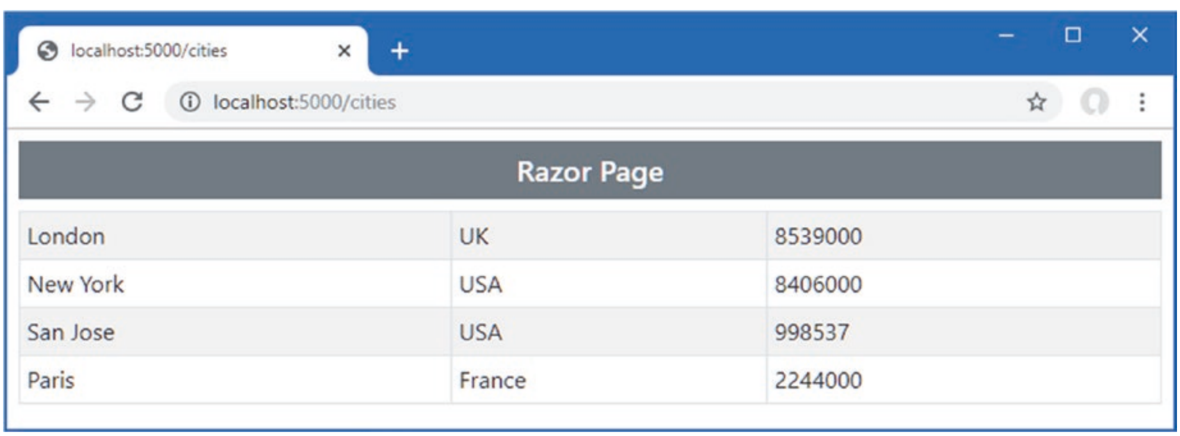

Figure 24-1. *Running the example application*

Understanding View Components

Applications commonly need to embed content in views that isn't related to the main purpose of the application. Common examples include site navigation tools and authentication panels that let the user log in without visiting a separate page.

The data for this type of feature isn't part of the model data passed from the action method or page model to the view. It is for this reason that I have created two sources of data in the example project: I am going to display some content generated using `City` data, which isn't easily done in a view that receives data from the Entity Framework Core repository and the Product, Category, and Supplier objects it contains.

Partial views are used to create reusable markup that is required in views, avoiding the need to duplicate the same content in multiple places in the application. Partial views are a useful feature, but they just contain fragments of HTML and Razor directives, and the data they operate on is received from the parent view. If you need to display different data, then you run into a problem. You could access the data you need directly from the partial view, but this breaks the development model and produces an application that is difficult to understand and maintain. Alternatively, you could extend the view models used by the application so that it includes the data you require, but this means you have to change every action method, which makes it hard to isolate the functionality of action methods for effective maintenance and testing.

This is where view components come in. A view component is a C# class that provides a partial view with the data that it needs, independently from the action method or Razor Page. In this regard, a view component can be thought of as a specialized action or page, but one that is used only to provide a partial view with data; it cannot receive HTTP requests, and the content that it provides will always be included in the parent view.

Creating and Using a View Component

A view component is any class whose name ends with `ViewComponent` and that defines an `Invoke` or `InvokeAsync` method or any class that is derived from the `ViewComponent` base class or that has been decorated with the `ViewComponent` attribute. I demonstrate the use of the attribute in the “Getting Context Data” section, but the other examples in this chapter rely on the base class.

View components can be defined anywhere in a project, but the convention is to group them in a folder named `Components`. Create the `WebApp/Components` folder and add to it a class file named `CitySummary.cs` with the content shown in Listing 24-7.

Listing 24-7. The Contents of the CitySummary.cs File in the Components Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using WebApp.Models;

namespace WebApp.Components {

    public class CitySummary: ViewComponent {
        private CitiesData data;

        public CitySummary(CitiesData cdata) {
            data = cdata;
        }

        public string Invoke() {
            return $"{data.Cities.Count()} cities, "
            + $"{data.Cities.Sum(c => c.Population)} people";
        }
    }
}
```

View components can take advantage of dependency injection to receive the services they require. In this example, the view component declares a dependency on the `CitiesData` class, which is then used in the `Invoke` method to create a `string` that contains the number of cities and the population total.

Applying a View Component

View components can be applied in two different ways. The first technique is to use the `Component` property that is added to the C# classes generated from views and Razor Pages. This property returns an object that implements the `IViewComponentHelper` interface, which provides the `InvokeAsync` method. Listing 24-8 uses this technique to apply the view component in the `Index.cshtml` file in the `Views/Home` folder.

Listing 24-8. Using a View Component in the Index.cshtml File in the Views/Index Folder

```
@model Product
@{
    Layout = "_Layout";
    ViewBag.Title = ViewBag.Title ?? "Product Table";
}

@section Header { Product Information }

<tr><th>Name</th><td>@Model.Name</td></tr>
<tr>
    <th>Price</th>
    <td>@Model.Price.ToString("c")</td>
</tr>
<tr><th>Category ID</th><td>@Model.CategoryId</td></tr>

@section Footer {
    @(((Model.Price / ViewBag.AveragePrice)
        * 100).ToString("F2"))% of average price
}

@section Summary {
    <div class="bg-info text-white m-2 p-2">
        @await Component.InvokeAsync("CitySummary")
    </div>
}
```

View components are applied using the `Component.InvokeAsync` method, using the name of the view component class as the argument. The syntax for this technique can be confusing. View component classes define either an `Invoke` or `InvokeAsync` method, depending on whether their work is performed synchronously or asynchronously. But the `Component.InvokeAsync` method is always used, even to apply view components that define the `Invoke` method and whose operations are entirely synchronous.

To add the namespace for the view components to the list that are included in views, I added the statement shown in Listing 24-9 to the `_ViewImports.json` file in the `Views` folder.

Listing 24-9. Adding a Namespace in the _ViewImports.json File in the Views Folder

```
@using WebApp.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@using WebApp.Components
```

Restart ASP.NET Core and use a browser to request `http://localhost:5000/home/index/1`, which will produce the result shown in Figure 24-2.

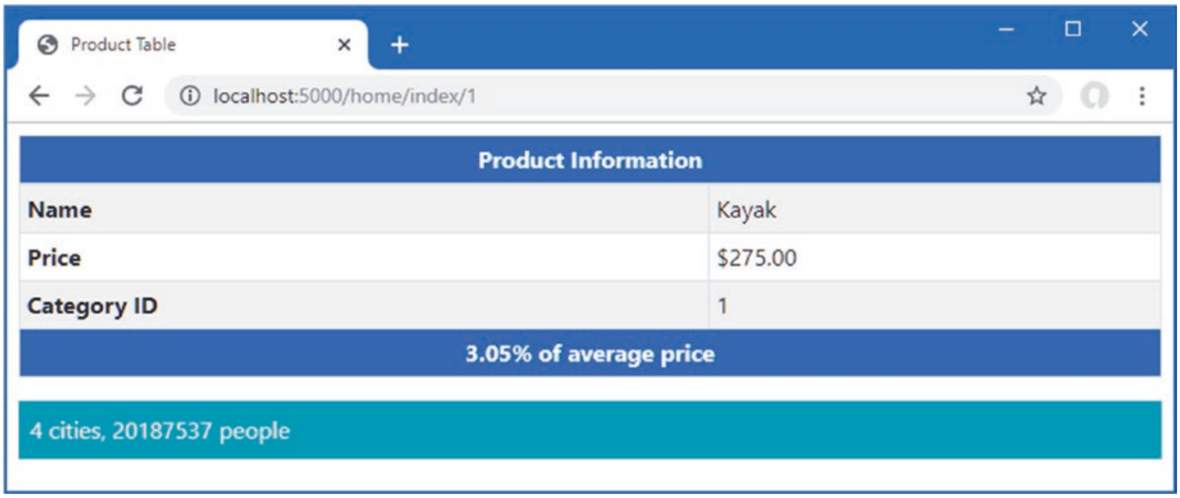

***Figure 24-2.** Using a view component*

Applying View Components Using a Tag Helper

Razor views and pages can contain tag helpers, which are custom HTML elements that are managed by C# classes. I explain how tag helpers work in detail in Chapter 25, but view components can be applied using an HTML element that is implemented as a tag helper. To enable this feature, add the directive shown in Listing 24-10 to the `_ViewImports.cshtml` file in the `Views` folder.

■ **Note** View components can be used only in controller views or Razor Pages and cannot be used to handle requests directly.

Listing 24-10. Configuring a Tag Helper in the _ViewImports.cshtml File in the Views Folder

```
@using WebApp.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@using WebApp.Components
@addTagHelper *, WebApp
```

The new directive adds tag helper support for the example project, which is specified by name. (You must change `WebApp` to the name of your project.) In Listing 24-11, I have used the custom HTML element to apply the view component.

Listing 24-11. Applying a View Component in the Index.cshtml File in the Views/Home Folder

```
@model Product
@{
    Layout = "_Layout";
    ViewBag.Title = ViewBag.Title ?? "Product Table";
}

@section Header { Product Information }

<tr><th>Name</th><td>@Model.Name</td></tr>
<tr>
    <th>Price</th>
    <td>@Model.Price.ToString("c")</td>
</tr>
```

```
<tr><th>Category ID</th><td>@Model.CategoryId</td></tr>

@section Footer {
    @(((Model.Price / ViewBag.AveragePrice)
        * 100).ToString("F2"))% of average price
}

@section Summary {
    <div class="bg-info text-white m-2 p-2">
        <vc:city-summary />
    </div>
}
```

The tag for the custom element is `vc`, followed by a colon, followed by the name of the view component class, which is transformed into kebab-case. Each capitalized word in the class name is converted to lowercase and separated by a hyphen so that `CitySummary` becomes `city-summary`, and the `CitySummary` view component is applied using the `vc:city-summary` element.

Applying View Components in Razor Pages

Razor Pages use view components in the same way, either through the `Component` property or through the custom HTML element. Since Razor Pages have their own view imports file, a separate `@addTagHelper` directive is required, as shown in Listing 24-12.

Listing 24-12. Adding a Directive in the _ViewImports.cshtml File in the Pages Folder

```
@namespace WebApp.Pages
@using WebApp.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, WebApp
```

Listing 24-13 applies the `CitySummary` view component to the `Data` page.

Listing 24-13. Using a View Component in the Data.cshtml File in the Pages Folder

```
@page
@inject DataContext context;

<h5 class="bg-primary text-white text-center m-2 p-2">Categories</h5>
<ul class="list-group m-2">
    @foreach (Category c in context.Categories) {
        <li class="list-group-item">@c.Name</li>
    }
</ul>

<div class="bg-info text-white m-2 p-2">
    <vc:city-summary />
</div>
```

Use a browser to request `http://localhost:5000/data`, and you will see the response shown in Figure 24-3, which displays the city data alongside the categories in the database.

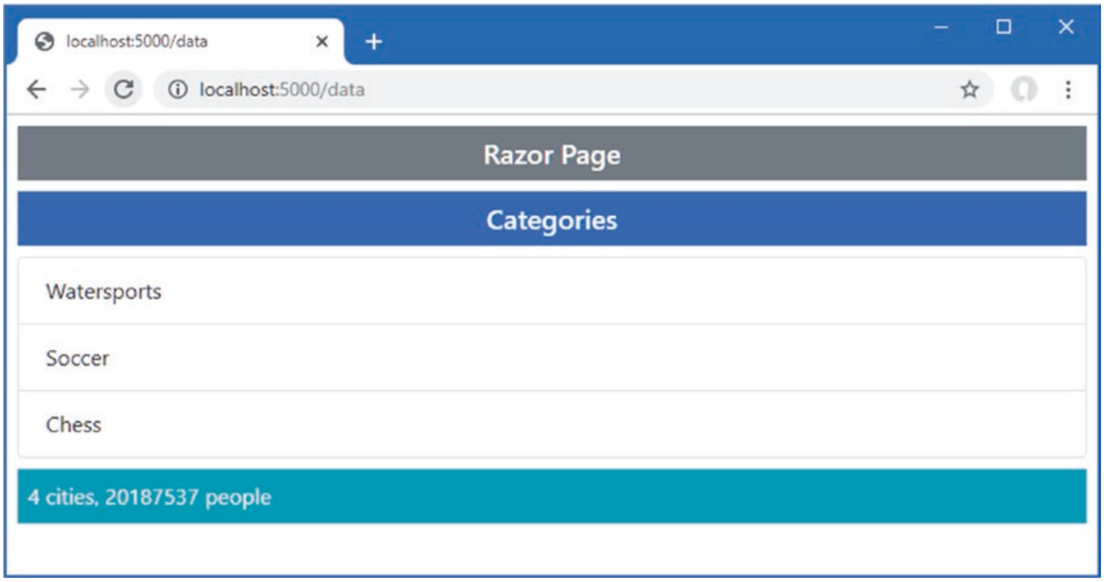

Figure 24-3. *Using a view component in a Razor Page*

Understanding View Component Results

The ability to insert simple string values into a view or page isn't especially useful, but fortunately, view components are capable of much more. More complex effects can be achieved by having the `Invoke` or `InvokeAsync` method return an object that implements the `IViewComponentResult` interface. There are three built-in classes that implement the `IViewComponentResult` interface, and they are described in Table 24-3, along with the convenience methods for creating them provided by the `ViewComponent` base class. I describe the use of each result type in the sections that follow.

Table 24-3. *The Built-in IViewComponentResult Implementation Classes*

Name	Description
`ViewViewComponentResult`	This class is used to specify a Razor view, with optional view model data. Instances of this class are created using the `View` method.
`ContentViewComponentResult`	This class is used to specify a text result that will be safely encoded for inclusion in an HTML document. Instances of this class are created using the `Content` method.
`HtmlContentViewComponentResult`	This class is used to specify a fragment of HTML that will be included in the HTML document without further encoding. There is no `ViewComponent` method to create this type of result.

There is special handling for two result types. If a view component returns a `string`, then it is used to create a `ContentViewComponentResult` object, which is what I relied on in earlier examples. If a view component returns an `IHtmlContent` object, then it is used to create an `HtmlContentViewComponentResult` object.

Returning a Partial View

The most useful response is the awkwardly named `ViewViewComponentResult` object, which tells Razor to render a partial view and include the result in the parent view. The `ViewComponent` base class provides the `View` method for creating `ViewViewComponentResult` objects, and four versions of the method are available, described in Table 24-4.

Table 24-4. The ViewComponent.View Methods

Name	Description
View()	Using this method selects the default view for the view component and does not provide a view model.
View(model)	Using the method selects the default view and uses the specified object as the view model.
View(viewName)	Using this method selects the specified view and does not provide a view model.
View(viewName, model)	Using this method selects the specified view and uses the specified object as the view model.

These methods correspond to those provided by the `Controller` base class and are used in much the same way. To create a view model class that the view component can use, add a class file named `CityViewModel.cs` to the `WebApp/Models` folder and use it to define the class shown in Listing 24-14.

Listing 24-14. The Contents of the CityViewModel.cs File in the Models Folder

```
namespace WebApp.Models {

    public class CityViewModel {
        public int Cities { get; set; }
        public int Population { get; set; }
    }
}
```

Listing 24-15 modifies the `Invoke` method of the `CitySummary` view component so it uses the `View` method to select a partial view and provides view data using a `CityViewModel` object.

Listing 24-15. Selecting a View in the CitySummary.cs File in the Components Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using WebApp.Models;

namespace WebApp.Components {

    public class CitySummary: ViewComponent {
        private CitiesData data;

        public CitySummary(CitiesData cdata) {
            data = cdata;
        }

        public IViewComponentResult Invoke() {
            return View(new CityViewModel {
                Cities = data.Cities.Count(),
                Population = data.Cities.Sum(c => c.Population)
            });
        }
    }
}
```

There is no view available for the view component currently, but the error message this produces reveals the locations that are searched. Restart ASP.NET Core and use a browser to request `http://localhost:5000/home/index/1` to see the locations that are searched when the view component is used with a controller. Request `http://localhost:5000/data` to see the locations searched when a view component is used with a Razor Page. Figure 24-4 shows both responses.

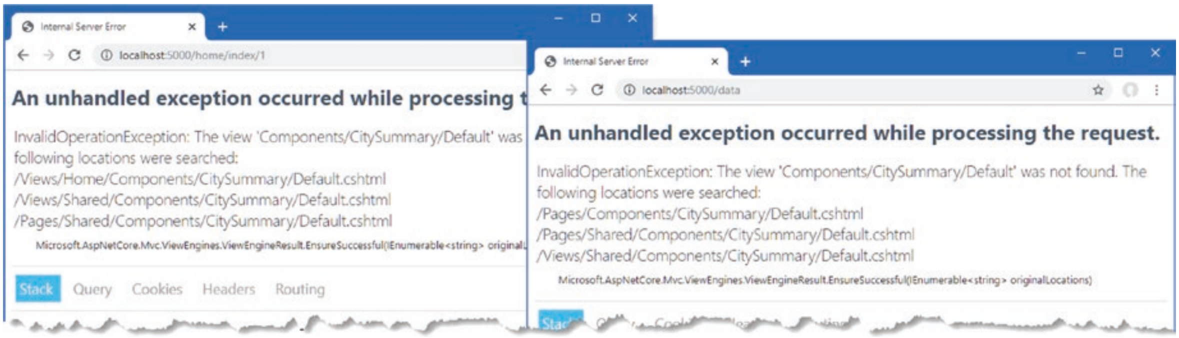

Figure 24-4. *The search locations for view component views*

Razor searches for a view named `Default.cshtml` when a view component invokes the `View` method without specifying a name. If the view component is used with a controller, then the search locations are as follows:

- `/Views/[controller]/Components/[viewcomponent]/Default.cshtml`
- `/Views/Shared/Components/[viewcomponent]/Default.cshtml`
- `/Pages/Shared/Components/[viewcomponent]/Default.cshtml`

When the `CitySummary` component is rendered by a view selected through the `Home` controller, for example, `[controller]` is `Home` and `[viewcomponent]` is `CitySummary`, which means the first search location is `/Views/Home/Components/CitySummary/Default.cshtml`. If the view component is used with a Razor Page, then the search locations are as follows:

- `/Pages/Components/[viewcomponent]/Default.cshtml`
- `/Pages/Shared/Components/[viewcomponent]/Default.cshtml`
- `/Views/Shared/Components/[viewcomponent]/Default.cshtml`

If the search paths for Razor Pages do not include the page name but a Razor Page is defined in a subfolder, then the Razor view engine will look for a view in the `Components/[viewcomponent]` folder, relative to the location in which the Razor Page is defined, working its way up the folder hierarchy until it finds a view or reaches the `Pages` folder.

■ **Tip** Notice that view components used in Razor Pages will find views defined in the `Views/Shared/Components` folder and that view components defined in controllers will find views in the `Pages/Shared/Components` folder. This means you don't have to duplicate views when a view component is used by controllers and Razor Pages.

Create the `WebApp/Views/Shared/Components/CitySummary` folder and add to it a Razor view named `Default.cshtml` with the content shown in Listing 24-16.

Listing 24-16. The Default.cshtml File in the Views/Shared/Components/CitySummary Folder

```
@model CityViewModel

<table class="table table-sm table-bordered text-white bg-secondary">
    <thead>
        <tr><th colspan="2">Cities Summary</th></tr>
    </thead>
    <tbody>
        <tr>
            <td>Cities:</td>
            <td class="text-right">
                @Model.Cities
            </td>
        </tr>
```

```
        <tr>
            <td>Population:</td>
            <td class="text-right">
                @Model.Population.ToString("#,###")
            </td>
        </tr>
    </tbody>
</table>
```

Views for view components are similar to partial views and use the `@model` directive to set the type of the view model object. This view receives a `CityViewModel` object from its view component, which is used to populate the cells in an HTML table. Use a browser to request `http://localhost:5000/home/index/1` and `http://localhost:5000/data`, and you will see the view incorporated into the responses, as shown in Figure 24-5.

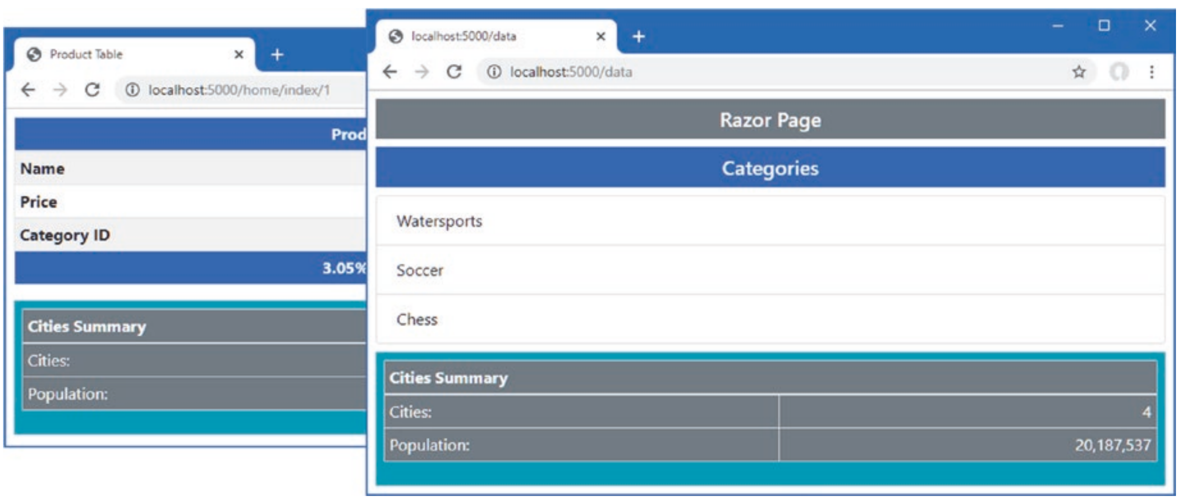

Figure 24-5. *Using a view with a view component*

Returning HTML Fragments

The `ContentViewComponentResult` class is used to include fragments of HTML in the parent view without using a view. Instances of the `ContentViewComponentResult` class are created using the `Content` method inherited from the `ViewComponent` base class, which accepts a `string` value. Listing 24-17 demonstrates the use of the `Content` method.

■ **Tip** In addition to the `Content` method, the `Invoke` method can return a `string`, which will be automatically converted to a `ContentViewComponentResult`. This is the approach I took in the view component when it was first defined.

Listing 24-17. Using the Content Method in the CitySummary.cs File in the Components Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using WebApp.Models;

namespace WebApp.Components {

    public class CitySummary: ViewComponent {
        private CitiesData data;
```

```
        public CitySummary(CitiesData cdata) {
            data = cdata;
        }

        public IViewComponentResult Invoke() {
            return Content("This is a <h3><i>string</i></h3>");
        }
    }
}
```

The string received by the Content method is encoded to make it safe to include in an HTML document. This is particularly important when dealing with content that has been provided by users or external systems because it prevents JavaScript content from being embedded into the HTML generated by the application.

In this example, the string that I passed to the Content method contains some basic HTML tags. Restart ASP.NET Core and use a browser to request http://localhost:5000/home/index/1. The response will include the encoded HTML fragment, as shown in Figure 24-6.

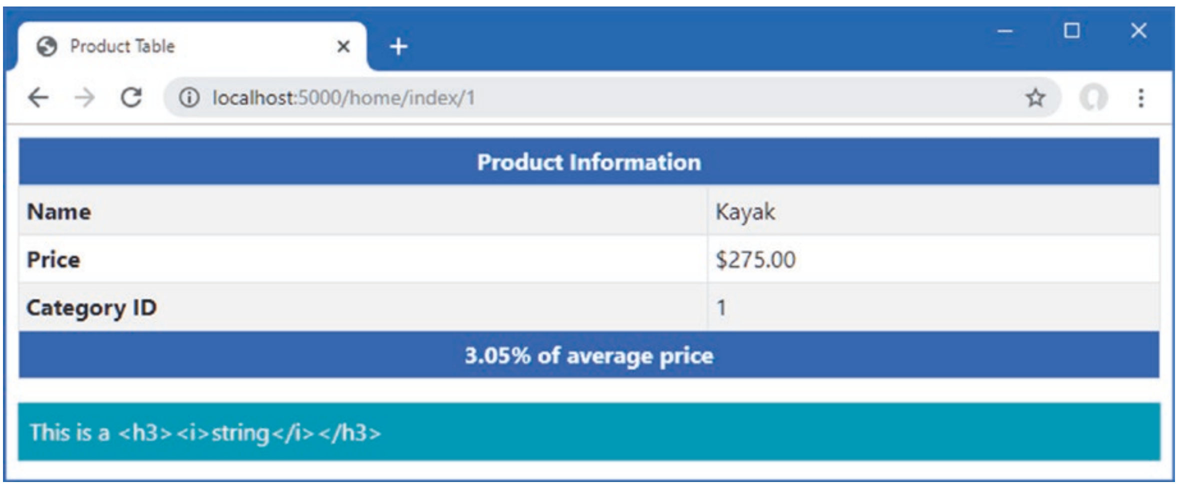

Figure 24-6. *Returning an encoded HTML fragment using a view component*

If you look at the HTML that the view component produced, you will see that the angle brackets have been replaced so that the browser doesn't interpret the content as HTML elements, as follows:

```
...
<div class="bg-info text-white m-2 p-2">
    This is a <h3><i>string</i></h3>
</div>
...
```

You don't need to encode content if you trust its source and want it to be interpreted as HTML. The Content method always encodes its argument, so you must create the HtmlContentViewComponentResult object directly and provide its constructor with an HtmlString object, which represents a string that you know is safe to display, either because it comes from a source that you trust or because you are confident that it has already been encoded, as shown in Listing 24-18.

Listing 24-18. Returning an HTML Fragment in the CitySummary.cs File in the Components Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using WebApp.Models;
using Microsoft.AspNetCore.Mvc.ViewComponents;
using Microsoft.AspNetCore.Html;
```

```
namespace WebApp.Components {

    public class CitySummary: ViewComponent {
        private CitiesData data;

        public CitySummary(CitiesData cdata) {
            data = cdata;
        }

        public IViewComponentResult Invoke() {
            return new HtmlContentViewComponentResult(
                new HtmlString("This is a <h3><i>string</i></h3>"));
        }
    }
}
```

This technique should be used with caution and only with sources of content that cannot be tampered with and that perform their own encoding. Restart ASP.NET Core and use a browser to request `http://localhost:5000/home/index/1`, and you will see the response isn't encoded and is interpreted as HTML elements, as shown in Figure 24-7.

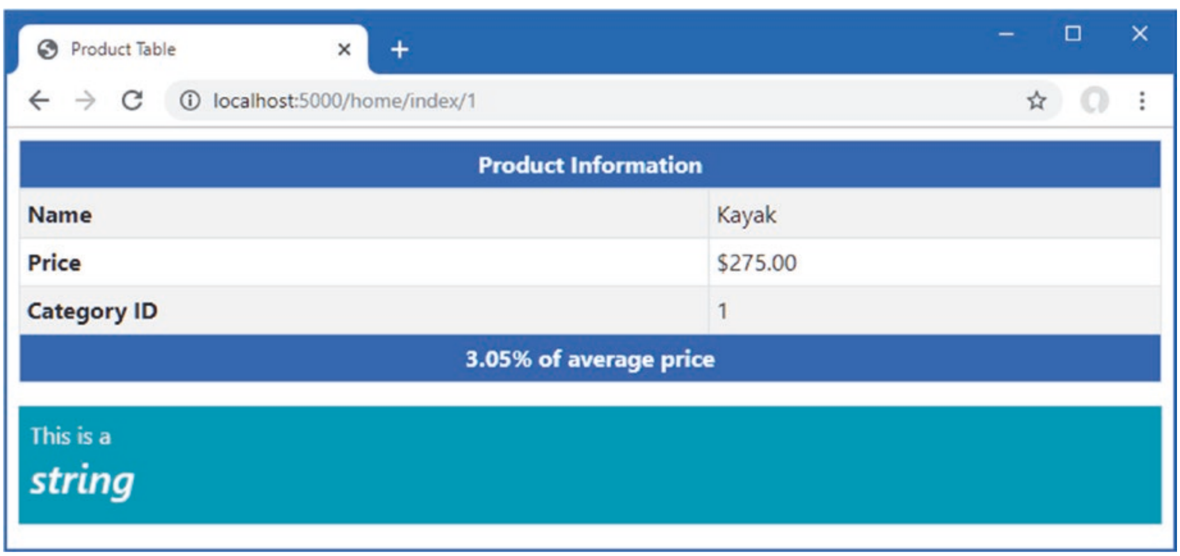

Figure 24-7. *Returning an unencoded HTML fragment using a view component*

Getting Context Data

Details about the current request and the parent view are provided to a view component through properties defined by the `ViewComponent` base class, as described in Table 24-5.

Table 24-5. *The ViewComponentContext Properties*

Name	Description
`HttpContext`	This property returns an `HttpContext` object that describes the current request and the response that is being prepared.
`Request`	This property returns an `HttpRequest` object that describes the current HTTP request.
`User`	This property returns an `IPrincipal` object that describes the current user, as described in Chapters 37 and 38.
`RouteData`	This property returns a `RouteData` object that describes the routing data for the current request.
`ViewBag`	This property returns the `dynamic` view bag object, which can be used to pass data between the view component and the view, as described in Chapter 22.
`ModelState`	This property returns a `ModelStateDictionary`, which provides details of the model binding process, as described in Chapter 29.
`ViewData`	This property returns a `ViewDataDictionary`, which provides access to the view data provided for the view component.

The context data can be used in whatever way helps the view component do its work, including varying the way that data is selected or rendering different content or views. It is hard to devise a representative example of using context data in a view component because the problems it solves are specific to each project. In Listing 24-19, I check the route data for the request to determine whether the routing pattern contains a controller segment variable, which indicates a request that will be handled by a controller and view.

Listing 24-19. Using Request Data in the CitySummary.cs File in the Components Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using WebApp.Models;
using Microsoft.AspNetCore.Mvc.ViewComponents;
using Microsoft.AspNetCore.Html;

namespace WebApp.Components {

    public class CitySummary: ViewComponent {
        private CitiesData data;

        public CitySummary(CitiesData cdata) {
            data = cdata;
        }

        public string Invoke() {
            if (RouteData.Values["controller"] != null) {
                return "Controller Request";
            } else {
                return "Razor Page Request";
            }
        }
    }
}
```

Restart ASP.NET Core and use a browser to request `http://localhost:5000/home/index/1` and `http://localhost:5000/data`, and you will see that the view component alters its output, as shown in Figure 24-8.

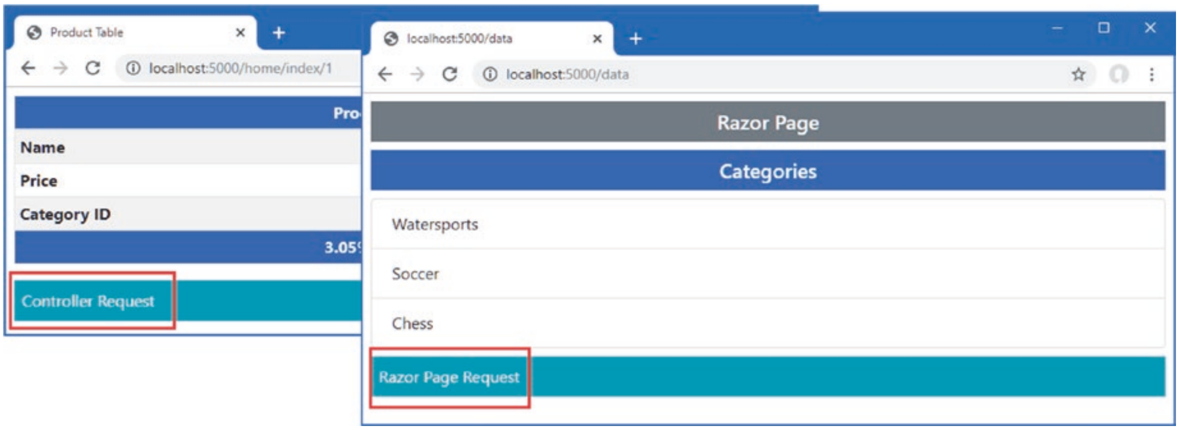

Figure 24-8. *Using context data in a view component*

Providing Context from the Parent View Using Arguments

Parent views can provide additional context data to view components, providing them with either data or guidance about the content that should be produced. The context data is received through the Invoke or InvokeAsync method, as shown in Listing 24-20.

Listing 24-20. Receiving a Value in the CitySummary.cs File in the Components Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using WebApp.Models;
using Microsoft.AspNetCore.Mvc.ViewComponents;
using Microsoft.AspNetCore.Html;

namespace WebApp.Components {

    public class CitySummary: ViewComponent {
        private CitiesData data;

        public CitySummary(CitiesData cdata) {
            data = cdata;
        }

        public IViewComponentResult Invoke(string themeName) {
            ViewBag.Theme = themeName;

            return View(new CityViewModel {
                Cities = data.Cities.Count(),
                Population = data.Cities.Sum(c => c.Population)
            });
        }
    }
}
```

The Invoke method defines a themeName parameter that is passed on to the partial view using the view bag, which was described in Chapter 22. Listing 24-21 updates the Default view to use the received value to style the content it produces.

Listing 24-21. Styling Content in the Default.cshtml File in the Views/Shared/Components/CitySummary Folder

```
@model CityViewModel

<table class="table table-sm table-bordered text-white bg-@ViewBag.Theme">
    <thead>
        <tr><th colspan="2">Cities Summary</th></tr>
    </thead>
    <tbody>
        <tr>
            <td>Cities:</td>
            <td class="text-right">
                @Model.Cities
            </td>
        </tr>
        <tr>
            <td>Population:</td>
            <td class="text-right">
                @Model.Population.ToString("#,###")
            </td>
        </tr>
    </tbody>
</table>
```

A value for all parameters defined by a view component's Invoke or InvokeAsync method must always be provided. Listing 24-22 provides a value for themeName parameter in the view selected by the Home controller.

■ **Tip** The view component will not be used if you do not provide values for all the parameters it defines but no error message is displayed. If you don't see any content from a view component, then the likely cause is a missing parameter value.

Listing 24-22. Supplying a Value in the Index.cshtml File in the Views/Home Folder

```
@model Product
@{
    Layout = "_Layout";
    ViewBag.Title = ViewBag.Title ?? "Product Table";
}

@section Header { Product Information }

<tr><th>Name</th><td>@Model.Name</td></tr>
<tr>
    <th>Price</th>
    <td>@Model.Price.ToString("c")</td>
</tr>
<tr><th>Category ID</th><td>@Model.CategoryId</td></tr>

@section Footer {
    @(((Model.Price / ViewBag.AveragePrice)
        * 100).ToString("F2"))% of average price
}

@section Summary {
    <div class="bg-info text-white m-2 p-2">
        <vc:city-summary theme-name="secondary" />
    </div>
}
```

The name of each parameter is expressed an attribute using kebab-case so that the theme-name attribute provides a value for the themeName parameter. Listing 24-23 sets a value in the Data.cshtml Razor Page.

Listing 24-23. Supplying a Value in the Data.cshtml File in the Pages Folder

```
@page
@inject DataContext context;

<h5 class="bg-primary text-white text-center m-2 p-2">Categories</h5>
<ul class="list-group m-2">
    @foreach (Category c in context.Categories) {
        <li class="list-group-item">@c.Name</li>
    }
</ul>

<div class="bg-info text-white m-2 p-2">
    <vc:city-summary theme-name="danger" />
</div>
```

Restart ASP.NET Core and use a browser to request http://localhost:5000/home/index/1 and http://localhost:5000/data. The view component is provided with different values for the themeName parameter, producing the responses shown in Figure 24-9.

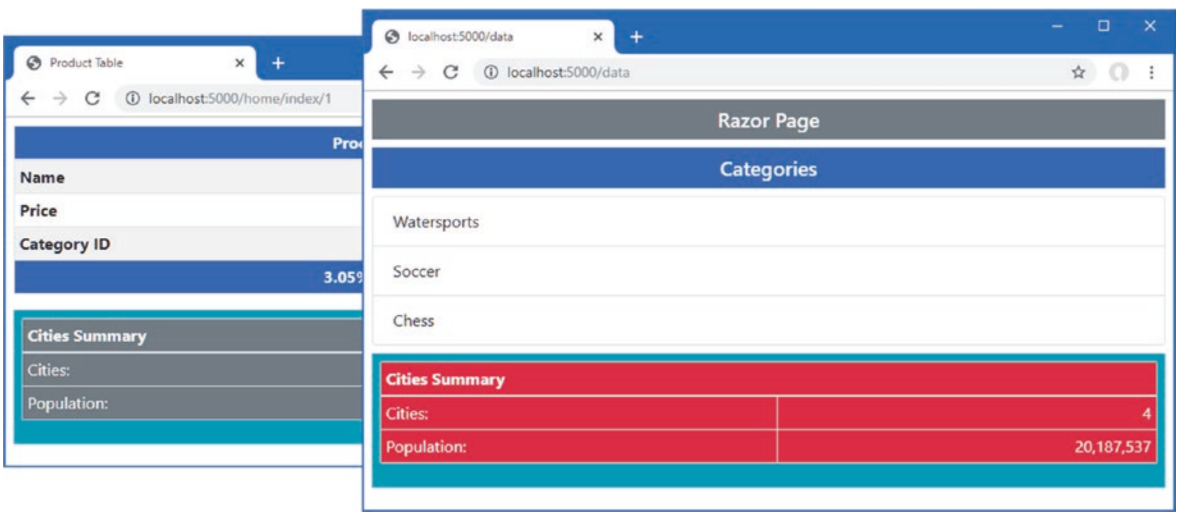

Figure 24-9. *Using context data in a view component*

PROVIDING VALUES USING THE COMPONENT HELPER

If you prefer applying view components using the Component.InvokeAsync helper, then you can provide context using method arguments, like this:

```
...
<div class="bg-info text-white m-2 p-2">
    @await Component.InvokeAsync("CitySummary", new { themeName = "danger" })
</div>
...
```

The first argument to the InvokeAsync method is the name of the view component class. The second argument is an object whose names correspond to the parameters defined by the view component.

Creating Asynchronous View Components

All the examples so far in this chapter have been synchronous view components, which can be recognized because they define the Invoke method. If your view component relies on asynchronous APIs, then you can create an asynchronous view component by defining an InvokeAsync method that returns a Task. When Razor receives the Task from the InvokeAsync method, it will wait for it to complete and then insert the result into the main view. To create a new component, add a class file named PageSize.cs to the Components folder and use it to define the class shown in Listing 24-24.

Listing 24-24. The Contents of the PageSize.cs File in the Components Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Net.Http;
using System.Threading.Tasks;

namespace WebApp.Components {

    public class PageSize : ViewComponent {

        public async Task<IViewComponentResult> InvokeAsync() {
            HttpClient client = new HttpClient();
            HttpResponseMessage response
                = await client.GetAsync("http://apress.com");
            return View(response.Content.Headers.ContentLength);
        }
    }
}
```

The InvokeAsync method uses the async and await keywords to consume the asynchronous API provided by the HttpClient class and get the length of the content returned by sending a GET request to Apress.com. The length is passed to the View method, which selects the default partial view associated with the view component.

Create the Views/Shared/Components/PageSize folder and add to it a Razor view named Default.cshtml with the content shown in Listing 24-25.

Listing 24-25. The Contents of the Default.cshtml File in the Views/Shared/Components/PageSize Folder

```
@model long
<div class="m-1 p-1 bg-light text-dark">Page size: @Model</div>
```

The final step is to use the component, which I have done in the Index view used by the Home controller, as shown in Listing 24-26. No change is required in the way that asynchronous view components are used.

Listing 24-26. Using an Asynchronous Component in the Index.cshtml File in the Views/Home Folder

```
@model Product
@{
    Layout = "_Layout";
    ViewBag.Title = ViewBag.Title ?? "Product Table";
}

@section Header { Product Information }

<tr><th>Name</th><td>@Model.Name</td></tr>
<tr>
    <th>Price</th>
    <td>@Model.Price.ToString("c")</td>
</tr>
<tr><th>Category ID</th><td>@Model.CategoryId</td></tr>
```

```
@section Footer {
    @(((Model.Price / ViewBag.AveragePrice)
        * 100).ToString("F2"))% of average price
}

@section Summary {
    <div class="bg-info text-white m-2 p-2">
        <vc:city-summary theme-name="secondary" />
        <vc:page-size />
    </div>
}
```

Restart ASP.NET Core and use a browser to request `http://localhost:5000/home/index/1`, which will produce a response that includes the size of the Apress.com home page, as shown in Figure 24-10. You may see a different number displayed since the Apress web site is updated frequently.

■ **Note** Asynchronous view components are useful when there are several different regions of content to be created, each of which can be performed concurrently. The response isn't sent to the browser until all the content is ready. If you want to update the content presented to the user dynamically, then you can use Blazor, as described in Part 4.

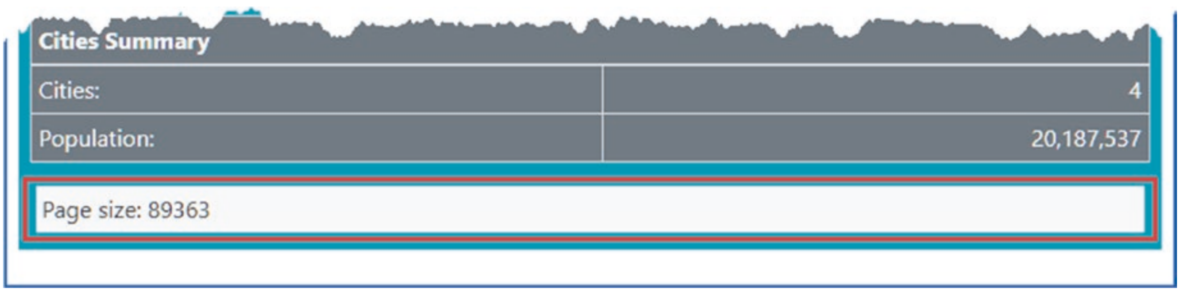

Figure 24-10. *Using an asynchronous component*

Creating View Components Classes

View components often provide a summary or snapshot of functionality that is handled in-depth by a controller or Razor Page. For a view component that summarizes a shopping basket, for example, there will often be a link that targets a controller that provides a detailed list of the products in the basket and that can be used to check out and complete the purchase.

In this situation, you can create a class that is a view component as well as a controller or Razor Page. If you are using Visual Studio, expand the `Cities.cshtml` item in the Solution Explorer to show the `Cities.cshtml.cs` file and replace its contents with those shown in Listing 24-27. If you are using Visual Studio Code, add a file named `Cities.cshtml.cs` to the `Pages` folder with the content shown in Listing 24-27.

Listing 24-27. The Contents of the Cities.cshtml.cs File in the Pages Folder

```
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.ViewComponents;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using WebApp.Models;
```

```
namespace WebApp.Pages {

    [ViewComponent(Name = "CitiesPageHybrid")]
    public class CitiesModel : PageModel {

        public CitiesModel(CitiesData cdata) {
            Data = cdata;
        }

        public CitiesData Data { get; set; }

        [ViewComponentContext]
        public ViewComponentContext Context { get; set; }

        public IViewComponentResult Invoke() {
            return new ViewViewComponentResult() {
                ViewData = new ViewDataDictionary<CityViewModel>(
                    Context.ViewData,
                    new CityViewModel {
                        Cities = Data.Cities.Count(),
                        Population = Data.Cities.Sum(c => c.Population)
                    })
            };
        }
    }
}
```

This page model class is decorated with the ViewComponent attribute, which allows it to be used as a view component. The Name argument specifies the name by which the view component will be applied. Since a page model cannot inherit from the ViewComponent base class, a property whose type is ViewComponentContext is decorated with the ViewComponentContext attribute, which signals that it should be assigned an object that defines the properties described in Table 24-5 before the Invoke or InvokeAsync method is invoked. The View method isn't available, so I have to create a ViewViewComponentResult object, which relies on the context object received through the decorated property. Listing 24-28 updates the view part of the page to use the new page model class.

Listing 24-28. Updating the View in the Cities.cshtml File in the Pages Folder

```
@page
@model WebApp.Pages.CitiesModel

<div class="m-2">
    <table class="table table-sm table-striped table-bordered">
        <tbody>
            @foreach (City c in Model.Data.Cities) {
                <tr>
                    <td>@c.Name</td>
                    <td>@c.Country</td>
                    <td>@c.Population</td>
                </tr>
            }
        </tbody>
    </table>
</div>
```

The changes update the directives to use the page model class. To create the view for the hybrid view component, create the Pages/Shared/Components/CitiesPageHybrid folder and add to it a Razor view named Default.cshtml with the content shown in Listing 24-29.

Listing 24-29. The Default.cshtml File in the Pages/Shared/Components/CitiesPageHybrid Folder

```
@model CityViewModel

<table class="table table-sm table-bordered text-white bg-dark">
    <thead><tr><th colspan="2">Hybrid Page Summary</th></tr></thead>
    <tbody>
        <tr>
            <td>Cities:</td>
            <td class="text-right">@Model.Cities</td>
        </tr>
        <tr>
            <td>Population:</td>
            <td class="text-right">
                @Model.Population.ToString("#,###")
            </td>
        </tr>
    </tbody>
</table>
```

Listing 24-30 applies the view component part of the hybrid class in another page.

Listing 24-30. Using a View Component in the Data.cshtml File in the Pages Folder

```
@page
@inject DataContext context;

<h5 class="bg-primary text-white text-center m-2 p-2">Categories</h5>
<ul class="list-group m-2">
    @foreach (Category c in context.Categories) {
        <li class="list-group-item">@c.Name</li>
    }
</ul>

<div class="bg-info text-white m-2 p-2">
    <vc:cities-page-hybrid  />
</div>
```

Hybrids are applied just like any other view component. Restart ASP.NET Core and request `http://localhost:5000/cities` and `http://localhost:5000/data`. Both URLs are processed by the same class. For the first URL, the class acts as a page model; for the second URL, the class acts as a view component. Figure 24-11 shows the output for both URLs.

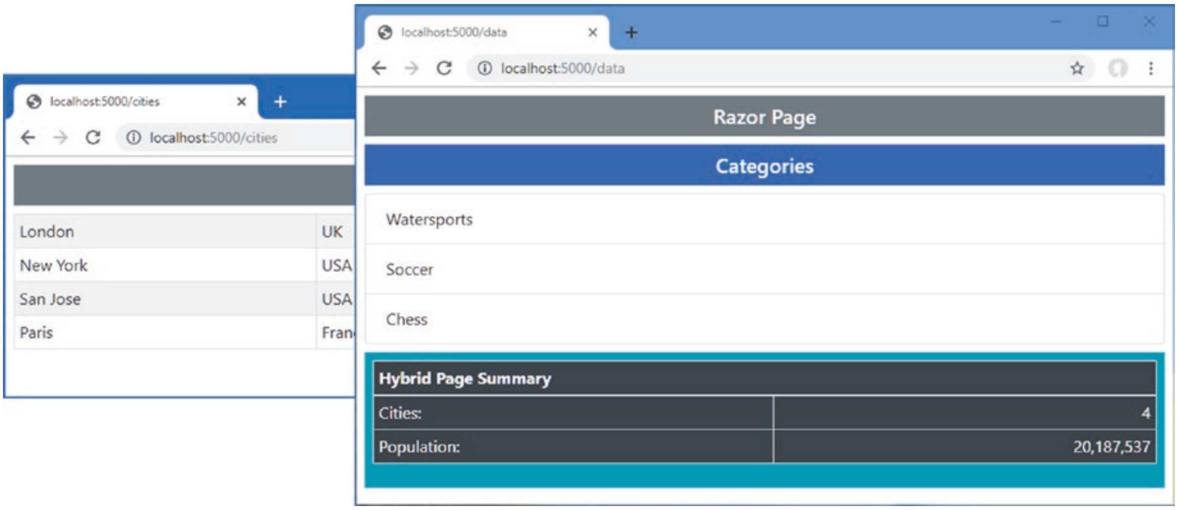

Figure 24-11. *A hybrid page model and view component class*

Creating a Hybrid Controller Class

The same technique can be applied to controllers. Add a class file named `CitiesController.cs` to the `Controllers` folder and add the statements shown in Listing 24-31.

Listing 24-31. The Contents of the CitiesController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.ViewComponents;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using System.Linq;
using WebApp.Models;

namespace WebApp.Controllers {

    [ViewComponent(Name = "CitiesControllerHybrid")]
    public class CitiesController: Controller {
        private CitiesData data;

        public CitiesController(CitiesData cdata) {
            data = cdata;
        }

        public IActionResult Index() {
            return View(data.Cities);
        }

        public IViewComponentResult Invoke() {
            return new ViewViewComponentResult() {
                ViewData = new ViewDataDictionary<CityViewModel>(
                    ViewData,
                    new CityViewModel {
                        Cities = data.Cities.Count(),
                        Population = data.Cities.Sum(c => c.Population)
                    })
            };
        }
    }
}
```

A quirk in the way that controllers are instantiated means that a property decorated with the `ViewComponentContext` attribute isn't required and the `ViewData` property inherited from the `Controller` base class can be used to create the view component result.

To provide a view for the action method, create the `Views/Cities` folder and add to it a file named `Index.cshtml` with the content shown in Listing 24-32.

Listing 24-32. The Contents of the Index.cshtml File in the Views/Cities Folder

```
@model IEnumerable<City>
@{
    Layout = "_ImportantLayout";
}

<div class="m-2">
    <table class="table table-sm table-striped table-bordered">
        <tbody>
            @foreach (City c in Model) {
                <tr>
                    <td>@c.Name</td>
                    <td>@c.Country</td>
                    <td>@c.Population</td>
                </tr>
            }
        </tbody>
    </table>
</div>
```

To provide a view for the view component, create the `Views/Shared/Components/CitiesControllerHybrid` folder and add to it a Razor view named `Default.cshtml` with the content shown in Listing 24-33.

Listing 24-33. The Default.cshtml File in the Views/Shared/Components/CitiesControllerHybrid Folder

```
@model CityViewModel
<table class="table table-sm table-bordered text-white bg-dark">
    <thead><tr><th colspan="2">Hybrid Controller Summary</th></tr></thead>
    <tbody>
        <tr>
            <td>Cities:</td>
            <td class="text-right">@Model.Cities</td>
        </tr>
        <tr>
            <td>Population:</td>
            <td class="text-right">
                @Model.Population.ToString("#,###")
            </td>
        </tr>
    </tbody>
</table>
```

Listing 24-34 applies the hybrid view component in the `Data.cshtml` Razor Page, replacing the hybrid class created in the previous section.

Listing 24-34. Applying the View Component in the Data.cshtml File in the Pages Folder

```
@page
@inject DataContext context;

<h5 class="bg-primary text-white text-center m-2 p-2">Categories</h5>
<ul class="list-group m-2">
```

```
    @foreach (Category c in context.Categories) {
        <li class="list-group-item">@c.Name</li>
    }
</ul>

<div class="bg-info text-white m-2 p-2">
    <vc:cities-controller-hybrid />
</div>
```

Restart ASP.NET Core and use a browser to request `http://localhost:5000/cities/index` and `http://localhost:5000/data`. For the first URL, the class in Listing 24-34 is used as a controller; for the second URL, the class is used as a view component. Figure 24-12 shows the responses for both URLs.

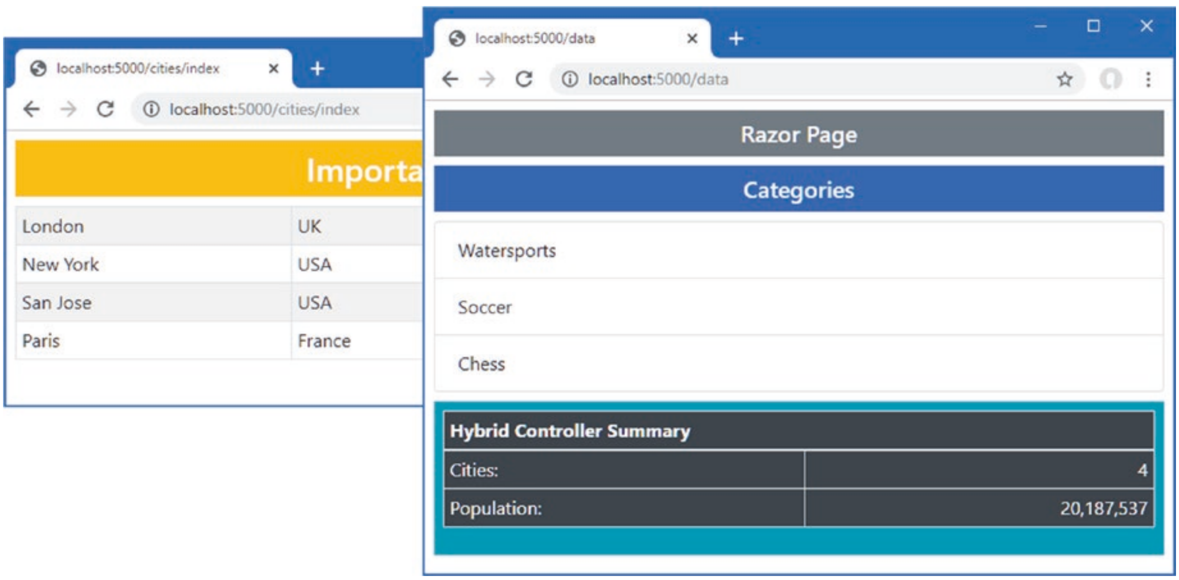

***Figure 24-12.** A hybrid controller and view component class*

Summary

In this chapter, I described the view components feature, which allows orthogonal features to be included in views used by controllers or Razor Pages. I explained how view components work and how they are applied, and I demonstrated the different types of results they produce. I completed the chapter by showing you how to create classes that are both view components and controllers or Razor Pages. In the next chapter, I introduce tag helpers. which are used to transform HTML elements.

CHAPTER 25

Using Tag Helpers

Tag helpers are C# classes that transform HTML elements in a view or page. Common uses for tag helpers include generating URLs for forms using the application's routing configuration, ensuring that elements of a specific type are styled consistently, and replacing custom shorthand elements with commonly used fragments of content. In this chapter, I describe how tag helpers work and how custom tag helpers are created and applied. In Chapter 26, I describe the built-in tag helpers, and in Chapter 27, I use tag helpers to explain how HTML forms are created. Table 25-1 puts tag helpers in context.

Table 25-1. *Putting Tag Helpers in Context*

Question	Answer
What are they?	Tag helpers are classes that manipulate HTML elements, either to change them in some way, to supplement them with additional content, or to replace them entirely with new content.
Why are they useful?	Tag helpers allow view content to be generated or transformed using C# logic, ensuring that the HTML sent to the client reflects the state of the application.
How are they used?	The HTML elements to which tag helpers are applied are selected based on the name of the class or with the `HTMLTargetElement` attribute. When a view is rendered, elements are transformed by tag helpers and included in the HTML sent to the client.
Are there any pitfalls or limitations?	It can be easy to get carried away and generate complex sections of HTML content using tag helpers, which is something that is more readily achieved using view components, described in Chapter 24.
Are there any alternatives?	You don't have to use tag helpers, but they make it easy to generate complex HTML in ASP.NET Core applications.

Table 25-2 summarizes the chapter.

Table 25-2. *Chapter Summary*

Problem	Solution	Listing
Creating a tag helper	Define a class that is derived from the `TagHelper` class	1–7
Controlling the scope of a tag helper	Alter the range of elements specified by the `HtmlTargetElement` attribute	8–11
Creating custom HTML elements that are replaced with content	Use shorthand elements	12, 13
Creating elements programmatically	Use the `TagBuilder` class	14
Controlling where content is inserted	Use the prepend and append features	15–18
Getting context data	Use the context object	19, 20
Operating on the view model or page model	Use a model expression	21–24
Creating coordinating tag helpers	Use the `Items` property	25–26
Suppressing content	Use the `SuppressOutput` method	27, 28
Defining tag helper as services	Create tag helper components	29–32

A. Freeman, *Pro ASP.NET Core 3*, https://doi.org/10.1007/978-1-4842-5440-0_25

Preparing for This Chapter

This chapter uses the WebApp project from Chapter 24. To prepare for this chapter, replace the contents of the Startup.cs file with those in Listing 25-1, removing some of the configuration statements used in earlier chapters.

■ **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-3. See Chapter 1 for how to get help if you have problems running the examples.

Listing 25-1. The Contents of the Startup.cs File in the WebApp Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using WebApp.Models;

namespace WebApp {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        public IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<DataContext>(opts => {
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:ProductConnection"]);
                opts.EnableSensitiveDataLogging(true);
            });
            services.AddControllersWithViews().AddRazorRuntimeCompilation();
            services.AddRazorPages().AddRazorRuntimeCompilation();
            services.AddSingleton<CitiesData>();
        }

        public void Configure(IApplicationBuilder app, DataContext context) {
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseRouting();
            app.UseEndpoints(endpoints => {
                endpoints.MapControllers();
                endpoints.MapDefaultControllerRoute();
                endpoints.MapRazorPages();
            });
            SeedData.SeedDatabase(context);
        }
    }
}
```

Next, replace the contents of the Index.cshtml file in the Views/Home folder with the content shown in Listing 25-2.

Listing 25-2. The Contents of the Index.cshtml File in the Views/Home Folder

```
@model Product
@{
    Layout = "_SimpleLayout";
}

<table class="table table-striped table-bordered table-sm">
    <thead>
        <tr>
            <th colspan="2">Product Summary</th>
        </tr>
    </thead>
    <tbody>
        <tr><th>Name</th><td>@Model.Name</td></tr>
        <tr>
            <th>Price</th>
            <td>@Model.Price.ToString("c")</td>
        </tr>
        <tr><th>Category ID</th><td>@Model.CategoryId</td></tr>
    </tbody>
</table>
```

The view in Listing 25-2 relies on a new layout. Add a Razor view file named `_SimpleLayout.cshtml` in the `Views/Shared` folder with the content shown in Listing 25-3.

Listing 25-3. The Contents of the _SimpleLayout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="m-2">
        @RenderBody()
    </div>
</body>
</html>
```

Dropping the Database

Open a new PowerShell command prompt, navigate to the folder that contains the `WebApp.csproj` file, and run the command shown in Listing 25-4 to drop the database.

Listing 25-4. Dropping the Database

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

Running the Example Application

Select Start Without Debugging or Run Without Debugging from the Debug menu or use the PowerShell command prompt to run the command shown in Listing 25-5.

Listing 25-5. Running the Example Application

```
dotnet run
```

Use a browser to request `http://localhost:5000/home`, which will produce the response shown in Figure 25-1.

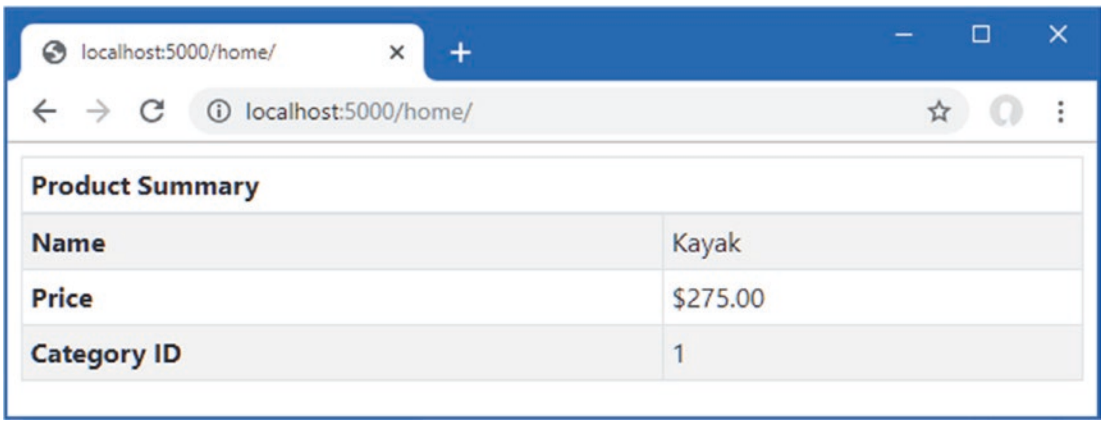

Figure 25-1. *Running the example application*

Creating a Tag Helper

The best way to understand tag helpers is to create one, which reveals how they operate and how they fit into an ASP.NET Core application. In the sections that follow, I go through the process of creating and applying a tag helper that will set the Bootstrap CSS classes for a `tr` element so that an element like this:

```
...
<tr tr-color="primary">
    <th colspan="2">Product Summary</th>
</tr>
...
```

will be transformed into this:

```
...
<tr class="bg-primary text-white text-center">
    <th colspan="2">Product Summary</th>
</tr>
...
```

The tag helper will recognize the `tr-color` attribute and use its value to set the `class` attribute on the element sent to the browser. This isn't the most dramatic—or useful—transformation, but it provides a foundation for explaining how tag helpers work.

Defining the Tag Helper Class

Tag helpers can be defined anywhere in the project, but it helps to keep them together because they need to be registered before they can be used. Create the WebApp/TagHelpers folder and add to it a class file named TrTagHelper.cs with the code shown in Listing 25-6.

Listing 25-6. The Contents of the TrTagHelper.cs File in the TagHelpers Folder

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace WebApp.TagHelpers {

    public class TrTagHelper: TagHelper {

        public string BgColor { get; set; } = "dark";
        public string TextColor { get; set; } = "white";

        public override void Process(TagHelperContext context,
                TagHelperOutput output) {
            output.Attributes.SetAttribute("class",
                $"bg-{BgColor} text-center text-{TextColor}");
        }
    }
}
```

Tag helpers are derived from the TagHelper class, which is defined in the Microsoft.AspNetCore.Razor.TagHelpers namespace. The TagHelper class defines a Process method, which is overridden by subclasses to implement the behavior that transforms elements.

The name of the tag helper combines the name of the element it transforms followed by TagHelper. In the case of the example, the class name TrTagHelper indicates this is a tag helper that operates on tr elements. The range of elements to which a tag helper can be applied can be broadened or narrowed using attributes, as described later in this chapter, but the default behavior is defined by the class name.

■ **Tip** Asynchronous tag helpers can be created by overriding the ProcessAsync method instead of the Process method, but this isn't required for most helpers, which tend to make small and focused changes to HTML elements. You can see an example of an asynchronous tag helper in the "Advanced Tag Helper Features" section.

Receiving Context Data

Tag helpers receive information about the element they are transforming through an instance of the TagHelperContext class, which is received as an argument to the Process method and which defines the properties described in Table 25-3.

Table 25-3. The TagHelperContext Properties

Name	Description
`AllAttributes`	This property returns a read-only dictionary of the attributes applied to the element being transformed, indexed by name and by index.
`Items`	This property returns a dictionary that is used to coordinate between tag helpers, as described in the "Coordinating Between Tag Helpers" section.
`UniqueId`	This property returns a unique identifier for the element being transformed.

Although you can access details of the element's attributes through the `AllAttributes` dictionary, a more convenient approach is to define a property whose name corresponds to the attribute you are interested in, like this:

```
...
public string BgColor { get; set; } = "dark";
public string TextColor { get; set; } = "white";
...
```

When a tag helper is being used, the properties it defines are inspected and assigned the value of any whose name matches attributes applied to the HTML element. As part of this process, the attribute value will be converted to match the type of the C# property so that `bool` properties can be used to receive `true` and `false` attribute values and `int` properties can be used to receive numeric attribute values such as `1` and `2`.

Properties for which there are no corresponding HTML element attributes are not set, which means you should check to ensure that you are not dealing with `null` or provide default values, which is the approach taken in Listing 25-6.

The name of the attribute is automatically converted from the default HTML style, `bg-color`, to the C# style, `BgColor`. You can use any attribute prefix except `asp-` (which Microsoft uses) and `data-` (which is reserved for custom attributes that are sent to the client). The example tag helper will be configured using `bg-color` and `text-color` attributes, which will provide values for the `BgColor` and `TextColor` properties and be used to configure the `tr` element in the `Process` method, as follows:

```
...
output.Attributes.SetAttribute("class",
    $"bg-{BgColor} text-center text-{TextColor}");
...
```

■ **Tip** Using the HTML attribute name for tag helper properties doesn't always lead to readable or understandable classes. You can break the link between the name of the property and the attribute it represents using the `HtmlAttributeName` attribute, which can be used to specify the HTML attribute that the property represents.

Producing Output

The `Process` method transforms an element by configuring the `TagHelperOutput` object that is received as an argument. The `TagHelperOuput` object starts by describing the HTML element as it appears in the view and is modified through the properties and methods described in Table 25-4.

Table 25-4. *The TagHelperOutput Properties and Methods*

Name	Description
`TagName`	This property is used to get or set the tag name for the output element.
`Attributes`	This property returns a dictionary containing the attributes for the output element.
`Content`	This property returns a `TagHelperContent` object that is used to set the content of the element.
`GetChildContentAsync()`	This asynchronous method provides access to the content of the element that will be transformed, as demonstrated in the “Creating Shorthand Elements” section.
`PreElement`	This property returns a `TagHelperContext` object that is used to insert content in the view before the output element. See the “Prepending and Appending Content and Elements” section.
`PostElement`	This property returns a `TagHelperContext` object that is used to insert content in the view after the output element. See the “Prepending and Appending Content and Elements” section.
`PreContent`	This property returns a `TagHelperContext` object that is used to insert content before the output element’s content. See the “Prepending and Appending Content and Elements” section.
`PostContent`	This property returns a `TagHelperContext` object that is used to insert content after the output element’s content. See the “Prepending and Appending Content and Elements” section.
`TagMode`	This property specifies how the output element will be written, using a value from the `TagMode` enumeration. See the “Creating Shorthand Elements” section.
`SupressOuput()`	Calling this method excludes an element from the view. See the “Suppressing the Output Element” section.

In the `TrTagHelper` class, I used the `Attributes` dictionary to add a `class` attribute to the HTML element that specifies Bootstrap styles, including the value of the `BgColor` and `TextColor` properties. The effect is that the background color for `tr` elements can be specified by setting `bg-color` and `text-color` attributes to Bootstrap names, such as `primary`, `info`, and `danger`.

Registering Tag Helpers

Tag helper classes must be registered with the `@addTagHelper` directive before they can be used. The set of views or pages to which a tag helper can be applied depends on where the `@addTagHelper` directive is used.

For a single view or page, the directive appears in the CSHTML file itself. To make a tag helper available more widely, it can be added to the view imports file, which is defined in the `Views` folder for controllers and the `Pages` folder for Razor Pages.

I want the tag helpers that I create in this chapter to be available anywhere in the application, which means that the `@addTagHelper` directive is added to the `_ViewImports.cshtml` files in the `Views` and `Pages` folders. The `vc` element used in Chapter 24 to apply view components is a tag helper, which is why the directive required to enable tag helpers is already in the `_ViewImports.cshtml` file.

```
@using WebApp.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@using WebApp.Components
@addTagHelper *, WebApp
```

The first part of the argument specifies the names of the tag helper classes, with support for wildcards, and the second part specifies the name of the assembly in which they are defined. This `@addTagHelper` directive uses the wildcard to select all namespaces in the `WebApp` assembly, with the effect that tag helpers defined anywhere in the project can be used in any controller view. There is an identical statement in the Razor Pages `_ViewImports.cshtml` file in the `Pages` folder.

```
@namespace WebApp.Pages
@using WebApp.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, WebApp
```

The other `@addTagHelper` directive enables the built-in tag helpers that Microsoft provides, which are described in Chapter 26.

Using a Tag Helper

The final step is to use the tag helper to transform an element. In Listing 25-7, I have added the attribute to the tr element, which will apply the tag helper.

Listing 25-7. Using a Tag Helper in the Index.cshtml File in the Views/Home Folder

```
@model Product
@{
    Layout = "_SimpleLayout";
}

<table class="table table-striped table-bordered table-sm">
    <thead>
        <tr bg-color="info" text-color="white">
            <th colspan="2">Product Summary</th>
        </tr>
    </thead>
    <tbody>
        <tr><th>Name</th><td>@Model.Name</td></tr>
        <tr>
            <th>Price</th>
            <td>@Model.Price.ToString("c")</td>
        </tr>
        <tr><th>Category ID</th><td>@Model.CategoryId</td></tr>
    </tbody>
</table>
```

Restart ASP.NET Core and use a browser to request http://localhost:5000/home, which produces the response shown in Figure 25-2.

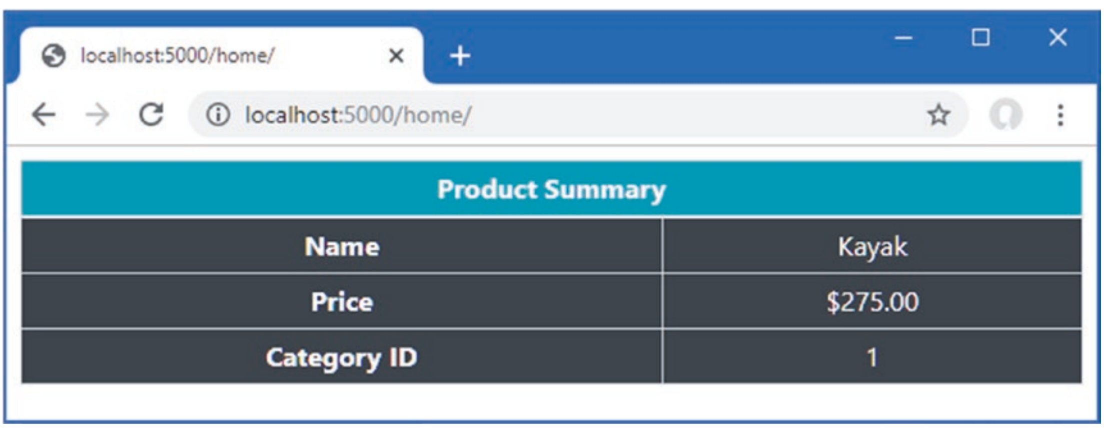

Figure 25-2. *Using a tag helper*

The tr element to which the attributes were applied in Listing 25-7 has been transformed, but that isn't the only change shown in the figure. By default, tag helpers apply to all elements of a specific type, which means that all the tr elements in the view have been transformed using the default values defined in the tag helper class, since no attributes were defined.

In fact, the problem is more serious because the @addTagHelper directives in the view import files mean that the example tag helper is applied to all tr elements used in any view rendered by controllers and Razor Pages. Use a browser to request http://localhost:5000/cities, for example, and you will see the tr elements in the response from Cities Razor Page have also been transformed, as shown in Figure 25-3.

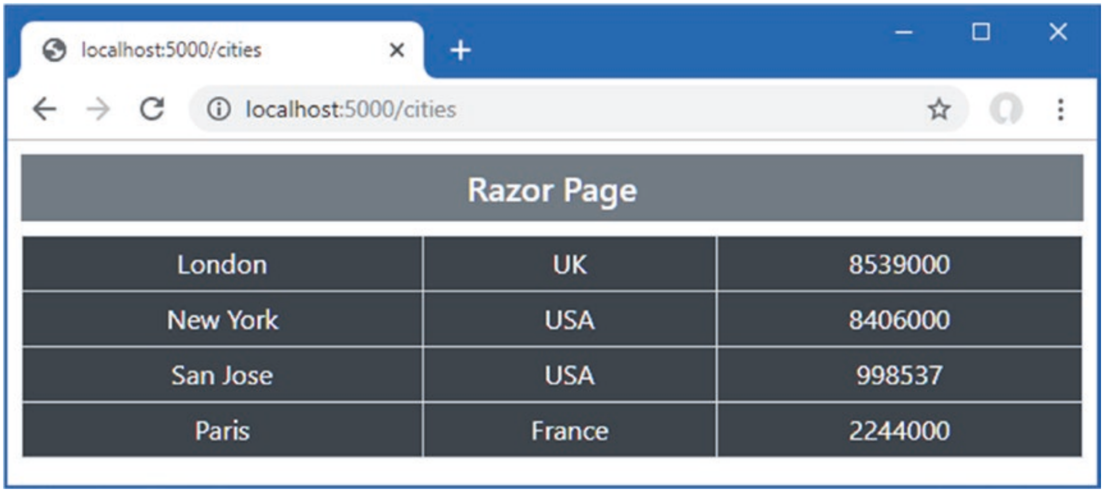

Figure 25-3. *Unexpectedly modifying elements with a tag helper*

Narrowing the Scope of a Tag Helper

The range of elements that are transformed by a tag helper can be controlled using the `HtmlTargetElement` element, as shown in Listing 25-8.

Listing 25-8. Narrowing Scope in the TrTagHelper.cs File in the TagHelpers Folder

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace WebApp.TagHelpers {

    [HtmlTargetElement("tr", Attributes = "bg-color,text-color", ParentTag ="thead")]
    public class TrTagHelper: TagHelper {

        public string BgColor { get; set; } = "dark";
        public string TextColor { get; set; } = "white";

        public override void Process(TagHelperContext context,
                TagHelperOutput output) {
            output.Attributes.SetAttribute("class",
                $"bg-{BgColor} text-center text-{TextColor}");
        }
    }
}
```

The `HtmlTargetElement` attribute describes the elements to which the tag helper applies. The first argument specifies the element type and supports the additional named properties described in Table 25-5.

Table 25-5. *The HtmlTargetElement Properties*

Name	Description
`Attributes`	This property is used to specify that a tag helper should be applied only to elements that have a given set of attributes, supplied as a comma-separated list. An attribute name that ends with an asterisk will be treated as a prefix so that `bg-*` will match `bg-color`, `bg-size`, and so on.
`ParentTag`	This property is used to specify that a tag helper should be applied only to elements that are contained within an element of a given type.
`TagStructure`	This property is used to specify that a tag helper should be applied only to elements whose tag structure corresponds to the given value from the `TagStructure` enumeration, which defines `Unspecified`, `NormalOrSelfClosing`, and `WithoutEndTag`.

The `Attributes` property supports CSS attribute selector syntax so that `[bg-color]` matches elements that have a `bg-color` attribute, `[bg-color=primary]` matches elements that have a `bg-color` attribute whose value is `primary`, and `[bg-color^=p]` matches elements with a `bg-color` attribute whose value begins with `p`. The attribute applied to the tag helper in Listing 25-8 matches `tr` elements with both `bg-color` and `text-color` attributes that are children of a `thead` element. Restart ASP.NET Core and use a browser to request `http://localhost:5000/home/index/1`, and you will see the scope of the tag helper has been narrowed, as shown in Figure 25-4.

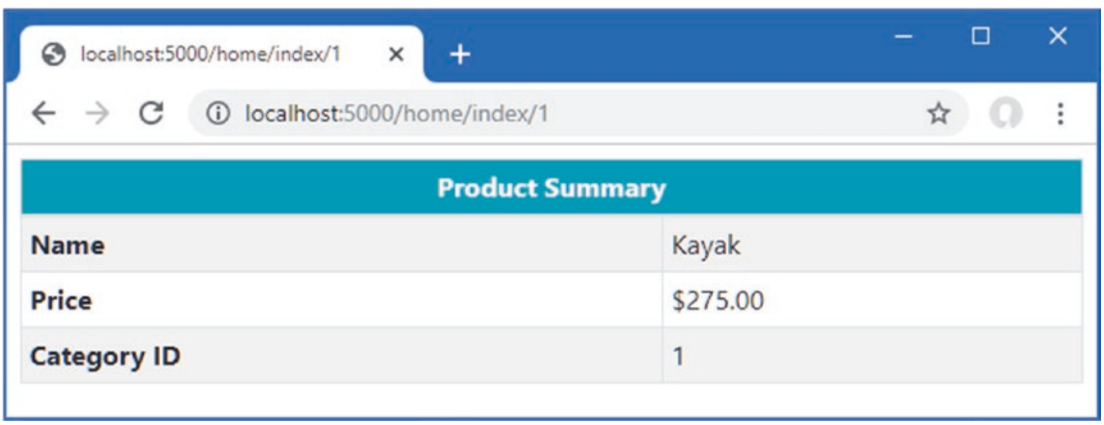

Figure 25-4. *Narrowing the scope of a tag helper*

Widening the Scope of a Tag Helper

The `HtmlTargetElement` attribute can also be used to widen the scope of a tag helper so that it matches a broader range of elements. This is done by setting the attribute's first argument to an asterisk (the * character), which matches any element. Listing 25-9 changes the attribute applied to the example tag helper so that it matches any element that has `bg-color` and `text-color` attributes.

Listing 25-9. Widening Scope in the TrTagHelper.cs File in the TagHelpers Folder

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace WebApp.TagHelpers {

    [HtmlTargetElement("*", Attributes = "bg-color,text-color")]
    public class TrTagHelper: TagHelper {

        public string BgColor { get; set; } = "dark";
        public string TextColor { get; set; } = "white";
```

```
        public override void Process(TagHelperContext context,
                TagHelperOutput output) {
            output.Attributes.SetAttribute("class",
                $"bg-{BgColor} text-center text-{TextColor}");
        }
    }
}
```

Care must be taken when using the asterisk because it is easy to match too widely and select elements that should not be transformed. A safer middle ground is to apply the `HtmlTargetElement` attribute for each type of element, as shown in Listing 25-10.

Listing 25-10. Balancing Scope in the TrTagHelper.cs File in the TagHelpers Folder

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace WebApp.TagHelpers {

    [HtmlTargetElement("tr", Attributes = "bg-color,text-color")]
    [HtmlTargetElement("td", Attributes = "bg-color")]
    public class TrTagHelper: TagHelper {

        public string BgColor { get; set; } = "dark";
        public string TextColor { get; set; } = "white";

        public override void Process(TagHelperContext context,
                TagHelperOutput output) {
            output.Attributes.SetAttribute("class",
                $"bg-{BgColor} text-center text-{TextColor}");
        }
    }
}
```

Each instance of the attribute can use different selection criteria. This tag helper matches `tr` elements with `bg-color` and `text-color` attributes and matches `td` elements with `bg-color` attributes. Listing 25-11 adds an element to be transformed to the `Index` view to demonstrate the revised scope.

Listing 25-11. Adding Attributes in the Index.cshtml File in the Views/Home Folder

```
@model Product
@{
    Layout = "_SimpleLayout";
}

<table class="table table-striped table-bordered table-sm">
    <thead>
        <tr bg-color="info" text-color="white">
            <th colspan="2">Product Summary</th>
        </tr>
    </thead>
    <tbody>
        <tr><th>Name</th><td>@Model.Name</td></tr>
        <tr>
            <th>Price</th>
            <td bg-color="dark">@Model.Price.ToString("c")</td>
        </tr>
        <tr><th>Category ID</th><td>@Model.CategoryId</td></tr>
    </tbody>
</table>
```

Restart ASP.NET Core and use a browser to request `http://localhost:5000/home/index/1`. The response will contain two transformed elements, as shown in Figure 25-5.

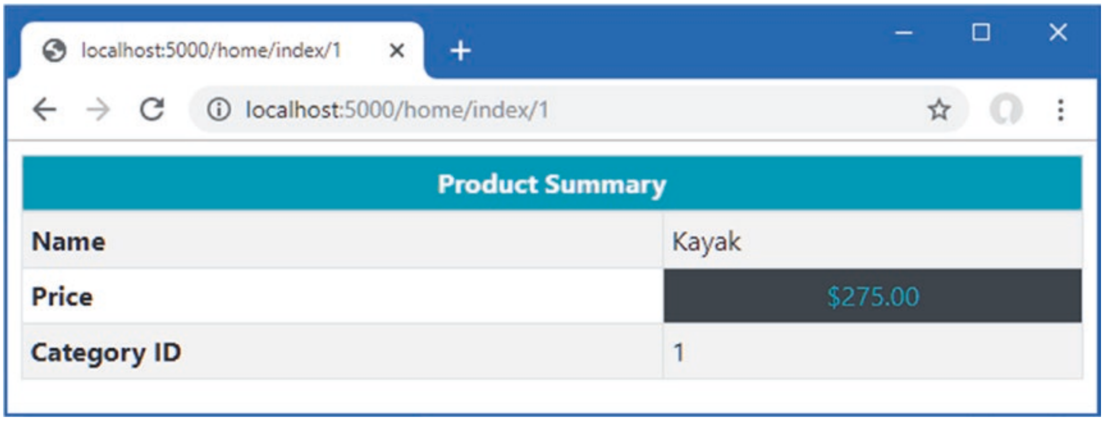

Figure 25-5. *Managing the scope of a tag helper*

> **ORDERING TAG HELPER EXECUTION**
>
> If you need to apply multiple tag helpers to an element, you can control the sequence in which they execute by setting the `Order` property, which is inherited from the `TagHelper` base class. Managing the sequence can help minimize the conflicts between tag helpers, although it is still easy to encounter problems.

Advanced Tag Helper Features

The previous section demonstrated how to create a basic tag helper, but that just scratches the surface of what's possible. In the sections that follow, I show more advanced uses for tag helpers and the features they provide.

Creating Shorthand Elements

Tag helpers are not restricted to transforming the standard HTML elements and can also be used to replace custom elements with commonly used content. This can be a useful feature for making views more concise and making their intent more obvious. To demonstrate, Listing 25-12 replaces the `thead` element in the `Index` view with a custom HTML element.

Listing 25-12. Adding a Custom HTML Element in the Index.cshtml File in the Views/Home Folder

```
@model Product
@{
    Layout = "_SimpleLayout";
}

<table class="table table-striped table-bordered table-sm">
    <tablehead bg-color="dark">Product Summary</tablehead>
    <tbody>
        <tr><th>Name</th><td>@Model.Name</td></tr>
        <tr>
            <th>Price</th>
            <td bg-color="dark">@Model.Price.ToString("c")</td>
        </tr>
```

```
        <tr><th>Category ID</th><td>@Model.CategoryId</td></tr>
    </tbody>
</table>
```

The tablehead element isn't part of the HTML specification and won't be understood by browsers. Instead, I am going to use this element as shorthand for generating the thead element and its content for the HTML table. Add a class named TableHeadTagHelper.cs to the TagHelpers folder and use it to define the class shown in Listing 25-13.

■ **Tip** When dealing with custom elements that are not part of the HTML specification, you must apply the HtmlTargetElement attribute and specify the element name, as shown in Listing 25-13. The convention of applying tag helpers to elements based on the class name works only for standard element names.

Listing 25-13. The Contents of TableHeadTagHelper.cs in the TagHelpers Folder

```
using Microsoft.AspNetCore.Razor.TagHelpers;
using System.Threading.Tasks;

namespace WebApp.TagHelpers {

    [HtmlTargetElement("tablehead")]
    public class TableHeadTagHelper: TagHelper {

        public string BgColor { get; set; } = "light";

        public override async Task ProcessAsync(TagHelperContext context,
                TagHelperOutput output) {

            output.TagName = "thead";
            output.TagMode = TagMode.StartTagAndEndTag;
            output.Attributes.SetAttribute("class",
                $"bg-{BgColor} text-white text-center");

            string content = (await output.GetChildContentAsync()).GetContent();
            output.Content
                .SetHtmlContent($"<tr><th colspan=\"2\">{content}</th></tr>");
        }
    }
}
```

This tag helper is asynchronous and overrides the ProcessAsync method so that it can access the existing content of the elements it transforms. The ProcessAsync method uses the properties of the TagHelperOuput object to generate a completely different element: the TagName property is used to specify a thead element, the TagMode property is used to specify that the element is written using start and end tags, the Attributes.SetAttribute method is used to define a class attribute, and the Content property is used to set the element content.

The existing content of the element is obtained through the asynchronous GetChildContentAsync method, which returns a TagHelperContent object. This is the same object that is returned by the TagHelperOutput.Content property and allows the content of the element to be inspected and changed using the same type, through the methods described in Table 25-6.

Table 25-6. Useful TagHelperContent Methods

Name	Description
`GetContent()`	This method returns the contents of the HTML element as a string.
`SetContent(text)`	This method sets the content of the output element. The `string` argument is encoded so that it is safe for inclusion in an HTML element.
`SetHtmlContent(html)`	This method sets the content of the output element. The `string` argument is assumed to be safely encoded. Use with caution.
`Append(text)`	This method safely encodes the specified `string` and adds it to the content of the output element.
`AppendHtml(html)`	This method adds the specified `string` to the content of the output element without performing any encoding. Use with caution.
`Clear()`	This method removes the content of the output element.

In Listing 25-13, the existing content of the element is read through the `GetContent` element and then set using the `SetHtmlContent` method. The effect is to wrap the existing content in the transformed element in `tr` and `th` elements.

Restart ASP.NET Core and navigate to `http://localhost:5000/home/index/1`, and you will see the effect of the tag helper, which is shown in Figure 25-6.

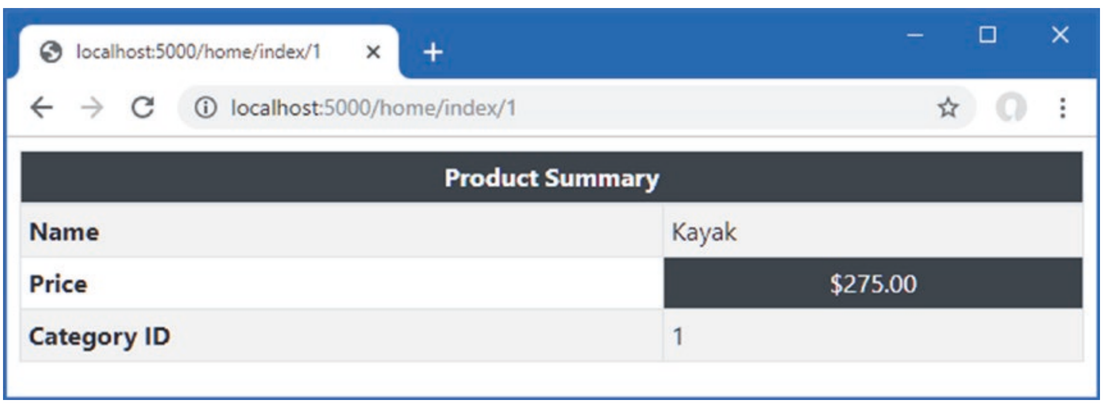

***Figure 25-6.** Using a shorthand element*

The tag helper transforms this shorthand element:

```
...
<tablehead bg-color="dark">Product Summary</tablehead>
...
```

into these elements:

```
...
<thead class="bg-dark text-white text-center">
    <tr>
        <th colspan="2">Product Summary</th>
    </tr>
</thead>
...
```

Notice that the transformed elements do not include the `bg-color` attribute. Attributes matched to properties defined by the tag helper are removed from the output element and must be explicitly redefined if they are required.

Creating Elements Programmatically

When generating new HTML elements, you can use standard C# string formatting to create the content you require, which is the approach I took in Listing 25-13. This works, but it can be awkward and requires close attention to avoid typos. A more robust approach is to use the `TagBuilder` class, which is defined in the `Microsoft.AspNetCore.Mvc.Rendering` namespace and allows elements to be created in a more structured manner. The `TagHelperContent` methods described in Table 25-6 accept `TagBuilder` objects, which makes it easy to create HTML content in tag helpers, as shown in Listing 25-14.

Listing 25-14. Creating HTML Elements in the TableHeadTagHelper.cs File in the TagHelpers Folder

```
using Microsoft.AspNetCore.Razor.TagHelpers;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Rendering;

namespace WebApp.TagHelpers {

    [HtmlTargetElement("tablehead")]
    public class TableHeadTagHelper: TagHelper {

        public string BgColor { get; set; } = "light";

        public override async Task ProcessAsync(TagHelperContext context,
                TagHelperOutput output) {

            output.TagName = "thead";
            output.TagMode = TagMode.StartTagAndEndTag;
            output.Attributes.SetAttribute("class",
                $"bg-{BgColor} text-white text-center");

            string content = (await output.GetChildContentAsync()).GetContent();

            TagBuilder header = new TagBuilder("th");
            header.Attributes["colspan"] = "2";
            header.InnerHtml.Append(content);

            TagBuilder row = new TagBuilder("tr");
            row.InnerHtml.AppendHtml(header);

            output.Content.SetHtmlContent(row);
        }
    }
}
```

This example creates each new element using a `TagBuilder` object and composes them to produce the same HTML structure as the string-based version in Listing 25-13.

Prepending and Appending Content and Elements

The `TagHelperOutput` class provides four properties that make it easy to inject new content into a view so that it surrounds an element or the element's content, as described in Table 25-7. In the sections that follow, I explain how you can insert content around and inside the target element.

Table 25-7. *The TagHelperOutput Properties for Appending Context and Elements*

Name	Description
PreElement	This property is used to insert elements into the view before the target element.
PostElement	This property is used to insert elements into the view after the target element.
PreContent	This property is used to insert content into the target element, before any existing content.
PostContent	This property is used to insert content into the target element, after any existing content.

Inserting Content Around the Output Element

The first TagHelperOuput properties are PreElement and PostElement, which are used to insert elements into the view before and after the output element. To demonstrate the use of these properties, add a class file named ContentWrapperTagHelper.cs to the WebApp/TagHelpers folder with the content shown in Listing 25-15.

Listing 25-15. The Contents of the WrapperTagHelper.cs File in the TagHelpers Folder

```
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace WebApp.TagHelpers {

    [HtmlTargetElement("*", Attributes = "[wrap=true]")]
    public class ContentWrapperTagHelper: TagHelper {

        public override void Process(TagHelperContext context,
                TagHelperOutput output) {
            TagBuilder elem = new TagBuilder("div");
            elem.Attributes["class"] = "bg-primary text-white p-2 m-2";
            elem.InnerHtml.AppendHtml("Wrapper");

            output.PreElement.AppendHtml(elem);
            output.PostElement.AppendHtml(elem);
        }
    }
}
```

This tag helper transforms elements that have a wrap attribute whose value is true, which it does using the PreElement and PostElement properties to add a div element before and after the output element. Listing 25-16 adds an element to the Index view that is transformed by the tag helper.

Listing 25-16. Adding an Element in the Index.cshtml File in the Views/Index Folder

```
@model Product
@{
    Layout = "_SimpleLayout";
}

<div class="m-2" wrap="true">Inner Content</div>

<table class="table table-striped table-bordered table-sm">
    <tablehead bg-color="dark">Product Summary</tablehead>
    <tbody>
        <tr><th>Name</th><td>@Model.Name</td></tr>
        <tr>
            <th>Price</th>
            <td bg-color="dark">@Model.Price.ToString("c")</td>
        </tr>
```

```
        <tr><th>Category ID</th><td>@Model.CategoryId</td></tr>
    </tbody>
</table>
```

Restart ASP.NET Core and use a browser to request `http://localhost:5000/home/index/1`. The response includes the transformed element, as shown in Figure 25-7.

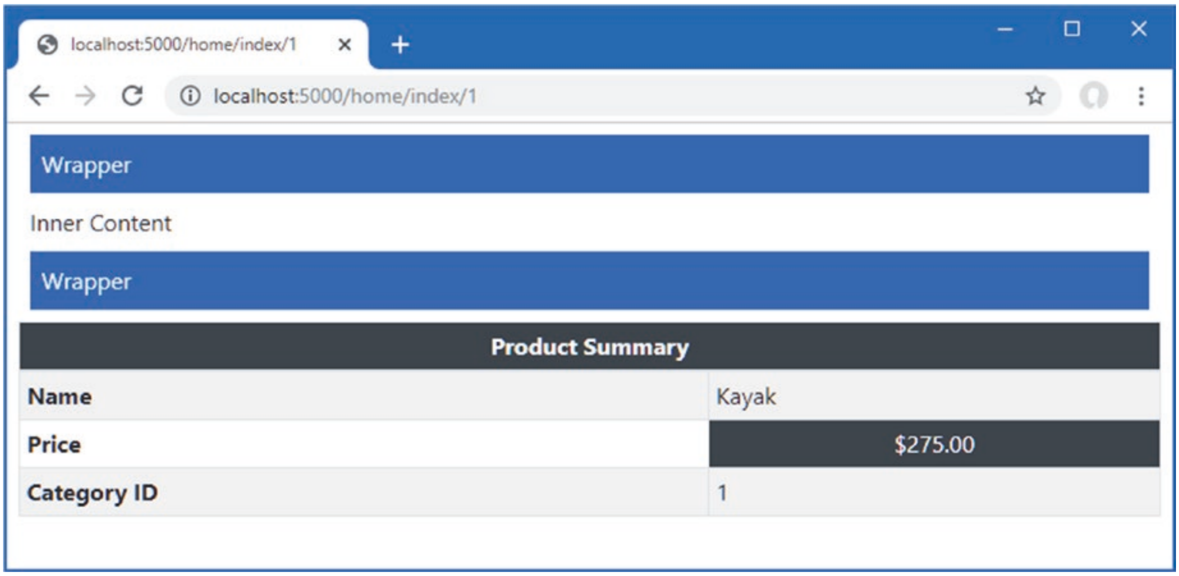

Figure 25-7. *Inserting content around the output element*

If you examine the HTML sent to the browser, you will see that this element:

```
...
<div class="m-2" wrap="true">Inner Content</div>
...
```

has been transformed into these elements:

```
...
<div class="bg-primary text-white p-2 m-2">Wrapper</div>
<div class="m-2" wrap="true">Inner Content</div>
<div class="bg-primary text-white p-2 m-2">Wrapper</div>
...
```

Notice that the `wrap` attribute has been left on the output element. This is because I didn't define a property in the tag helper class that corresponds to this attribute. If you want to prevent attributes from being included in the output, then define a property for them in the tag helper class, even if you don't use the attribute value.

Inserting Content Inside the Output Element

The `PreContent` and `PostContent` properties are used to insert content inside the output element, surrounding the original content. To demonstrate this feature, add a class file named `HighlightTagHelper.cs` to the `TagHelpers` folder and use it to define the tag helper shown in Listing 25-17.

Listing 25-17. The Contents of the HighlightTagHelper.cs File in the TagHelpers Folder

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace WebApp.TagHelpers {

    [HtmlTargetElement("*", Attributes = "[highlight=true]")]
    public class HighlightTagHelper: TagHelper {

        public override void Process(TagHelperContext context,
                TagHelperOutput output) {

            output.PreContent.SetHtmlContent("<b><i>");
            output.PostContent.SetHtmlContent("</i></b>");
        }
    }
}
```

This tag helper inserts `b` and `i` elements around the output element's content. Listing 25-18 adds the wrap attribute to one of the table cells in the `Index` view.

Listing 25-18. Adding an Attribute in the Index.cshtml File in the Views/Home Folder

```
@model Product
@{
    Layout = "_SimpleLayout";
}

<div class="m-2" wrap="true">Inner Content</div>

<table class="table table-striped table-bordered table-sm">
    <tablehead bg-color="dark">Product Summary</tablehead>
    <tbody>
        <tr><th>Name</th><td highlight="true">@Model.Name</td></tr>
        <tr>
            <th>Price</th>
            <td bg-color="dark">@Model.Price.ToString("c")</td>
        </tr>
        <tr><th>Category ID</th><td>@Model.CategoryId</td></tr>
    </tbody>
</table>
```

Restart ASP.NET Core and use a browser to request `http://localhost:5000/home/index/1`. The response includes the transformed element, as shown in Figure 25-8.

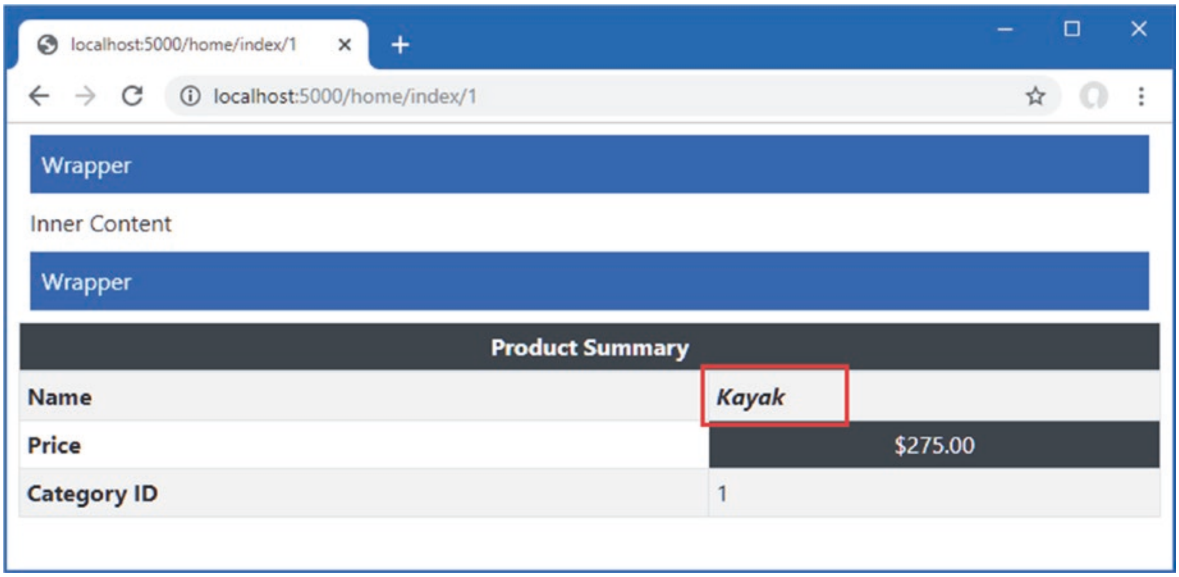

Figure 25-8. *Inserting content inside an element*

If you examine the HTML sent to the browser, you will see that this element:

```
...
<td highlight="true">@Model.Name</td>
...
```

has been transformed into these elements:

```
...
<td highlight="true"><b><i>Kayak</i></b></td>
...
```

Getting View Context Data

A common use for tag helpers is to transform elements so they contain details of the current request or the view model/ page model, which requires access to context data. To create this type of tag helper, add a file named `RouteDataTagHelper.cs` to the `TagHelpers` folder, with the content shown in Listing 25-19.

Listing 25-19. The Contents of the RouteDataTagHelper.cs File in the WebApps/TagHelpers Folder

```
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Microsoft.AspNetCore.Razor.TagHelpers;
using Microsoft.AspNetCore.Routing;

namespace WebApp.TagHelpers {

    [HtmlTargetElement("div", Attributes="[route-data=true]")]
    public class RouteDataTagHelper: TagHelper {

        [ViewContext]
        [HtmlAttributeNotBound]
        public ViewContext Context { get; set; }
```

```
        public override void Process(TagHelperContext context,
                TagHelperOutput output) {

            output.Attributes.SetAttribute("class", "bg-primary m-2 p-2");

            TagBuilder list = new TagBuilder("ul");
            list.Attributes["class"] = "list-group";
            RouteValueDictionary rd = Context.RouteData.Values;
            if (rd.Count > 0) {
                foreach (var kvp in rd) {
                    TagBuilder item = new TagBuilder("li");
                    item.Attributes["class"] = "list-group-item";
                    item.InnerHtml.Append($"{kvp.Key}: {kvp.Value}");
                    list.InnerHtml.AppendHtml(item);
                }
                output.Content.AppendHtml(list);
            } else {
                output.Content.Append("No route data");
            }
        }
    }
}
```

The tag helper transforms `div` elements that have a `route-data` attribute whose value is `true` and populates the output element with a list of the segment variables obtained by the routing system.

To get the route data, I added a property called `Context` and decorated it with two attributes, like this:

```
...
[ViewContext]
[HtmlAttributeNotBound]
public ViewContext Context { get; set; }
...
```

The `ViewContext` attribute denotes that the value of this property should be assigned a `ViewContext` object when a new instance of the tag helper class is created, which provides details of the view that is being rendered, including the routing data, as described in Chapter 13.

The `HtmlAttributeNotBound` attribute prevents a value from being assigned to this property if there is a matching attribute defined on the `div` element. This is good practice, especially if you are writing tag helpers for other developers to use.

■ **Tip** Tag helpers can declare dependencies on services in their constructors, which are resolved using the dependency injection feature described in Chapter 14.

Listing 25-20 adds an element to the `Home` controller's `Index` view that will be transformed by the new tag helper.

Listing 25-20. Adding an Element in the Index.cshtml File in the Views/Home Folder

```
@model Product
@{
    Layout = "_SimpleLayout";
}

<div route-data="true"></div>

<table class="table table-striped table-bordered table-sm">
    <tablehead bg-color="dark">Product Summary</tablehead>
```

```
    <tbody>
        <tr><th>Name</th><td highlight="true">@Model.Name</td></tr>
        <tr>
            <th>Price</th>
            <td bg-color="dark">@Model.Price.ToString("c")</td>
        </tr>
        <tr><th>Category ID</th><td>@Model.CategoryId</td></tr>
    </tbody>
</table>
```

Restart ASP.NET Core and use a browser to request `http://localhost:5000/home/index/1`. The response will include a list of the segment variables the routing system has matched, as shown in Figure 25-9.

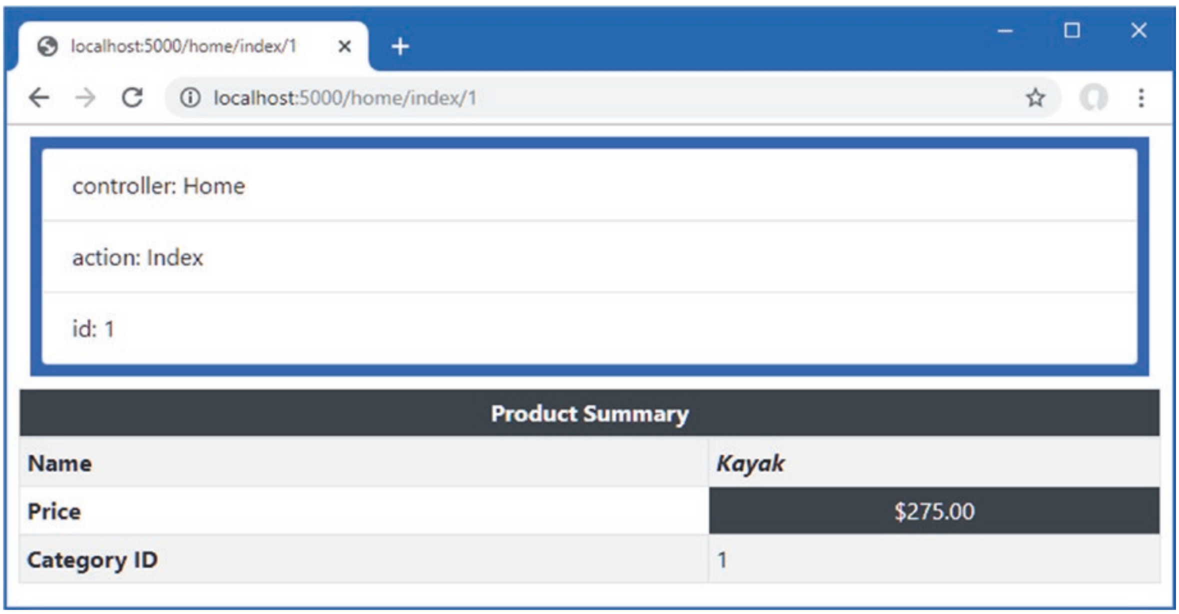

Figure 25-9. *Displaying context data with a tag helper*

Working with Model Expressions

Tag helpers can operate on the view model, tailoring the transformations they perform or the output they create. To see how this feature works, add a class file named `ModelRowTagHelper.cs` to the `TagHelpers` folder, with the code shown in Listing 25-21.

Listing 25-21. The Contents of the ModelRowTagHelper.cs File in the TagHelpers Folder

```
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace WebApp.TagHelpers {

    [HtmlTargetElement("tr", Attributes = "for")]
    public class ModelRowTagHelper : TagHelper {

        public string Format { get; set; }
        public ModelExpression For { get; set; }
```

```
        public override void Process(TagHelperContext context,
                TagHelperOutput output) {

            output.TagMode = TagMode.StartTagAndEndTag;

            TagBuilder th = new TagBuilder("th");
            th.InnerHtml.Append(For.Name);
            output.Content.AppendHtml(th);

            TagBuilder td = new TagBuilder("td");
            if (Format != null && For.Metadata.ModelType == typeof(decimal)) {
                td.InnerHtml.Append(((decimal)For.Model).ToString(Format));
            } else {
                td.InnerHtml.Append(For.Model.ToString());
            }
            output.Content.AppendHtml(td);
        }
    }
}
```

This tag helper transforms `tr` elements that have a `for` attribute. The important part of this tag helper is the type of the `For` property, which is used to receive the value of the `for` attribute.

```
...
public ModelExpression For { get; set; }
...
```

The `ModelExpression` class is used when you want to operate on part of the view model, which is most easily explained by jumping forward and showing how the tag helper is applied in the view, as shown in Listing 25-22.

■ **Note** The `ModelExpression` feature can be used only on view models or page models. It cannot be used on variables that are created within a view, such as with an `@foreach` expression.

Listing 25-22. Using the Tag Helper in the Index.cshtml File in the Views/Home Folder

```
@model Product
@{
    Layout = "_SimpleLayout";
}

<div route-data="true"></div>

<table class="table table-striped table-bordered table-sm">
    <tablehead bg-color="dark">Product Summary</tablehead>
    <tbody>
        <tr for="Name" />
        <tr for="Price" format="c" />
        <tr for="CategoryId" />
    </tbody>
</table>
```

The value of the `for` attribute is the name of a property defined by the view model class. When the tag helper is created, the type of the `For` property is detected and assigned a `ModelExpression` object that describes the selected property.

I am not going to describe the ModelExpression class in any detail because any introspection on types leads to endless lists of classes and properties. Further, ASP.NET Core provides a useful set of built-in tag helpers that use the view model to transform elements, as described in Chapter 26, which means you don't need to create your own.

For the example tag helper, I use three basic features that are worth describing. The first is to get the name of the model property so that I can include it in the output element, like this:

```
...
th.InnerHtml.Append(For.Name);
...
```

The Name property returns the name of the model property. The second feature is to get the type of the model property so that I can determine whether to format the value, like this:

```
...
if (Format != null && For.Metadata.ModelType == typeof(decimal)) {
...
```

The third feature is to get the value of the property so that it can be included in the response.

```
...
td.InnerHtml.Append(For.Model.ToString());
...
```

Restart ASP.NET Core and use a browser to request http://localhost:5000/home/index/2, and you will see the response shown in Figure 25-10.

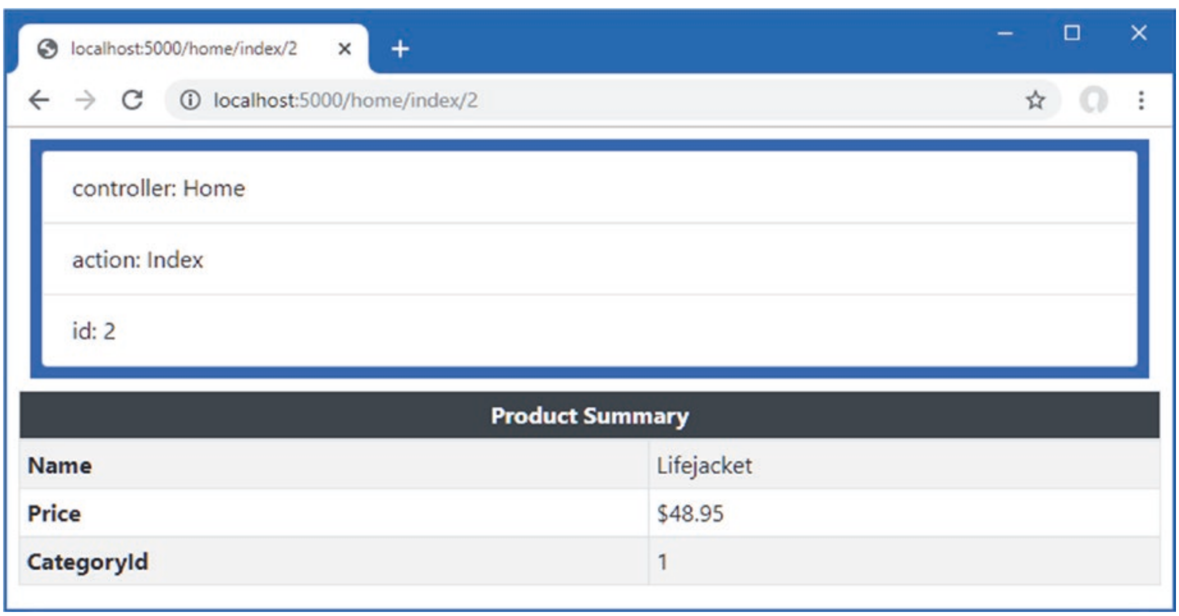

Figure 25-10. *Using the view model in a tag helper*

Working with the Page Model

Tag helpers with model expressions can be applied in Razor Pages, although the expression that selects the property must account for the way that the Model property returns the page model class. Listing 25-23 applies the tag helper to the Editor Razor Page, whose page model defines a Product property.

Listing 25-23. Applying a Tag Helper in the Editor.cshtml File in the Pages Folder

```
@page "{id:long}"
@model EditorModel
@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="bg-primary text-white text-center m-2 p-2">Editor</div>
    <div class="m-2">
        <table class="table table-sm table-striped table-bordered">
            <tbody>
                <tr for="Product.Name" />
                <tr for="Product.Price" format="c" />
            </tbody>
        </table>
        <form method="post">
            @Html.AntiForgeryToken()
            <div class="form-group">
                <label>Price</label>
                <input name="price" class="form-control"
                       value="@Model.Product.Price" />
            </div>
            <button class="btn btn-primary" type="submit">Submit</button>
        </form>
    </div>
</body>
</html>
```

The value for the `for` attribute selects the nested properties through the `Product` property, which provides the tag helper with the `ModelExpression` it requires. Use a browser to request `http://localhost:5000/editor/1` to see the response from the page, which is shown on the left of Figure 25-11.

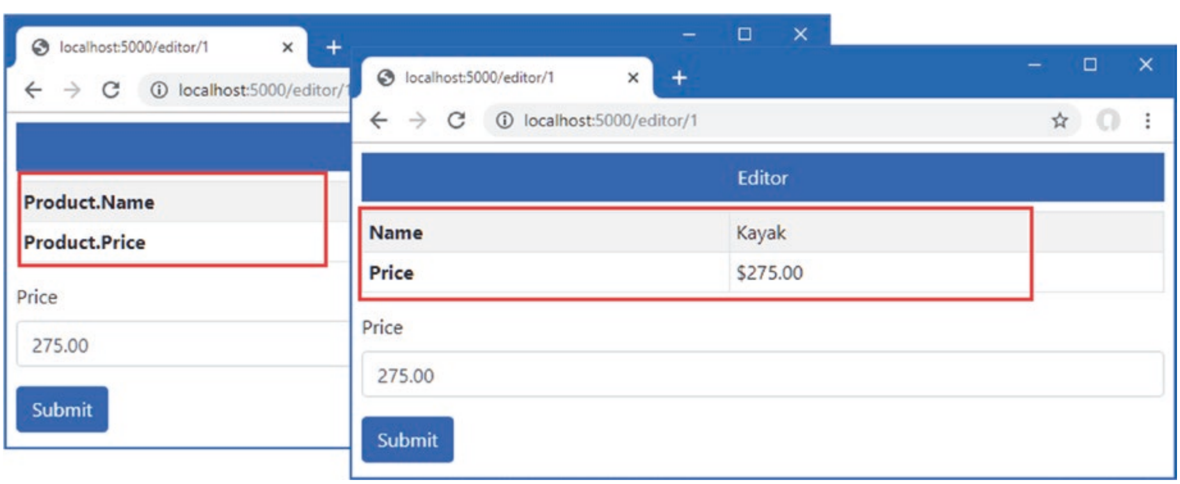

Figure 25-11. *Using a model expression tag helper with a Razor Page*

One consequence of the page model is that the `ModelExpression.Name` property will return `Product.Name`, for example, instead of just `Name`. Listing 25-24 updates the tag helper so that it will display just the last part of the model expression name.

■ **Note** This example is intended to highlight the effect of the page model on model expressions. Instead of displaying just the last part of the name, a more flexible approach is to add support for another attribute that allows the display value to be overridden as needed.

Listing 25-24. Processing Names in the ModelRowTagHelper.cs File in the TagHelpers Folder

```
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Microsoft.AspNetCore.Razor.TagHelpers;
using System.Linq;

namespace WebApp.TagHelpers {

    [HtmlTargetElement("tr", Attributes = "for")]
    public class ModelRowTagHelper : TagHelper {

        public string Format { get; set; }
        public ModelExpression For { get; set; }

        public override void Process(TagHelperContext context,
                TagHelperOutput output) {

            output.TagMode = TagMode.StartTagAndEndTag;

            TagBuilder th = new TagBuilder("th");
            th.InnerHtml.Append(For.Name.Split(".").Last());
            output.Content.AppendHtml(th);

            TagBuilder td = new TagBuilder("td");
            if (Format != null && For.Metadata.ModelType == typeof(decimal)) {
                td.InnerHtml.Append(((decimal)For.Model).ToString(Format));
            } else {
                td.InnerHtml.Append(For.Model.ToString());
            }
            output.Content.AppendHtml(td);
        }
    }
}
```

Restart ASP.NET Core and use a browser to request `http://localhost:5000/editor/1`; you will see the revised response, which is shown on the right of Figure 25-11.

Coordinating Between Tag Helpers

The `TagHelperContext.Items` property provides a dictionary used by tag helpers that operate on elements and those that operate on their descendants. To demonstrate the use of the `Items` collection, add a class file named `CoordinatingTagHelpers.cs` to the `WebApp/TagHelpers` folder and add the code shown in Listing 25-25.

Listing 25-25. The Contents of the CoordinatingTagHelpers.cs File in the TagHelpers Folder

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace WebApp.TagHelpers {

    [HtmlTargetElement("tr", Attributes = "theme")]
    public class RowTagHelper: TagHelper {

        public string Theme { get; set; }

        public override void Process(TagHelperContext context,
                TagHelperOutput output) {
            context.Items["theme"] = Theme;
        }
    }

    [HtmlTargetElement("th")]
    [HtmlTargetElement("td")]
    public class CellTagHelper : TagHelper {

        public override void Process(TagHelperContext context,
                TagHelperOutput output) {

            if (context.Items.ContainsKey("theme")) {
                output.Attributes.SetAttribute("class",
                    $"bg-{context.Items["theme"]} text-white");
            }
        }
    }
}
```

The first tag helper operates on `tr` elements that have a `theme` attribute. Coordinating tag helpers can transform their own elements, but this example simply adds the value of the `theme` attribute to the `Items` dictionary so that it is available to tag helpers that operate on elements contained within the `tr` element. The second tag helper operates on `th` and `td` elements and uses the `theme` value from the `Items` dictionary to set the Bootstrap style for its output elements.

Listing 25-26 adds elements to the Home controller's `Index` view that apply the coordinating tag helpers.

■ **Note** Notice that I have added the `th` and `td` elements that are transformed in Listing 25-26, instead of relying on a tag helper to generate them. Tag helpers are not applied to elements generated by other tag helpers and affect only the elements defined in the view.

Listing 25-26. Applying a Tag Helper in the Index.cshtml File in the Views/Home Folder

```
@model Product
@{
    Layout = "_SimpleLayout";
}

<table class="table table-striped table-bordered table-sm">
    <tablehead bg-color="dark">Product Summary</tablehead>
    <tbody>
        <tr theme="primary">
            <th>Name</th><td>@Model.Name</td>
        </tr>
```

```
        <tr theme="secondary">
            <th>Price</th><td>@Model.Price.ToString("c")</td>
        </tr>
        <tr theme="info">
            <th>Category</th><td>@Model.CategoryId</td>
        </tr>
    </tbody>
</table>
```

Restart ASP.NET Core and use a browser to request `http://localhost:5000/home`, which produces the response shown in Figure 25-12. The value of the `theme` element has been passed from one tag helper to another, and a color theme is applied without needing to define attributes on each of the elements that is transformed.

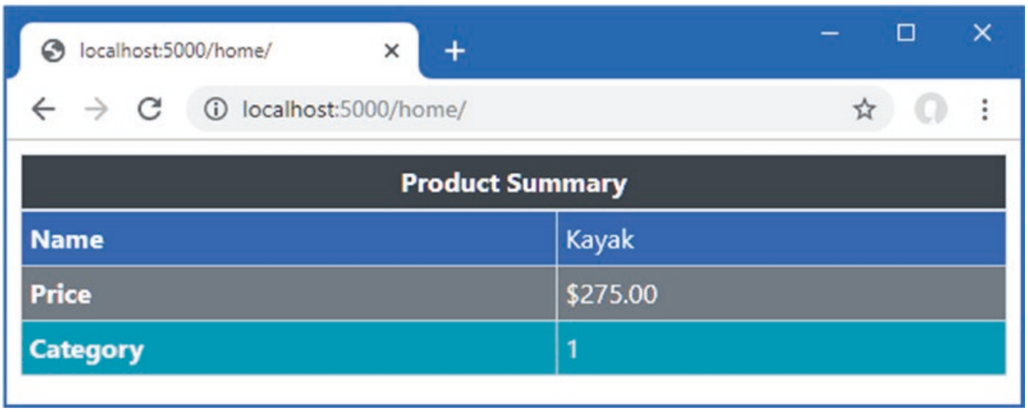

Figure 25-12. *Coordination between tag helpers*

Suppressing the Output Element

Tag helpers can be used to prevent an element from being included in the HTML response by calling the `SuppressOuput` method on the `TagHelperOutput` object that is received as an argument to the `Process` method. In Listing 25-27, I have added an element to the `Home` controller's `Index` view that should be displayed only if the `Price` property of the view model exceeds a specified value.

Listing 25-27. Adding an Element in the Index.cshtml File in the Views/Home Folder

```
@model Product
@{
    Layout = "_SimpleLayout";
}

<div show-when-gt="500" for="Price">
    <h5 class="bg-danger text-white text-center p-2">
        Warning: Expensive Item
    </h5>
</div>

<table class="table table-striped table-bordered table-sm">
    <tablehead bg-color="dark">Product Summary</tablehead>
    <tbody>
        <tr theme="primary">
            <th>Name</th><td>@Model.Name</td>
        </tr>
```

```
        <tr theme="secondary">
            <th>Price</th><td>@Model.Price.ToString("c")</td>
        </tr>
        <tr theme="info">
            <th>Category</th><td>@Model.CategoryId</td>
        </tr>
    </tbody>
</table>
```

The `show-when-gt` attribute specifies the value above which the `div` element should be displayed, and the `for` property selects the model property that will be inspected. To create the tag helper that will manage the elements, including the response, add a class file named `SelectiveTagHelper.cs` to the `WebApp/TagHelpers` folder with the code shown in Listing 25-28.

Listing 25-28. The Contents of the SelectiveTagHelper.cs File in the TagHelpers Folder

```
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace WebApp.TagHelpers {

    [HtmlTargetElement("div", Attributes = "show-when-gt, for")]
    public class SelectiveTagHelper: TagHelper {

        public decimal ShowWhenGt { get; set; }
        public ModelExpression For { get; set; }

        public override void Process(TagHelperContext context,
                TagHelperOutput output) {

            if (For.Model.GetType() == typeof(decimal)
                    && (decimal)For.Model <= ShowWhenGt) {
                output.SuppressOutput();
            }
        }
    }
}
```

The tag helper uses the model expression to access the property and calls the `SuppressOutput` method unless the threshold is exceeded. To see the effect, restart ASP.NET Core and use a browser to request `http://localhost:5000/home/index/1` and `http://localhost:5000/home/index/5`. The value for the `Price` property of the `Product` selected by the first URL is less than the threshold, so the element is suppressed. The value for the `Price` property of the `Product` selected by the second URL is more than the threshold, so the element is displayed. Figure 25-13 shows both responses.

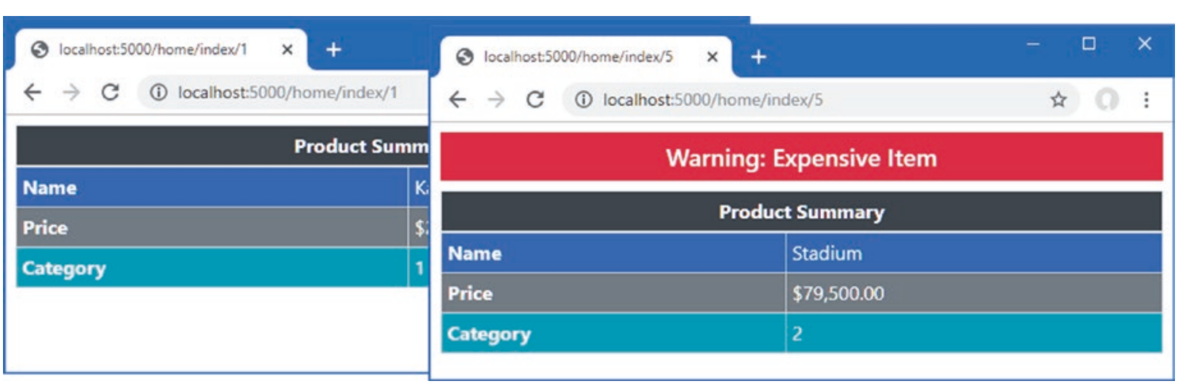

Figure 25-13. *Suppressing output elements*

Using Tag Helper Components

Tag helper components provide an alternative approach to applying tag helpers as services. This feature can be useful when you need to set up tag helpers to support another service or middleware component, which is typically the case for diagnostic tools or functionality that has both a client-side component and a server-side component, such as Blazor, which is described in Part 4. In the sections that follow, I show you how to create and apply tag helper components.

Creating a Tag Helper Component

Tag helper components are derived from the `TagHelperComponent` class, which provides a similar API to the `TagHelper` base class used in earlier examples. To create a tag helper component, add a class file called `TimeTagHelperComponent.cs` in the `TagHelpers` folder with the content shown in Listing 25-29.

Listing 25-29. The Contents of the TimeTagHelperComponent.cs File in the TagHelpers Folder

```
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Razor.TagHelpers;
using System;

namespace WebApp.TagHelpers {

    public class TimeTagHelperComponent: TagHelperComponent {

        public override void Process(TagHelperContext context,
                TagHelperOutput output) {

            string timestamp = DateTime.Now.ToLongTimeString();

            if (output.TagName == "body") {
                TagBuilder elem = new TagBuilder("div");
                elem.Attributes.Add("class", "bg-info text-white m-2 p-2");
                elem.InnerHtml.Append($"Time: {timestamp}");
                output.PreContent.AppendHtml(elem);
            }
        }
    }
}
```

Tag helper components do not specify the elements they transform, and the `Process` method is invoked for every element for which the tag helper component feature has been configured. By default, tag helper components are applied to transform head and body elements. This means that tag helper component classes must check the `TagName` property of the output element to ensure they perform only their intended transformations. The tag helper component in Listing 25-29 looks for body elements and uses the `PreContent` property to insert a `div` element containing a timestamp before the rest of the element's content.

■ **Tip** I show you how to increase the range of elements handled by tag helper components in the next section.

Tag helper components are registered as services that implement the `ITagHelperComponent` interface, as shown in Listing 25-30.

Listing 25-30. Registering a Tag Helper Component in the Startup.cs File in the WebApp Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using WebApp.Models;
using Microsoft.AspNetCore.Razor.TagHelpers;
using WebApp.TagHelpers;
```

```
namespace WebApp {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        public IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<DataContext>(opts => {
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:ProductConnection"]);
                opts.EnableSensitiveDataLogging(true);
            });
            services.AddControllersWithViews().AddRazorRuntimeCompilation();
            services.AddRazorPages().AddRazorRuntimeCompilation();
            services.AddSingleton<CitiesData>();
            services.AddTransient<ITagHelperComponent, TimeTagHelperComponent>();
        }

        public void Configure(IApplicationBuilder app, DataContext context) {
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseRouting();
            app.UseEndpoints(endpoints => {
                endpoints.MapControllers();
                endpoints.MapDefaultControllerRoute();
                endpoints.MapRazorPages();
            });
            SeedData.SeedDatabase(context);
        }
    }
}
```

The AddTransient method is used to ensure that each request is handled using its own instance of the tag helper component class. To see the effect of the tag helper component, restart ASP.NET Core and use a browser to request http://localhost:5000/home. This response—and all other HTML responses from the application—contain the content generated by the tag helper component, as shown in Figure 25-14.

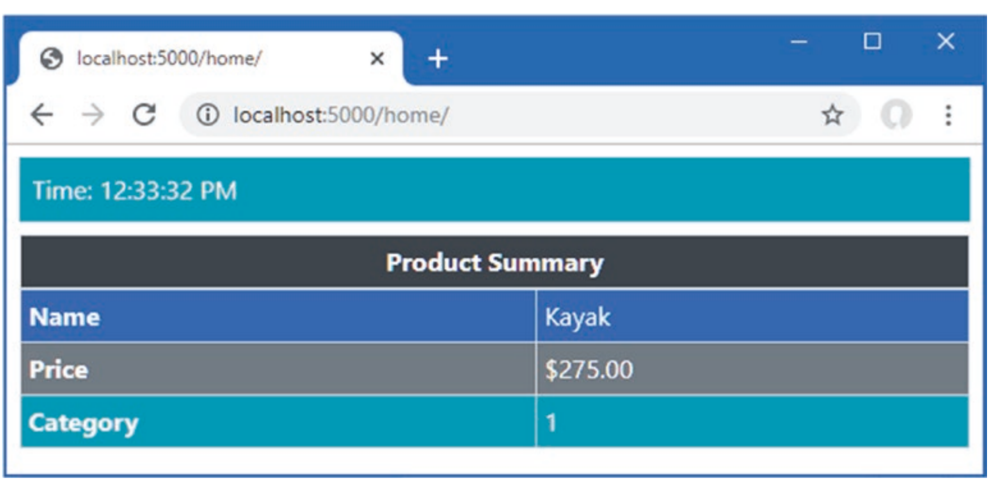

Figure 25-14. *Using a tag helper component*

Expanding Tag Helper Component Element Selection

By default, only the head and body elements are processed by the tag helper components, but additional elements can be selected by creating a class derived from the terribly named `TagHelperComponentTagHelper` class. Add a class file named `TableFooterTagHelperComponent.cs` to the `TagHelpers` folder and use it to define the classes shown in Listing 25-31.

Listing 25-31. The Contents of the TableFooterTagHelperComponent.cs File in the TagHelpers Folder

```
using Microsoft.AspNetCore.Mvc.Razor.TagHelpers;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Razor.TagHelpers;
using Microsoft.Extensions.Logging;

namespace WebApp.TagHelpers {

   [HtmlTargetElement("table")]
    public class TableFooterSelector: TagHelperComponentTagHelper {

        public TableFooterSelector(ITagHelperComponentManager mgr,
            ILoggerFactory log): base(mgr, log) { }
    }

    public class TableFooterTagHelperComponent: TagHelperComponent {

        public override void Process(TagHelperContext context,
                TagHelperOutput output) {

            if (output.TagName == "table") {
                TagBuilder cell = new TagBuilder("td");
                cell.Attributes.Add("colspan", "2");
                cell.Attributes.Add("class", "bg-dark text-white text-center");
                cell.InnerHtml.Append("Table Footer");
                TagBuilder row = new TagBuilder("tr");
                row.InnerHtml.AppendHtml(cell);
                TagBuilder footer = new TagBuilder("tfoot");
                footer.InnerHtml.AppendHtml(row);
                output.PostContent.AppendHtml(footer);
            }
        }
    }
}
```

The `TableFooterSelector` class is derived from `TagHelperComponentTagHelper`, and it is decorated with the `HtmlTargetElement` attribute that expands the range of elements processed by the application's tag helper components. In this case, the attribute selects `table` elements.

The `TableFooterTagHelperComponent` class, defined in the same file, is a tag helper component that transforms `table` elements by adding a `tfoot` element, which represents a table footer.

■ **Caution** Bear in mind that when you create a new `TagHelperComponentTagHelper`, all the tag helper components will receive the elements selected by the `HtmlTargetAttribute` element.

The tag helper component must be registered as a service to receive elements for transformation, but the tag helper component tag helper (which is one of the worst naming choices I have seen for some years) is discovered and applied automatically. Listing 25-32 adds the tag helper component service.

Listing 25-32. Registering a Tag Helper Component in the Startup.cs File in the WebApp Folder

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddDbContext<DataContext>(opts => {
        opts.UseSqlServer(Configuration[
            "ConnectionStrings:ProductConnection"]);
        opts.EnableSensitiveDataLogging(true);
    });
    services.AddControllersWithViews().AddRazorRuntimeCompilation();
    services.AddRazorPages().AddRazorRuntimeCompilation();
    services.AddSingleton<CitiesData>();
    services.AddTransient<ITagHelperComponent, TimeTagHelperComponent>();
    services.AddTransient<ITagHelperComponent, TableFooterTagHelperComponent>();
}
...
```

Restart ASP.NET Core and use a browser to request a URL that renders a table, such as `http://localhost:5000/home` or `http://localhost:5000/cities`. Each table will contain a table footer, as shown in Figure 25-15.

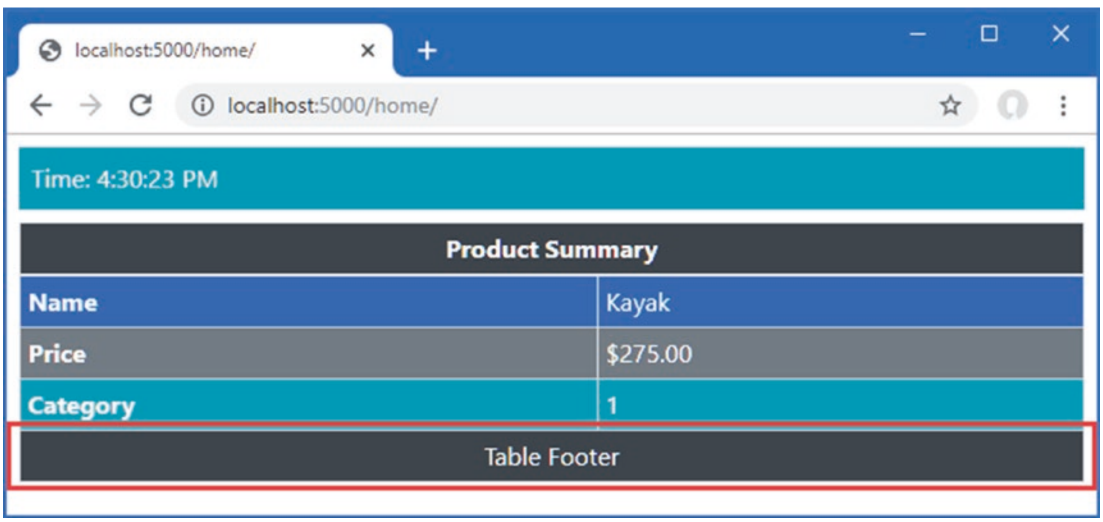

Figure 25-15. *Expanding tag helper component element selection*

Summary

In this chapter, I explained how tag helpers work and their role in transforming HTML elements in views and pages. I showed you how to create and apply tag helpers, how to control the elements that are selected for transformation, and how to use the advanced features to get specific results. I finished the chapter by explaining the tag helper component feature, which are defined as services. In the next chapter, I describe the built-in tag helpers that ASP.NET Core provides.

CHAPTER 26

Using the Built-in Tag Helpers

ASP.NET Core provides a set of built-in tag helpers that apply the most commonly required element transformations. In this chapter, I explain those tag helpers that deal with anchor, `script`, `link`, and image elements, as well as features for caching content and selecting content based on the environment. In Chapter 27, I describe the tag helpers that support HTML forms. Table 26-1 puts the built-in tag helpers in context.

Table 26-1. *Putting the Built-in Tag Helpers in Context*

Question	Answer
What are they?	The built-in tag helpers perform commonly required transformations on HTML elements.
Why are they useful?	Using the built-in tag helpers means you don't have to create custom helpers using the techniques in Chapter 25.
How are they used?	The tag helpers are applied using attributes on standard HTML elements or through custom HTML elements.
Are there any pitfalls or limitations?	No, these tag helpers are well-tested and easy to use. Unless you have unusual needs, using these tag helpers is preferable to custom implementation.
Are there any alternatives?	These tag helpers are optional, and their use is not required.

Table 26-2 summarizes the chapter.

Table 26-2. *Chapter Summary*

Problem	Solution	Listing
Creating elements that target endpoints	Use the anchor element tag helper attributes	7, 8
Including JavaScript files in a response	Use the JavaScript tag helper attributes	9–13
Including CSS files in a response	Use the CSS tag helper attributes	14, 15
Managing image caching	Use the image tag helper attributes	16
Caching sections of a view	Use the caching tag helper	17–21
Varying content based on the application environment	Use the environment tag helper	22

Preparing for This Chapter

This chapter uses the WebApp project from Chapter 25. To prepare for this chapter, comment out the statements that register the tag component helpers in the `Startup` class, as shown in Listing 26-1.

A. Freeman, *Pro ASP.NET Core 3*, https://doi.org/10.1007/978-1-4842-5440-0_26

■ **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-3. See Chapter 1 for how to get help if you have problems running the examples.

Listing 26-1. The Contents of the Startup.cs File in the WebApp Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using WebApp.Models;
//using Microsoft.AspNetCore.Razor.TagHelpers;
//using WebApp.TagHelpers;

namespace WebApp {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        public IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<DataContext>(opts => {
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:ProductConnection"]);
                opts.EnableSensitiveDataLogging(true);
            });
            services.AddControllersWithViews().AddRazorRuntimeCompilation();
            services.AddRazorPages().AddRazorRuntimeCompilation();
            services.AddSingleton<CitiesData>();
            //services.AddTransient<ITagHelperComponent, TimeTagHelperComponent>();
            //services.AddTransient<ITagHelperComponent,
            //        TableFooterTagHelperComponent>();
        }

        public void Configure(IApplicationBuilder app, DataContext context) {
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseRouting();
            app.UseEndpoints(endpoints => {
                endpoints.MapControllers();
                endpoints.MapDefaultControllerRoute();
                endpoints.MapRazorPages();
            });
            SeedData.SeedDatabase(context);
        }
    }
}
```

Next, update the `_RowPartial.cshtml` partial view in the `Views/Home` folder, making the changes shown in Listing 26-2.

Listing 26-2. Making Changes in the _RowPartial.cshtml File in the Views/Home Folder

```
@model Product

<tr>
    <td>@Model.Name</td>
    <td>@Model.Price.ToString("c")</td>
    <td>@Model.CategoryId</td>
    <td>@Model.SupplierId</td>
    <td></td>
</tr>
```

Add the elements shown in Listing 26-3 to define additional columns in the table rendered in the `Home` controller's `List` view.

Listing 26-3. Adding Elements in the List.cshtml File in the Views/Home Folder

```
@model IEnumerable<Product>
@{ Layout = "_SimpleLayout"; }

<h6 class="bg-secondary text-white text-center m-2 p-2">Products</h6>
<div class="m-2">
    <table class="table table-sm table-striped table-bordered">
        <thead>
            <tr>
                <th>Name</th><th>Price</th>
                <th>Category</th><th>Supplier</th><th></th>
            </tr>
        </thead>
        <tbody>
            @foreach (Product p in Model) {
                <partial name="_RowPartial" model="p" />
            }
        </tbody>
    </table>
</div>
```

Adding an Image File

One of the tag helpers described in this chapter provides services for images. I created the `wwwroot/images` folder and added an image file called `city.png`. This is a public domain panorama of the New York City skyline, as shown in Figure 26-1.

Figure 26-1. *Adding an image to the project*

This image file is included in the source code for this chapter, which is available in the GitHub repository for this book. You can substitute your own image if you don't want to download the example project.

Installing a Client-Side Package

Some of the examples in this chapter demonstrate the tag helper support for working with JavaScript files, for which I use the jQuery package. Use a PowerShell command prompt to run the command shown in Listing 26-4 in the project folder, which contains the `WebApp.csproj` file. If you are using Visual Studio, you can select Project ➤ Manage Client-Side Libraries to select the jQuery package.

Listing 26-4. Installing a Package

```
libman install jquery@3.4.1 -d wwwroot/lib/jquery
```

Dropping the Database

Open a new PowerShell command prompt, navigate to the folder that contains the `WebApp.csproj` file, and run the command shown in Listing 26-5 to drop the database.

Listing 26-5. Dropping the Database

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

Running the Example Application

Select Start Without Debugging or Run Without Debugging from the Debug menu or use the PowerShell command prompt to run the command shown in Listing 26-6.

Listing 26-6. Running the Example Application

```
dotnet run
```

Use a browser to request `http://localhost:5000/Home/list`, which will display a list of products, as shown in Figure 26-2.

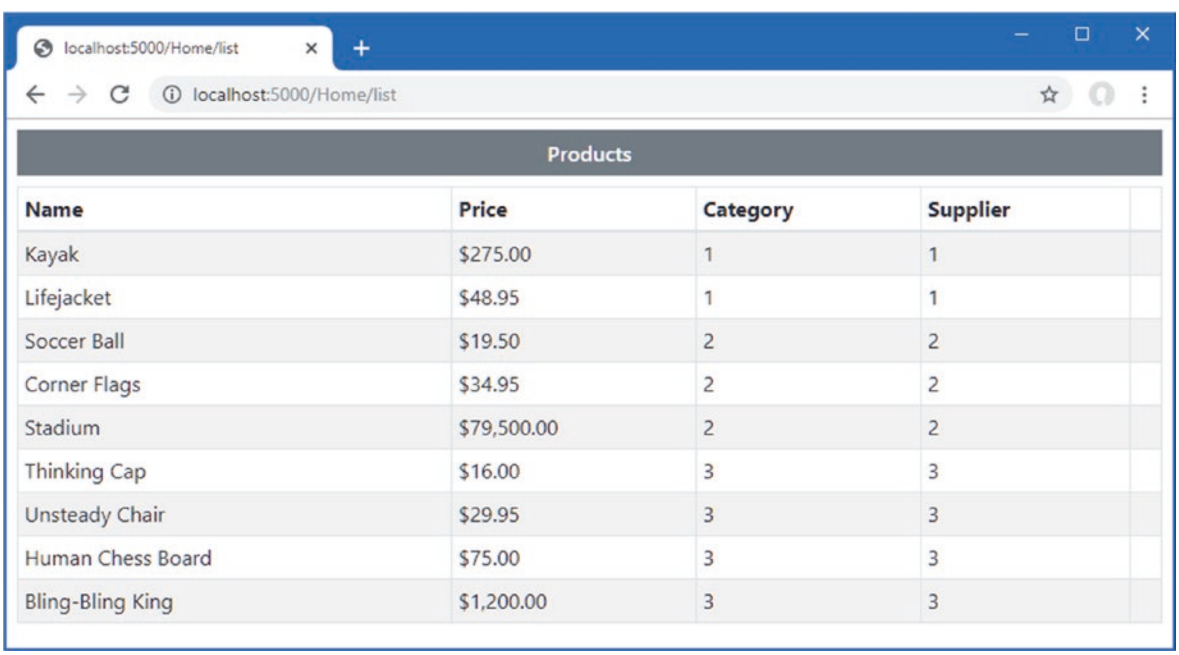

Figure 26-2. *Running the example application*

Enabling the Built-in Tag Helpers

The built-in tag helpers are all defined in the `Microsoft.AspNetCore.Mvc.TagHelpers` namespace and are enabled by adding an `@addTagHelpers` directive to individual views or pages or, as in the case of the example project, to the view imports file. Here is the required directive from the `_ViewImports.cshtml` file in the `Views` folder, which enables the built-in tag helpers for controller views:

```
@using WebApp.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@using WebApp.Components
@addTagHelper *, WebApp
```

Here is the corresponding directive in the `_ViewImports.cshtml` file in the `Pages` folder, which enables the built-in tag helpers for Razor Pages:

```
@namespace WebApp.Pages
@using WebApp.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, WebApp
```

These directives were added to the example project in Chapter 24 to enable the view components feature.

Transforming Anchor Elements

The a element is the basic tool for navigating around an application and sending GET requests to the application. The `AnchorTagHelper` class is used to transform the `href` attribute of a elements so they target URLs generated using the routing system, which means that hard-coded URLs are not required and a change in the routing configuration will be automatically reflected in the application's anchor elements. Table 26-3 describes the attributes the `AnchorTagHelper` class supports.

Table 26-3. *The Built-in Tag Helper Attributes for Anchor Elements*

Name	Description
`asp-action`	This attribute specifies the action method that the URL will target.
`asp-controller`	This attribute specifies the controller that the URL will target. If this attribute is omitted, then the URL will target the controller or page that rendered the current view.
`asp-page`	This attribute specifies the Razor Page that the URL will target.
`asp-page-handler`	This attribute specifies the Razor Page handler function that will process the request, as described in Chapter 23.
`asp-fragment`	This attribute is used to specify the URL fragment (which appears after the # character).
`asp-host`	This attribute specifies the name of the host that the URL will target.
`asp-protocol`	This attribute specifies the protocol that the URL will use.
`asp-route`	This attribute specifies the name of the route that will be used to generate the URL.
`asp-route-*`	Attributes whose name begins with `asp-route-` are used to specify additional values for the URL so that the `asp-route-id` attribute is used to provide a value for the `id` segment to the routing system.
`asp-all-route-data`	This attribute provides values used for routing as a single value, rather than using individual attributes.

The `AnchorTagHelper` is simple and predictable and makes it easy to generate URLs in a elements that use the application's routing configuration. Listing 26-7 adds an anchor element that uses attributes from the table to create a URL that targets another action defined by the `Home` controller.

Listing 26-7. Transforming an Element in the _RowPartial.cshtml File in the Views/Home Folder

```
@model Product

<tr>
    <td>@Model.Name</td>
    <td>@Model.Price.ToString("c")</td>
    <td>@Model.CategoryId</td>
    <td>@Model.SupplierId</td>
    <td>
        <a asp-action="index" asp-controller="home" asp-route-id="@Model.ProductId"
                class="btn btn-sm btn-info">
            Select
        </a>
    </td>
</tr>
```

The `asp-action` and `asp-controller` attributes specify the name of the action method and the controller that defines it. Values for segment variables are defined using `asp-route-[name]` attributes, such that the `asp-route-id` attribute provides a value for the `id` segment variable that is used to provide an argument for the action method selected by the `asp-action` attribute.

■ **Tip** The `class` attributes added to the anchor elements in Listing 26-7 apply Bootstrap CSS Framework styles that give the elements the appearance of buttons. This is not a requirement for using the tag helper.

To see the anchor element transformations, use a browser to request `http://localhost:5000/home/list`, which will produce the response shown in Figure 26-3.

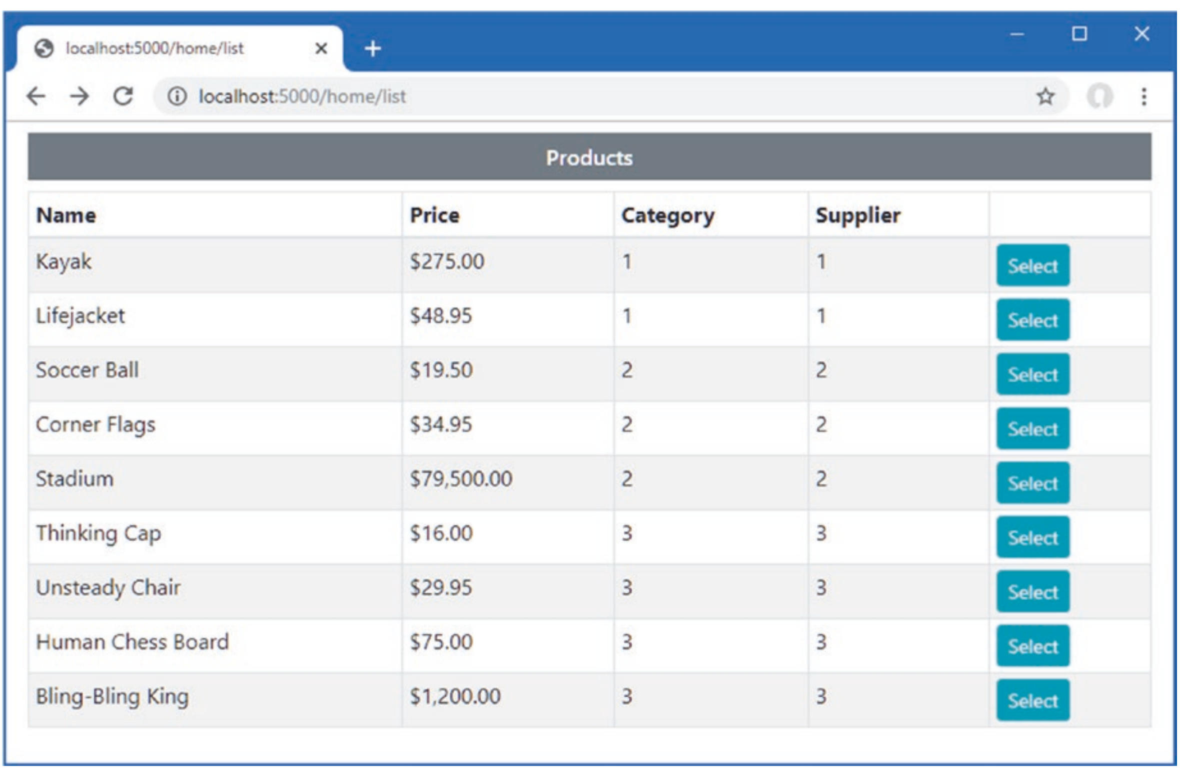

Figure 26-3. *Transforming anchor elements*

If you examine the Select anchor elements, you will see that each href attribute includes the ProductId value of the Product object it relates to, like this:

```
...
<a class="btn btn-sm btn-info" href="/Home/index/3">Select</a>
...
```

In this case, the value provided by the asp-route-id attribute means the default URL cannot be used, so the routing system has generated a URL that includes segments for the controller and action name, as well as a segment that will be used to provide a parameter to the action method. In both cases, since only an action method was specified, the URLs created by the tag helper target the controller that rendered the view. Clicking the anchor elements will send an HTTP GET request that targets the Home controller's Index method.

Using Anchor Elements for Razor Pages

The asp-page attribute is used to specify a Razor Page as the target for an anchor element's href attribute. The path to the page is prefixed with the / character, and values for route segments defined by the @page directive are defined using asp-route-[name] attributes. Listing 26-8 adds an anchor element that targets the List page defined in the Pages/Suppliers folder.

■ **Note** The asp-page-handler attribute can be used to specify the name of the page model handler method that will process the request.

Listing 26-8. Targeting a Razor Page in the List.cshtml File in the Views/Home Folder

```
@model IEnumerable<Product>
@{
    Layout = "_SimpleLayout";
}

<h6 class="bg-secondary text-white text-center m-2 p-2">Products</h6>
<div class="m-2">
    <table class="table table-sm table-striped table-bordered">
        <thead>
            <tr>
                <th>Name</th><th>Price</th>
                <th>Category</th><th>Supplier</th><th></th>
            </tr>
        </thead>
        <tbody>
            @foreach (Product p in Model) {
                <partial name="_RowPartial" model="p" />
            }
        </tbody>
    </table>
    <a asp-page="/suppliers/list" class="btn btn-secondary">Suppliers</a>
</div>
```

Use a browser to request http://localhost:5000/home/list, and you will see the anchor element, which is styled to appear as a button. If you examine the HTML sent to the client, you will see the anchor element has been transformed like this:

```
...
<a class="btn btn-secondary" href="/lists/suppliers">Suppliers</a>
...
```

This URL used in the href attribute reflects the @page directive, which has been used to override the default routing convention in this page. Click the element, and the browser will display the Razor Page, as shown in Figure 26-4.

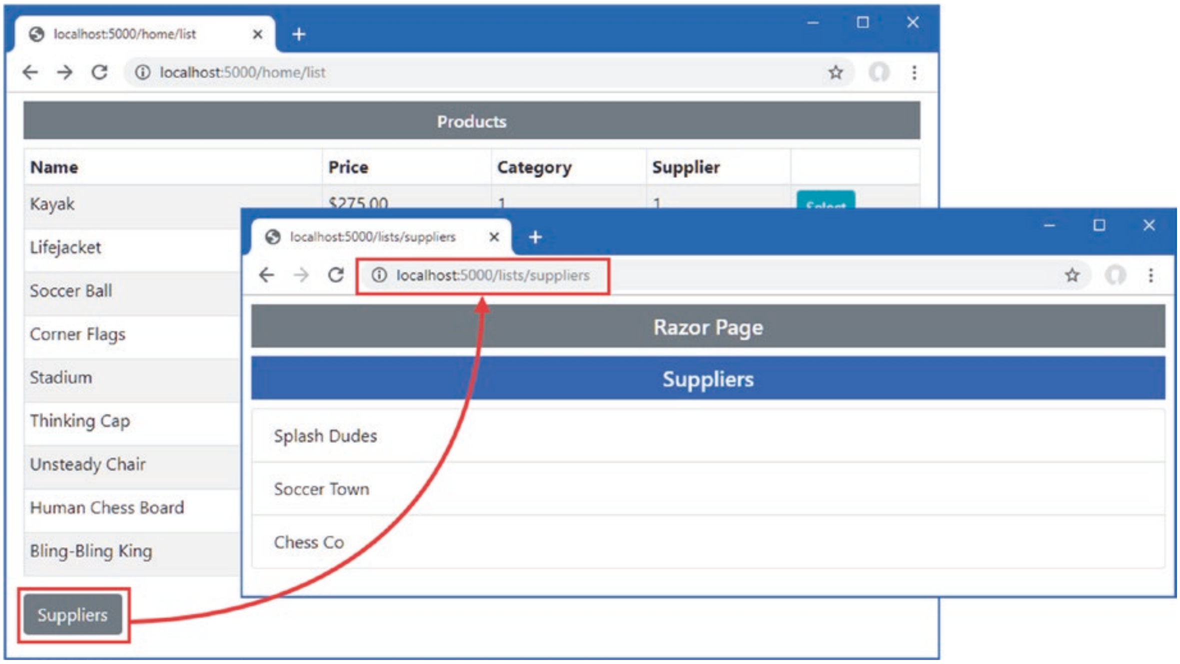

Figure 26-4. *Targeting a Razor Page with an anchor element*

GENERATING URLS (AND NOT LINKS)

The tag helper generates URLs only in anchor elements. If you need to generate a URL, rather than a link, then you can use the `Url` property, which is available in controllers, page models, and views. This property returns an object that implements the `IUrlHelper` interface, which provides a set of methods and extension methods that generate URLs. Here is a Razor fragment that generates a URL in a view:

```
...
<div>@Url.Page("/suppliers/list")</div>
...
```

This fragment produces a `div` element whose content is the URL that targets the `/Suppliers/List` Razor Page. The same interface is used in controllers or page model classes, such as with this statement:

```
...
string url = Url.Action("List", "Home");
...
```

The statement generates a URL that targets the `List` action on the `Home` controller and assigns it to the `string` variable named `url`.

Using the JavaScript and CSS Tag Helpers

ASP.NET Core provides tag helpers that are used to manage JavaScript files and CSS stylesheets through the `script` and `link` elements. As you will see in the sections that follow, these tag helpers are powerful and flexible but require close attention to avoid creating unexpected results.

Managing JavaScript Files

The `ScriptTagHelper` class is the built-in tag helper for `script` elements and is used to manage the inclusion of JavaScript files in views using the attributes described in Table 26-4, which I describe in the sections that follow.

Table 26-4. *The Built-in Tag Helper Attributes for script Elements*

Name	Description
`asp-src-include`	This attribute is used to specify JavaScript files that will be included in the view.
`asp-src-exclude`	This attribute is used to specify JavaScript files that will be excluded from the view.
`asp-append-version`	This attribute is used for cache busting, as described in the “Understanding Cache Busting” sidebar.
`asp-fallback-src`	This attribute is used to specify a fallback JavaScript file to use if there is a problem with a content delivery network.
`asp-fallback-src-include`	This attribute is used to select JavaScript files that will be used if there is a content delivery network problem.
`asp-fallback-src-exclude`	This attribute is used to exclude JavaScript files to present their use when there is a content delivery network problem.
`asp-fallback-test`	This attribute is used to specify a fragment of JavaScript that will be used to determine whether JavaScript code has been correctly loaded from a content delivery network.

Selecting JavaScript Files

The `asp-src-include` attribute is used to include JavaScript files in a view using globbing patterns. Globbing patterns support a set of wildcards that are used to match files, and Table 26-5 describes the most common globbing patterns.

Table 26-5. *Common Globbing Patterns*

Pattern	Example	Description
`?`	`js/src?.js`	This pattern matches any single character except `/`. The example matches any file contained in the `js` directory whose name is `src`, followed by any character, followed by `.js`, such as `js/src1.js` and `js/srcX.js` but not `js/src123.js` or `js/mydir/src1.js`.
`*`	`js/*.js`	This pattern matches any number of characters except `/`. The example matches any file contained in the `js` directory with the `.js` file extension, such as `js/src1.js` and `js/src123.js` but not `js/mydir/src1.js`.
`**`	`js/**/*.js`	This pattern matches any number of characters including `/`. The example matches any file with the `.js` extension that is contained within the `js` directory or any subdirectory, such as `/js/src1.js` and `/js/mydir/src1.js`.

Globbing is a useful way of ensuring that a view includes the JavaScript files that the application requires, even when the exact path to the file changes, which usually happens when the version number is included in the file name or when a package adds additional files.

Listing 26-9 uses the `asp-src-include` attribute to include all the JavaScript files in the `wwwroot/lib/jquery` folder, which is the location of the jQuery package installed with the command in Listing 26-4.

Listing 26-9. Selecting JS Files in the _SimpleLayout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
    <script asp-src-include="lib/jquery/**/*.js"></script>
</head>
<body>
    <div class="m-2">
        @RenderBody()
    </div>
</body>
</html>
```

Patterns are evaluated within the wwwroot folder, and the pattern I used locates any file with the js file extension, regardless of its location within the wwwroot folder; this means that any JavaScript package added to the project will be included in the HTML sent to the client.

Use a browser to request http://localhost:5000/home/list and examine the HTML sent to the browser. You will see the single script element in the layout has been transformed into a script element for each JavaScript file, like this:

```
...
<head>
  <title></title>
  <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet">
  <script src="/lib/jquery/core.js"></script>
  <script src="/lib/jquery/jquery.js"></script>
  <script src="/lib/jquery/jquery.min.js"></script>
  <script src="/lib/jquery/jquery.slim.js"></script>
  <script src="/lib/jquery/jquery.slim.min.js"></script>

</head>
...
```

If you are using Visual Studio, you may not have realized that the jQuery packages contain so many JavaScript files because Visual Studio hides them in the Solution Explorer. To reveal the full contents of the client-side package folders, you can either expand the individual nested entries in the Solution Explorer window or disable file nesting by clicking the button at the top of the Solution Explorer window, as shown in Figure 26-5. (Visual Studio Code does not nest files.)

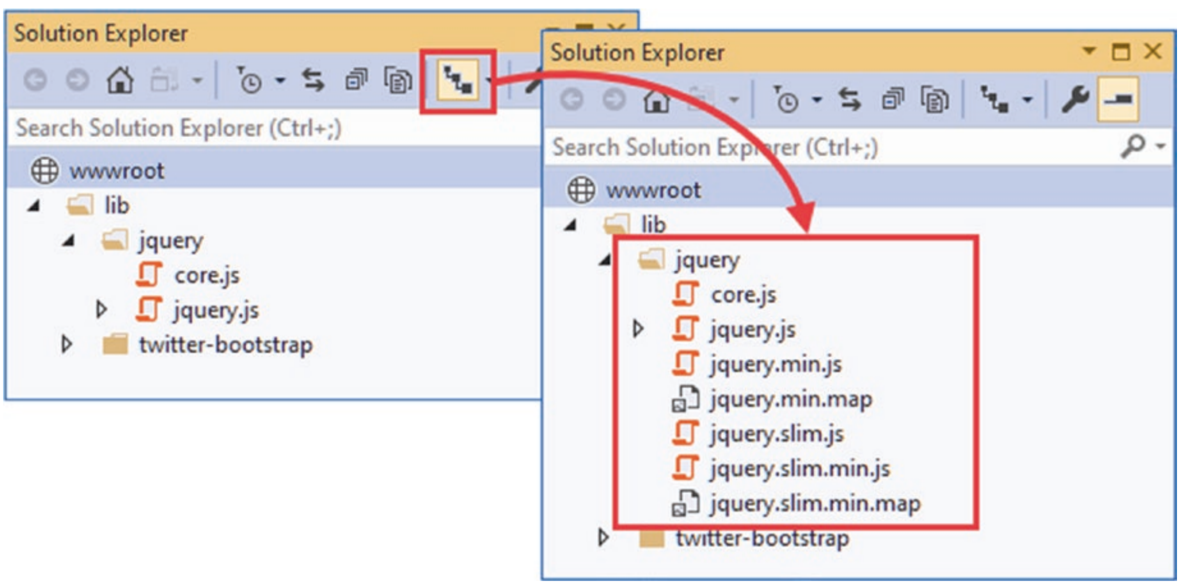

Figure 26-5. *Disabling file nesting in the Visual Studio Solution Explorer*

UNDERSTANDING SOURCE MAPS

JavaScript files are minified to make them smaller, which means they can be delivered to the client faster and using less bandwidth. The minification process removes all the whitespace from the file and renames functions and variables so that meaningful names such as myHelpfullyNamedFunction will be represented by a smaller number of characters, such as x1. When using the browser's JavaScript debugger to track down problems in your minified code, names like x1 make it almost impossible to follow progress through the code.

The files that have the map file extension are *source maps*, which browsers use to help debug minified code by providing a map between the minified code and the developer-readable, unminified source file. When you open the browser's F12 developer tools, the browser will automatically request source maps and use them to help debug the application's client-side code.

Narrowing the Globbing Pattern

No application would require all the files selected by the pattern in Listing 26-9. Many packages include multiple JavaScript files that contain similar content, often removing less popular features to save bandwidth. The jQuery package includes the `jquery.slim.js` file, which contains the same code as the `jquery.js` file but without the features that handle asynchronous HTTP requests and animation effects. (There is also a `core.js` file, but this is included in the package by error and should be ignored.)

Each of these files has a counterpart with the `min.js` file extension, which denotes a minified file. Minification reduces the size of a JavaScript file by removing all whitespace and renaming functions and variables to use shorter names.

Only one JavaScript file is required for each package and if you only require the minified versions, which will be the case in most projects, then you can restrict the set of files that the globbing pattern matches, as shown in Listing 26-10.

Listing 26-10. Selecting Minified Files in the _SimpleLayout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
    <script asp-src-include="lib/jquery**/*.min.js"></script>
</head>
<body>
    <div class="m-2">
        @RenderBody()
    </div>
</body>
</html>
```

Use a browser to request `http://localhost:5000/home/list` again and examine the HTML sent by the application. You will see that only the minified files have been selected.

```
...
<head>
  <title></title>
  <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet">
  <script src="/lib/jquery/jquery.min.js"></script>
  <script src="/lib/jquery/jquery.slim.min.js"></script>
</head>
...
```

Narrowing the pattern for the JavaScript files has helped, but the browser will still end up with the normal and slim versions of jQuery and the bundled and unbundled versions of the Bootstrap JavaScript files. To narrow the selection further, I can include `slim` in the pattern, as shown in Listing 26-11.

Listing 26-11. Narrowing the Focus in the _SimpleLayout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
    <script asp-src-include="lib/jquery**/*slim.min.js"></script>
</head>
<body>
    <div class="m-2">
        @RenderBody()
    </div>
</body>
</html>
```

Use the browser to request `http://localhost:5000/home/list` and examine the HTML the browser receives. The `script` element has been transformed like this:

```
...
<head>
  <title></title>
  <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet">
  <script src="/lib/jquery/jquery.slim.min.js"></script>
</head>
...
```

Only one version of the jQuery file will be sent to the browser while preserving the flexibility for the location of the file.

Excluding Files

Narrowing the pattern for the JavaScript files helps when you want to select a file whose name contains a specific term, such as `slim`. It isn't helpful when the file you want doesn't have that term, such as when you want the full version of the minified file. Fortunately, you can use the `asp-src-exclude` attribute to remove files from the list matched by the `asp-src-include` attribute, as shown in Listing 26-12.

Listing 26-12. Excluding Files in the _SimpleLayout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
    <script asp-src-include="/lib/jquery/**/*.min.js"
         asp-src-exclude="**.slim.**">
    </script>
</head>
<body>
    <div class="m-2">
        @RenderBody()
    </div>
</body>
</html>
```

If you use the browser to request `http://localhost:5000/home/list` and examine the HTML response, you will see that the `script` element links only to the full minified version of the jQuery library, like this:

```
...
<head>
    <title></title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet">
    <script src="/lib/jquery/jquery.min.js"></script>
</head>
...
```

UNDERSTANDING CACHE BUSTING

Static content, such as images, CSS stylesheets, and JavaScript files, is often cached to stop requests for content that rarely changes from reaching the application servers. Caching can be done in different ways: the browser can be told to cache content by the server, the application can use cache servers to supplement the application servers, or the content can be distributed using a content delivery network. Not all caching will be under your control. Large corporations, for example, often install caches to reduce their bandwidth demands since a substantial percentage of requests tend to go to the same sites or applications.

One problem with caching is that clients don't immediately receive new versions of static files when you deploy them because their requests are still being serviced by previously cached content. Eventually, the cached content will expire, and the new content will be used, but that leaves a period where the dynamic content generated by the application's controllers is out of step with the static content being delivered by the caches. This can lead to layout problems or unexpected application behavior, depending on the content that has been updated.

Addressing this problem is called *cache busting*. The idea is to allow caches to handle static content but immediately reflect any changes that are made at the server. The tag helper classes support cache busting by adding a query string to the URLs for static content that includes a checksum that acts as a version number. For JavaScript files, for example, the `ScriptTagHelper` class supports cache busting through the `asp-append-version` attribute, like this:

```
...
<script asp-src-include="/lib/jquery/**/*.min.js"
    asp-src-exclude="**.slim.**" asp-append-version="true">
</script>
...
```

Enabling the cache busting feature produces an element like this in the HTML sent to the browser:

```
...
<script src="/lib/jquery/dist/jquery.min.js?v=3zRSQ1HF-ocUiVcdv9yKTXqM"></script>
...
```

The same version number will be used by the tag helper until you change the contents of the file, such as by updating a JavaScript library, at which point a different checksum will be calculated. The addition of the version number means that each time you change the file, the client will request a different URL, which caches treat as a request for new content that cannot be satisfied with the previously cached content and pass on to the application server. The content is then cached as normal until the next update, which produces another URL with a different version.

Working with Content Delivery Networks

Content delivery networks (CDNs) are used to offload requests for application content to servers that are closer to the user. Rather than requesting a JavaScript file from your servers, the browser requests it from a hostname that resolves to a geographically local server, which reduces the amount of time required to load files and reduces the amount of bandwidth you have to provision for your application. If you have a large, geographically disbursed set of users, then it can make commercial sense to sign up to a CDN, but even the smallest and simplest application can benefit from using the free CDNs operated by major technology companies to deliver common JavaScript packages, such as jQuery.

For this chapter, I am going to use CDNJS, which is the same CDN used by the Library Manager tool to install client-side packages in the ASP.NET Core project. You can search for packages at `https://cdnjs.com`; for jQuery 3.4.1, which is the package and version installed in Listing 26-4, there are six CDNJS URLs.

- `https://cdnjs.cloudflare.com/ajax/libs/jquery/3.4.1/jquery.js`
- `https://cdnjs.cloudflare.com/ajax/libs/jquery/3.4.1/jquery.min.js`
- `https://cdnjs.cloudflare.com/ajax/libs/jquery/3.4.1/jquery.min.map`
- `https://cdnjs.cloudflare.com/ajax/libs/jquery/3.4.1/jquery.slim.js`
- `https://cdnjs.cloudflare.com/ajax/libs/jquery/3.4.1/jquery.slim.min.js`
- `https://cdnjs.cloudflare.com/ajax/libs/jquery/3.4.1/jquery.slim.min.map`

These URLs provide the regular JavaScript file, the minified JavaScript file, and the source map for the minified file for both the full and slim versions of jQuery. (There is also a URL for the `core.js` file, but, as noted earlier, this file is not used and will be removed from future jQuery releases.)

The problem with CDNs is that they are not under your organization's control, and that means they can fail, leaving your application running but unable to work as expected because the CDN content isn't available. The `ScriptTagHelper` class provides the ability to fall back to local files when the CDN content cannot be loaded by the client, as shown in Listing 26-13.

Listing 26-13. Using CDN Fallback in the _SimpleLayout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.4.1/jquery.min.js"
         asp-fallback-src="/lib/jquery/jquery.min.js"
         asp-fallback-test="window.jQuery">
    </script>
</head>
<body>
    <div class="m-2">
        @RenderBody()
    </div>
</body>
</html>
```

The `src` attribute is used to specify the CDN URL. The `asp-fallback-src` attribute is used to specify a local file that will be used if the CDN is unable to deliver the file specified by the regular `src` attribute. To figure out whether the CDN is working, the `asp-fallback-test` attribute is used to define a fragment of JavaScript that will be evaluated at the browser. If the fragment evaluates as `false`, then the fallback files will be requested.

■ **Tip** The `asp-fallback-src-include` and `asp-fallback-src-exclude` attributes can be used to select the local files with globbing patterns. However, given that CDN `script` elements select a single file, I recommend using the `asp-fallback-src attribute` to select the corresponding local file, as shown in the example.

Use a browser to request `http://localhost:5000/home/list`, and you will see that the HTML response contains two `script` elements, like this:

```
...
<head>
    <title></title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet">
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.4.1/jquery.min.js"></script>
    <script>
        (window.jQuery||document.write("\u003Cscript
           src=\u0022/lib/jquery/jquery.min.js\u0022\u003E\u003C/script\u003E"));
    </script>
</head>
...
```

The first `script` element requests the JavaScript file from the CDN. The second `script` element evaluates the JavaScript fragment specified by the `asp-fallback-test` attribute, which checks to see whether the first `script` element has worked. If the fragment evaluates to `true`, then no action is taken because the CDN worked. If the fragment evaluates to `false`, a new `script` element is added to the HTML document that instructs the browser to load the JavaScript file from the fallback URL.

It is important to test your fallback settings because you won't find out if they fail until the CDN has stopped working and your users cannot access your application. The simplest way to check the fallback is to change the name of the file specified by the `src` attribute to something that you know doesn't exist (I append the word `FAIL` to the file name) and then look at the network requests that the browser makes using the F12 developer tools. You should see an error for the CDN file followed by a request for the fallback file.

■ **Caution** The CDN fallback feature relies on browsers loading and executing the contents of `script` elements synchronously and in the order in which they are defined. There are a number of techniques in use to speed up JavaScript loading and execution by making the process asynchronous, but these can lead to the fallback test being performed before the browser has retrieved a file from the CDN and executed its contents, resulting in requests for the fallback files even when the CDN is working perfectly and defeating the use of a CDN in the first place. Do not mix asynchronous script loading with the CDN fallback feature.

Managing CSS Stylesheets

The `LinkTagHelper` class is the built-in tag helper for `link` elements and is used to manage the inclusion of CSS style sheets in a view. This tag helper supports the attributes described in Table 26-6, which I demonstrate in the following sections.

Table 26-6. *The Built-in Tag Helper Attributes for link Elements*

Name	Description
`asp-href-include`	This attribute is used to select files for the `href` attribute of the output element.
`asp-href-exclude`	This attribute is used to exclude files from the `href` attribute of the output element.
`asp-append-version`	This attribute is used to enable cache busting, as described in the "Understanding Cache Busting" sidebar.
`asp-fallback-href`	This attribute is used to specify a fallback file if there is a problem with a CDN.
`asp-fallback-href-include`	This attribute is used to select files that will be used if there is a CDN problem.
`asp-fallback-href-exclude`	This attribute is used to exclude files from the set that will be used when there is a CDN problem.
`asp-fallback-href-test-class`	This attribute is used to specify the CSS class that will be used to test the CDN.
`asp-fallback-href-test-property`	This attribute is used to specify the CSS property that will be used to test the CDN.
`asp-fallback-href-test-value`	This attribute is used to specify the CSS value that will be used to test the CDN.

Selecting Stylesheets

The `LinkTagHelper` shares many features with the `ScriptTagHelper`, including support for globbing patterns to select or exclude CSS files so they do not have to be specified individually. Being able to accurately select CSS files is as important as it is for JavaScript files because stylesheets can come in regular and minified versions and support source maps. The popular Bootstrap package, which I have been using to style HTML elements throughout this book, includes its CSS stylesheets in the `wwwroot/lib/twitter-bootstrap/css` folder. These will be visible in Visual Studio Code, but you will have to expand each item in the Solution Explorer or disable nesting to see them in the Visual Studio Solution Explorer, as shown in Figure 26-6.

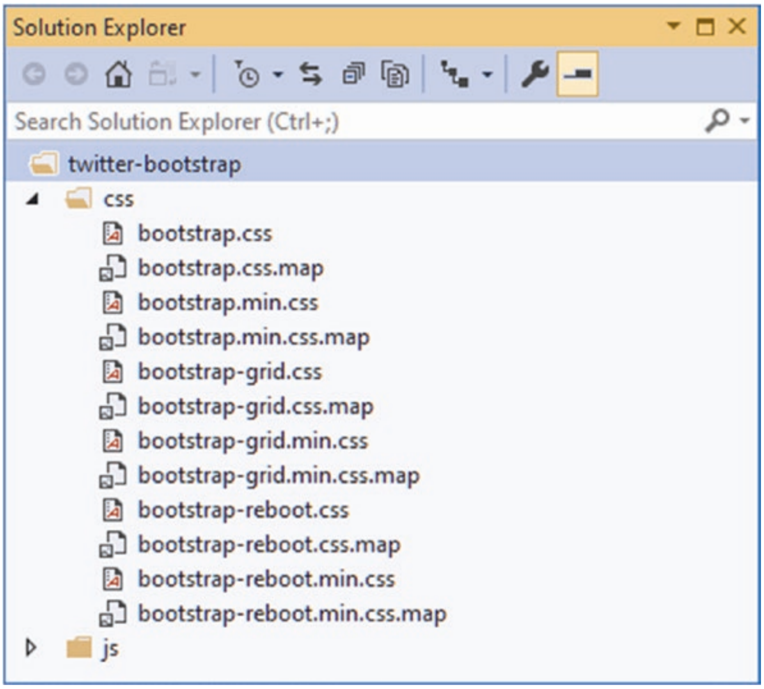

Figure 26-6. *The Bootstrap CSS files*

The `bootstrap.css` file is the regular stylesheet, the `bootstrap.min.css` file is the minified version, and the `bootstrap.css.map` file is a source map. The other files contain subsets of the CSS features to save bandwidth in applications that don't use them.

Listing 26-14 replaces the regular `link` element in the layout with one that uses the `asp-href-include` and `asp-href-exclude` attributes. (I removed the `script` element for jQuery, which is no longer required.)

Listing 26-14. Selecting a Stylesheet in the _SimpleLayout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link asp-href-include="/lib/twitter-bootstrap/css/*.min.css"
          asp-href-exclude="**/*-reboot*,**/*-grid*" rel="stylesheet" />
</head>
<body>
    <div class="m-2">
        @RenderBody()
    </div>
</body>
</html>
```

The same attention to detail is required as when selecting JavaScript files because it is easy to generate `link` elements for multiple versions of the same file or files that you don't want.

Working with Content Delivery Networks

The `LinkTag` helper class provides a set of attributes for falling back to local content when a CDN isn't available, although the process for testing to see whether a stylesheet has loaded is more complex than testing for a JavaScript file. Listing 26-15 uses the CDNJS URL for the Bootstrap CSS stylesheet.

Listing 26-15. Using a CDN for CSS in the _SimpleLayout.cshtml File in the Views/Home Folder

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/4.3.1/css/bootstrap.min.css"
        asp-fallback-href="/lib/twitter-bootstrap/css/bootstrap.min.css"
        asp-fallback-test-class="btn"
        asp-fallback-test-property="display"
        asp-fallback-test-value="inline-block"
        rel="stylesheet" />
</head>
<body>
    <div class="m-2">
        @RenderBody()
    </div>
</body>
</html>
```

The `href` attribute is used to specify the CDN URL, and I have used the `asp-fallback-href` attribute to select the file that will be used if the CDN is unavailable. Testing whether the CDN works, however, requires the use of three different attributes and an understanding of the CSS classes defined by the CSS stylesheet that is being used.

Use a browser to request `http://localhost:5000/home/list` and examine the HTML elements in the response. You will see that the `link` element from the layout has been transformed into three separate elements, like this:

```
...
<head>
    <title></title>
    <link href="https://cdnjs.cloudflare.com/.../bootstrap.min.css" rel="stylesheet">
    <meta name="x-stylesheet-fallback-test" content="" class="btn">
    <script>
      ! function(a, b, c, d) {
        var e, f = document,
          g = f.getElementsByTagName("SCRIPT"),
          h = g[g.length1].previousElementSibling,
          i = f.defaultView && f.defaultView.getComputedStyle ?
          f.defaultView.getComputedStyle(h) : h.currentStyle;
        if (i && i[a] !== b)
          for (e = 0; e < c.length; e++)
            f.write('<link href="' + c[e] + '" ' + d + "/>")
       }("display", "inline-block", ["/lib/twitter-bootstrap/css/bootstrap.min.css"],
          "rel=\u0022stylesheet\u0022 ");
    </script>
</head>
...
```

To make the transformation easier to understand, I have formatted the JavaScript code and shortened the URL.

The first element is a regular `link` whose `href` attribute specifies the CDN file. The second element is a `meta` element, which specifies the class from the `asp-fallback-test-class` attribute in the view. I specified the `btn` class in the listing, which means that an element like this is added to the HTML sent to the browser:

```
<meta name="x-stylesheet-fallback-test" content="" class="btn">
```

The CSS class that you specify must be defined in the stylesheet that will be loaded from the CDN. The `btn` class that I specified provides the basic formatting for Bootstrap button elements.

The `asp-fallback-test-property` attribute is used to specify a CSS property that is set when the CSS class is applied to an element, and the `asp-fallback-test-value` attribute is used to specify the value that it will be set to.

The `script` element created by the tag helper contains JavaScript code that adds an element to the specified class and then tests the value of the CSS property to determine whether the CDN stylesheet has been loaded. If not, a `link` element is created for the fallback file. The Bootstrap `btn` class sets the `display` property to `inline-block`, and this provides the test to see whether the browser has been able to load the Bootstrap stylesheet from the CDN.

■ **Tip** The easiest way to figure out how to test for third-party packages like Bootstrap is to use the browser's F12 developer tools. To determine the test in Listing 26-15, I assigned an element to the `btn` class and then inspected it in the browser, looking at the individual CSS properties that the class changes. I find this easier than trying to read through long and complex style sheets.

Working with Image Elements

The `ImageTagHelper` class is used to provide cache busting for images through the `src` attribute of `img` elements, allowing an application to take advantage of caching while ensuring that modifications to images are reflected immediately. The `ImageTagHelper` class operates in `img` elements that define the `asp-append-version` attribute, which is described in Table 26-7 for quick reference.

Table 26-7. *The Built-in Tag Helper Attribute for Image Elements*

Name	Description
`asp-append-version`	This attribute is used to enable cache busting, as described in the "Understanding Cache Busting" sidebar.

In Listing 26-16, I have added an `img` element to the shared layout for the city skyline image that I added to the project at the start of the chapter. I have also reset the `link` element to use a local file for brevity.

Listing 26-16. Adding an Image in the _SimpleLayout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="m-2">
        <img src="/images/city.png" asp-append-version="true" class="m-2" />
        @RenderBody()
    </div>
</body>
</html>
```

Use a browser to request `http://localhost:5000/home/list`, which will produce the response shown in Figure 26-7.

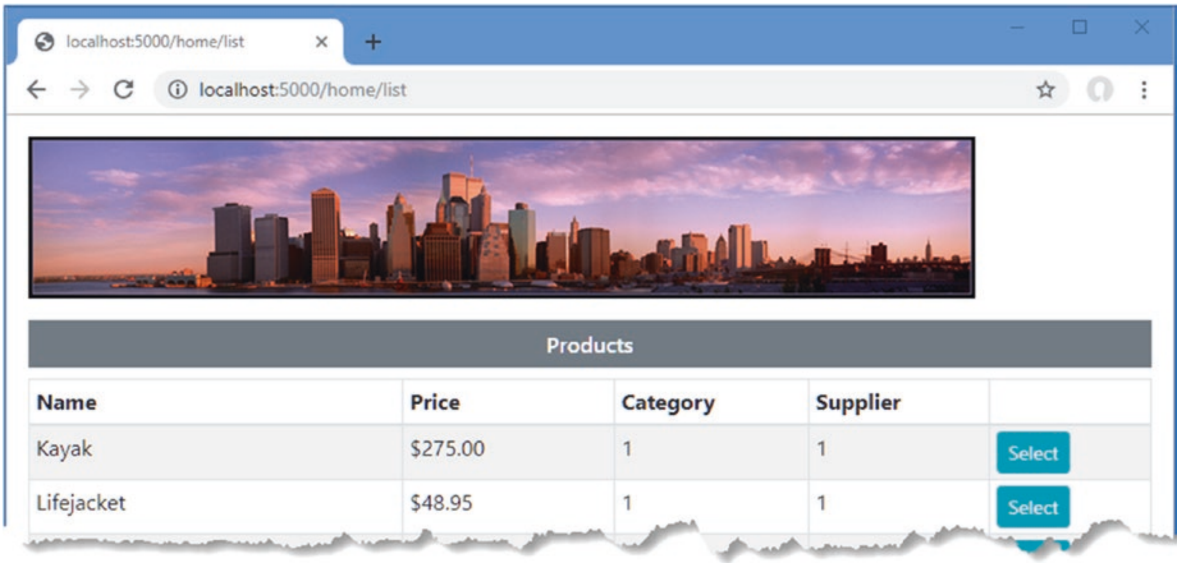

Figure 26-7. *Using an image*

Examine the HTML response, and you will see that the URL used to request the image file includes a version checksum, like this:

```
...
<img src="/images/city.png?v=KaMNDSZFAJufRcRDpKhOK_IIPNc7E" class="m-2">
...
```

The addition of the checksum ensures that any changes to the file will pass through any caches, avoiding stale content.

Using the Data Cache

The `CacheTagHelper` class allows fragments of content to be cached to speed up rendering of views or pages. The content to be cached is denoted using the `cache` element, which is configured using the attributes shown in Table 26-8.

■ **Note** Caching is a useful tool for reusing sections of content so they don't have to be generated for every request. But using caching effectively requires careful thought and planning. While caching can improve the performance of an application, it can also create odd effects, such as users receiving stale content, multiple caches containing different versions of content, and update deployments that are broken because content cached from the previous version of the application is mixed with content from the new version. Don't enable caching unless you have a clearly defined performance problem to resolve, and make sure you understand the impact that caching will have.

Table 26-8. *The Built-in Tag Helper Attributes for cache Elements*

Name	Description
enabled	This bool attribute is used to control whether the contents of the cache element are cached. Omitting this attribute enables caching.
expires-on	This attribute is used to specify an absolute time at which the cached content will expire, expressed as a DateTime value.
expires-after	This attribute is used to specify a relative time at which the cached content will expire, expressed as a TimeSpan value.
expires-sliding	This attribute is used to specify the period since it was last used when the cached content will expire, expressed as a TimeSpan value.
vary-by-header	This attribute is used to specify the name of a request header that will be used to manage different versions of the cached content.
vary-by-query	This attribute is used to specify the name of a query string key that will be used to manage different versions of the cached content.
vary-by-route	This attribute is used to specify the name of a routing variable that will be used to manage different versions of the cached content.
vary-by-cookie	This attribute is used to specify the name of a cookie that will be used to manage different versions of the cached content.
vary-by-user	This bool attribute is used to specify whether the name of the authenticated user will be used to manage different versions of the cached content.
vary-by	This attribute is evaluated to provide a key used to manage different versions of the content.
priority	This attribute is used to specify a relative priority that will be taken into account when the memory cache runs out of space and purges unexpired cached content.

Listing 26-17 replaces the img element from the previous section with content that contains timestamps.

Listing 26-17. Caching Content in the _SimpleLayout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="m-2">
        <h6 class="bg-primary text-white m-2 p-2">
            Uncached timestamp: @DateTime.Now.ToLongTimeString()
        </h6>
        <cache>
            <h6 class="bg-primary text-white m-2 p-2">
                Cached timestamp: @DateTime.Now.ToLongTimeString()
            </h6>
        </cache>
        @RenderBody()
    </div>
</body>
</html>
```

The cache element is used to denote a region of content that should be cached and has been applied to one of the h6 elements that contains a timestamp. Use a browser to request http://localhost:5000/home/list, and both timestamps will be the same. Reload the browser, and you will see that the cached content is used for one of the h6 elements and the timestamp doesn't change, as shown in Figure 26-8.

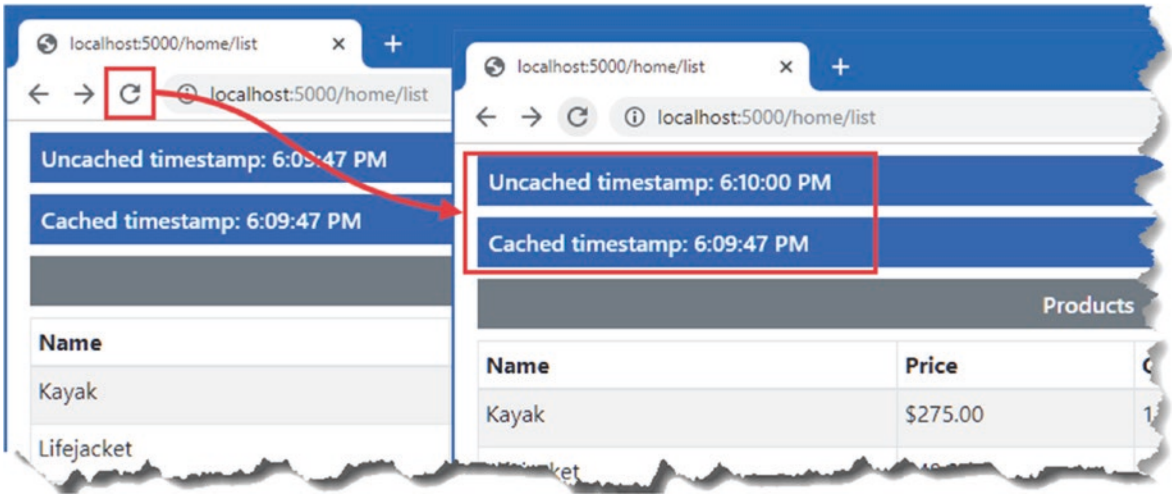

Figure 26-8. *Using the caching tag helper*

USING DISTRIBUTED CACHING FOR CONTENT

The cache used by the `CacheTagHelper` class is memory-based, which means that its capacity is limited by the available RAM and that each application server maintains a separate cache. Content will be ejected from the cache when there is a shortage of capacity available, and the entire contents are lost when the application is stopped or restarted.

The `distributed-cache` element can be used to store content in a shared cache, which ensures that all application servers use the same data and that the cache survives restarts. The `distributed-cache` element is configured with the same attributes as the cache element, as described in Table 26-8. See Chapter 17 for details of setting up a distributed cache.

Setting Cache Expiry

The `expires-*` attributes allow you to specify when cached content will expire, expressed either as an absolute time or a time relative to the current time, or to specify a duration during which the cached content isn't requested. In Listing 26-18, I have used the `expires-after` attribute to specify that the content should be cached for 15 seconds.

Listing 26-18. Setting Cache Expiry in the _SimpleLayout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="m-2">
        <h6 class="bg-primary text-white m-2 p-2">
            Uncached timestamp: @DateTime.Now.ToLongTimeString()
        </h6>
        <cache expires-after="@TimeSpan.FromSeconds(15)">
            <h6 class="bg-primary text-white m-2 p-2">
                Cached timestamp: @DateTime.Now.ToLongTimeString()
            </h6>
        </cache>
        @RenderBody()
```

```
    </div>
</body>
</html>
```

Use a browser to request `http://localhost:5000/home/list` and then reload the page. After 15 seconds the cached content will expire, and a new section of content will be created.

Setting a Fixed Expiry Point

You can specify a fixed time at which cached content will expire using the `expires-on` attribute, which accepts a `DateTime` value, as shown in Listing 26-19.

Listing 26-19. Setting Cache Expiry in the _SimpleLayout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="m-2">
        <h6 class="bg-primary text-white m-2 p-2">
            Uncached timestamp: @DateTime.Now.ToLongTimeString()
        </h6>
        <cache expires-on="@DateTime.Parse("2100-01-01")">
            <h6 class="bg-primary text-white m-2 p-2">
                Cached timestamp: @DateTime.Now.ToLongTimeString()
            </h6>
        </cache>
        @RenderBody()
    </div>
</body>
</html>
```

I have specified that that data should be cached until the year 2100. This isn't a useful caching strategy since the application is likely to be restarted before the next century starts, but it does illustrate how you can specify a fixed point in the future rather than expressing the expiry point relative to the moment when the content is cached.

Setting a Last-Used Expiry Period

The `expires-sliding` attribute is used to specify a period after which content is expired if it hasn't been retrieved from the cache. In Listing 26-20, I have specified a sliding expiry of 10 seconds.

Listing 26-20. Using a Sliding Expiry in the _SimpleLayout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
```

```
<body>
    <div class="m-2">
        <h6 class="bg-primary text-white m-2 p-2">
            Uncached timestamp: @DateTime.Now.ToLongTimeString()
        </h6>
        <cache expires-sliding="@TimeSpan.FromSeconds(10)">
            <h6 class="bg-primary text-white m-2 p-2">
                Cached timestamp: @DateTime.Now.ToLongTimeString()
            </h6>
        </cache>
        @RenderBody()
    </div>
</body>
</html>
```

You can see the effect of the express-sliding attribute by requesting http://localhost:5000/home/list and periodically reloading the page. If you reload the page within 10 seconds, the cached content will be used. If you wait longer than 10 seconds to reload the page, then the cached content will be discarded, the view component will be used to generate new content, and the process will begin anew.

Using Cache Variations

By default, all requests receive the same cached content. The CacheTagHelper class can maintain different versions of cached content and use them to satisfy different types of HTTP requests, specified using one of the attributes whose name begins with vary-by. Listing 26-21 shows the use of the vary-by-route attribute to create cache variations based on the action value matched by the routing system.

Listing 26-21. Creating a Variation in the _SimpleLayout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="m-2">
        <h6 class="bg-primary text-white m-2 p-2">
            Uncached timestamp: @DateTime.Now.ToLongTimeString()
        </h6>
        <cache expires-sliding="@TimeSpan.FromSeconds(10)" vary-by-route="action">
            <h6 class="bg-primary text-white m-2 p-2">
                Cached timestamp: @DateTime.Now.ToLongTimeString()
            </h6>
        </cache>
        @RenderBody()
    </div>
</body>
</html>
```

If you use two browser tabs to request http://localhost:5000/home/index and http://localhost:5000/home/list, you will see that each window receives its own cached content with its own expiration, since each request produces a different action routing value.

■ **Tip** If you are using Razor Pages, then you can achieve the same effect using `page` as the value matched by the routing system.

Using the Hosting Environment Tag Helper

The `EnvironmentTagHelper` class is applied to the custom `environment` element and determines whether a region of content is included in the HTML sent to the browser-based on the hosting environment, which I described in Chapters 15 and 16. The `environment` element relies on the `names` attribute, which I have described in Table 26-9.

Table 26-9. *The Built-in Tag Helper Attribute for environment Elements*

Name	Description
`names`	This attribute is used to specify a comma-separated list of hosting environment names for which the content contained within the `environment` element will be included in the HTML sent to the client.

In Listing 26-22, I have added `environment` elements to the shared layout including different content in the view for the development and production hosting environments.

Listing 26-22. Using environment in the _SimpleLayout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="m-2">
        <environment names="development">
            <h2 class="bg-info text-white m-2 p-2">This is Development</h2>
        </environment>
        <environment names="production">
            <h2 class="bg-danger text-white m-2 p-2">This is Production</h2>
        </environment>
        @RenderBody()
    </div>
</body>
</html>
```

The `environment` element checks the current hosting environment name and either includes the content it contains or omits it (the `environment` element itself is always omitted from the HTML sent to the client). Figure 26-9 shows the output for the development and production environments. (See Chapter 15 for details of how to set the environment.)

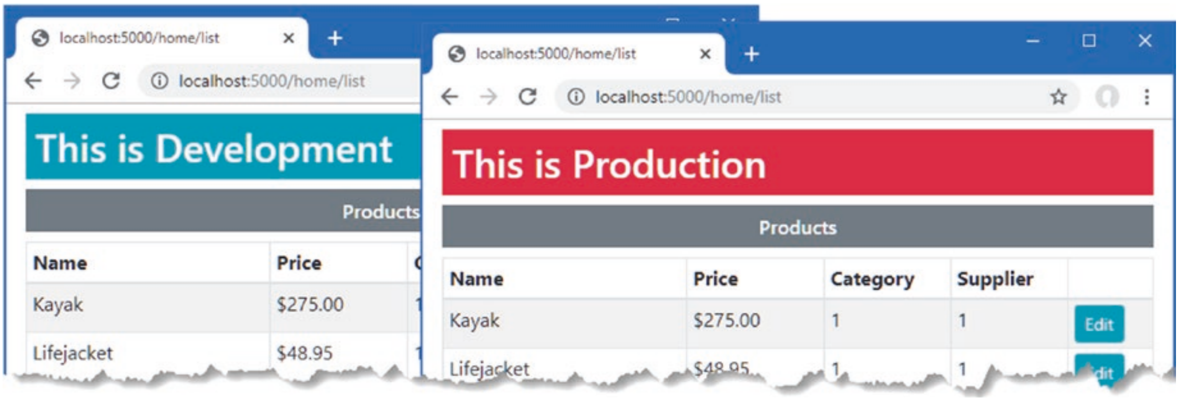

Figure 26-9. *Managing content using the hosting environment*

Summary

In this chapter, I described the basic built-in tag helpers and explained how they are used to transform anchor, `link`, `script`, and image elements. I also explained how to cache sections of content and how to render content based on the application's environment. In the next chapter, I describe the tag helpers that ASP.NET Core provides for working with HTML forms.

CHAPTER 27

Using the Forms Tag Helpers

In this chapter, I describe the built-in tag helpers that are used to create HTML forms. These tag helpers ensure forms are submitted to the correct action or page handler method and that elements accurately represent specific model properties. Table 27-1 puts the form tag helpers in context.

Table 27-1. *Putting Form Tag Helpers in Context*

Question	Answer
What are they?	These built-in tag helpers transform HTML form elements.
Why are they useful?	These tag helpers ensure that HTML forms reflect the application's routing configuration and data model.
How are they used?	Tag helpers are applied to HTML elements using `asp-*` attributes.
Are there any pitfalls or limitations?	These tag helpers are reliable and predictable and present no serious issues.
Are there any alternatives?	You don't have to use tag helpers and can define forms without them if you prefer.

Table 27-2 summarizes the chapter.

Table 27-2. *Chapter Summary*

Problem	Solution	Listing
Specifying how a form will be submitted	Use the form tag helper attributes	10–13
Transforming `input` elements	Use the input tag helper attributes	14–22
Transforming `label` elements	Use the label tag helper attributes	23
Populating `select` elements	Use the select tag helper attributes	24–26
Transforming text areas	Use the text area tag helper attributes	27
Protecting against cross-site request forgery	Enable the anti-forgery feature	28–32

Preparing for This Chapter

This chapter uses the WebApp project from Chapter 26. To prepare for this chapter, replace the contents of the `_SimpleLayout.cshtml` file in the `Views/Shared` folder with those shown in Listing 27-1.

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-3`. See Chapter 1 for how to get help if you have problems running the examples.

A. Freeman, *Pro ASP.NET Core 3*, https://doi.org/10.1007/978-1-4842-5440-0_27

Listing 27-1. The Contents of the _SimpleLayout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="m-2">
        @RenderBody()
    </div>
</body>
</html>
```

This chapter uses controller views and Razor Pages to present similar content. To differentiate more readily between controllers and pages, add the route shown in Listing 27-2 to the `Startup` class.

Listing 27-2. Adding a Route in the Startup.cs File in the WebApp Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using WebApp.Models;

namespace WebApp {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        public IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<DataContext>(opts => {
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:ProductConnection"]);
                opts.EnableSensitiveDataLogging(true);
            });
            services.AddControllersWithViews().AddRazorRuntimeCompilation();
            services.AddRazorPages().AddRazorRuntimeCompilation();
            services.AddSingleton<CitiesData>();
        }

        public void Configure(IApplicationBuilder app, DataContext context) {
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseRouting();
            app.UseEndpoints(endpoints => {
                endpoints.MapControllers();
                endpoints.MapControllerRoute("forms",
                    "controllers/{controller=Home}/{action=Index}/{id?}");
                endpoints.MapDefaultControllerRoute();
                endpoints.MapRazorPages();
            });
```

```
            SeedData.SeedDatabase(context);
        }
    }
}
```

The new route introduces a static path segment that makes it obvious that a URL targets a controller.

Dropping the Database

Open a new PowerShell command prompt, navigate to the folder that contains the `WebApp.csproj` file, and run the command shown in Listing 27-3 to drop the database.

Listing 27-3. Dropping the Database

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

Running the Example Application

Select Start Without Debugging or Run Without Debugging from the Debug menu or use the PowerShell command prompt to run the command shown in Listing 27-4.

Listing 27-4. Running the Example Application

```
dotnet run
```

Use a browser to request `http://localhost:5000/controllers/home/list`, which will display a list of products, as shown in Figure 27-1.

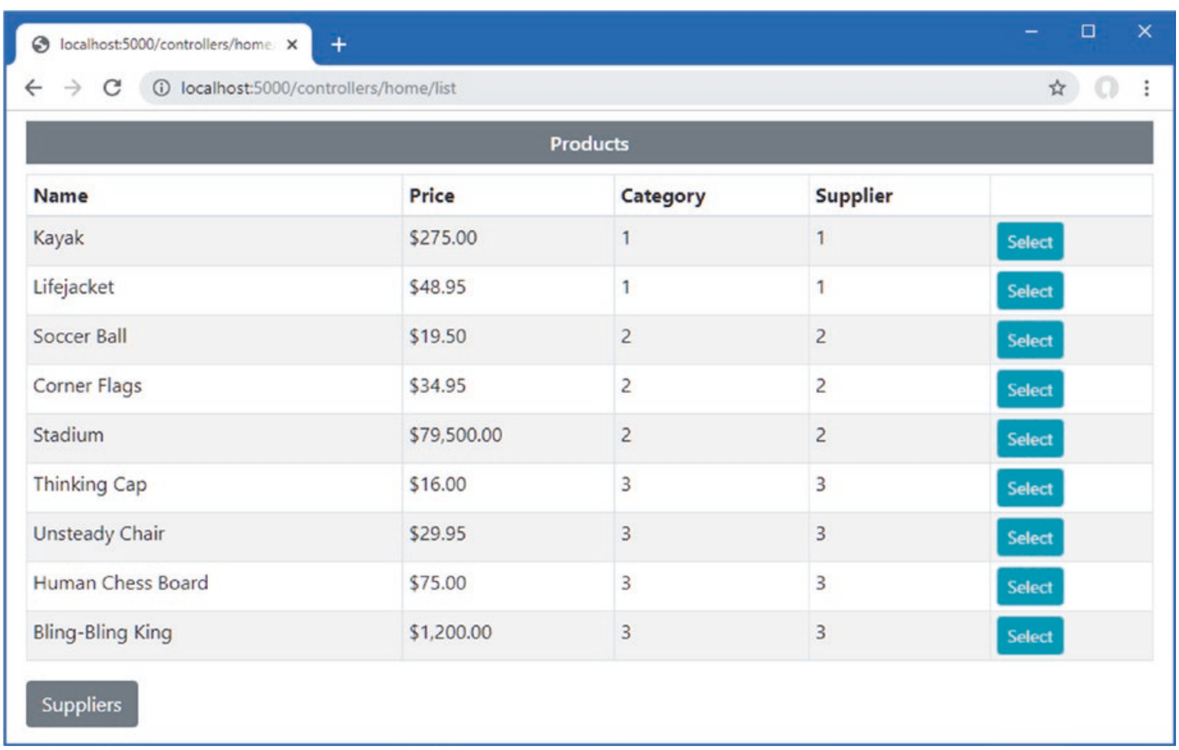

Figure 27-1. *Running the example application*

Understanding the Form Handling Pattern

Most HTML forms exist within a well-defined pattern, shown in Figure 27-2. First, the browser sends an HTTP GET request, which results in an HTML response containing a form, making it possible for the user to provide the application with data. The user clicks a button that submits the form data with an HTTP POST request, which allows the application to receive and process the user's data. Once the data has been processed, a response is sent that redirects the browser to a URL that provides confirmation of the user's actions.

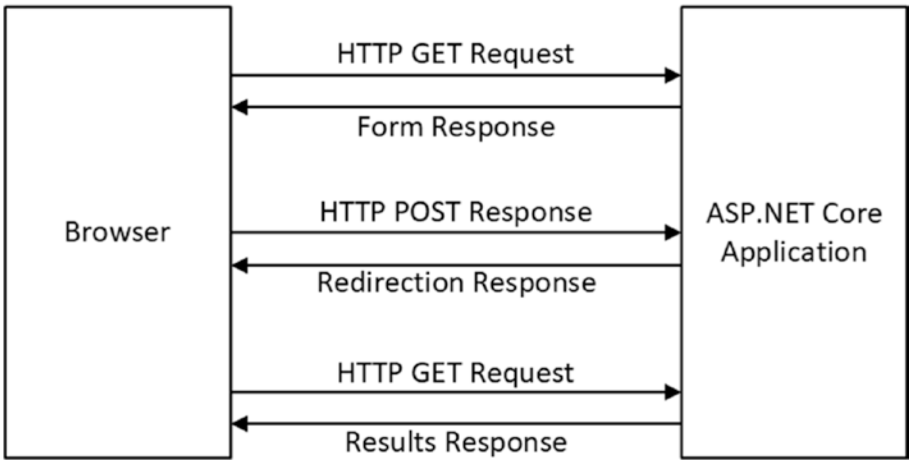

Figure 27-2. *The HTML Post/Redirect/Get pattern*

This is known as the Post/Redirect/Get pattern, and the redirection is important because it means the user can click the browser's reload button without sending another POST request, which can lead to inadvertently repeating an operation.

In the sections that follow, I show how to follow the pattern with controllers and Razor Pages. I start with a basic implementation of the pattern and then demonstrate improvements using tag helpers and, in Chapter 28, the model binding feature.

Creating a Controller to Handle Forms

Controllers that handle forms are created by combining features described in earlier chapters. Add a class file named `FormController.cs` to the `Controllers` folder with the code shown in Listing 27-5.

Listing 27-5. The Contents of the FormController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using System.Threading.Tasks;
using WebApp.Models;

namespace WebApp.Controllers {

    public class FormController : Controller {
        private DataContext context;

        public FormController(DataContext dbContext) {
            context = dbContext;
        }

        public async Task<IActionResult> Index(long id = 1) {
            return View("Form", await context.Products.FindAsync(id));
        }
```

```
        [HttpPost]
        public IActionResult SubmitForm() {
            foreach (string key in Request.Form.Keys
                    .Where(k => !k.StartsWith("_"))) {
                TempData[key] = string.Join(", ", Request.Form[key]);
            }
            return RedirectToAction(nameof(Results));
        }

        public IActionResult Results() {
            return View(TempData);
        }
    }
}
```

The Index action method selects a view named Form, which will render an HTML form to the user. When the user submits the form, it will be received by the SubmitForm action, which has been decorated with the HttpPost attribute so that it can only receive HTTP POST requests. This action method processes the HTML form data available through the HttpRequest.Form property so that it can be stored using the temp data feature. The temp data feature can be used to pass data from one request to another but can be used only to store simple data types. Each form data value is presented as a string array, which I convert to a single comma-separated string for storage. The browser is redirected to the Results action method, which selects the default view and provides the temp data as the view model.

■ **Tip** Only form data values whose name doesn't begin with an underscore are displayed. I explain why in the "Using the Anti-forgery Feature" section, later in this chapter.

To provide the controller with views, create the Views/Form folder and add to it a Razor view file named Form.cshtml with the content shown in Listing 27-6.

Listing 27-6. The Contents of the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<form action="/controllers/form/submitform" method="post">
    <div class="form-group">
        <label>Name</label>
        <input class="form-control" name="Name" value="@Model.Name" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

This view contains a simple HTML form that is configured to submit its data to the SubmitForm action method using a POST request. The form contains an input element whose value is set using a Razor expression. Next, add a Razor view named Results.cshtml to the Views/Forms folder with the content shown in Listing 27-7.

Listing 27-7. The Contents of the Results.cshtml File in the Views/Form Folder

```
@model TempDataDictionary
@{ Layout = "_SimpleLayout"; }

<table class="table table-striped table-bordered table-sm">
    <thead>
        <tr class="bg-primary text-white text-center">
            <th colspan="2">Form Data</th>
        </tr>
```

```
    </thead>
    <tbody>
        @foreach (string key in Model.Keys) {
            <tr>
                <th>@key</th>
                <td>@Model[key]</td>
            </tr>
        }
    </tbody>
</table>
<a class="btn btn-primary" asp-action="Index">Return</a>
```

This view displays the form data back to the user. I'll show you how to process form data in more useful ways in Chapter 31, but for this chapter the focus is on creating the forms, and seeing the data contained in the form is enough to get started.

Restart ASP.NET Core and use a browser to request `http://localhost:5000/controllers/form` to see the HTML form. Enter a value into the text field and click Submit to send a POST request, which will be handled by the `SubmitForm` action. The form data will be stored as temp data, and the browser will be redirected, producing the response shown in Figure 27-3.

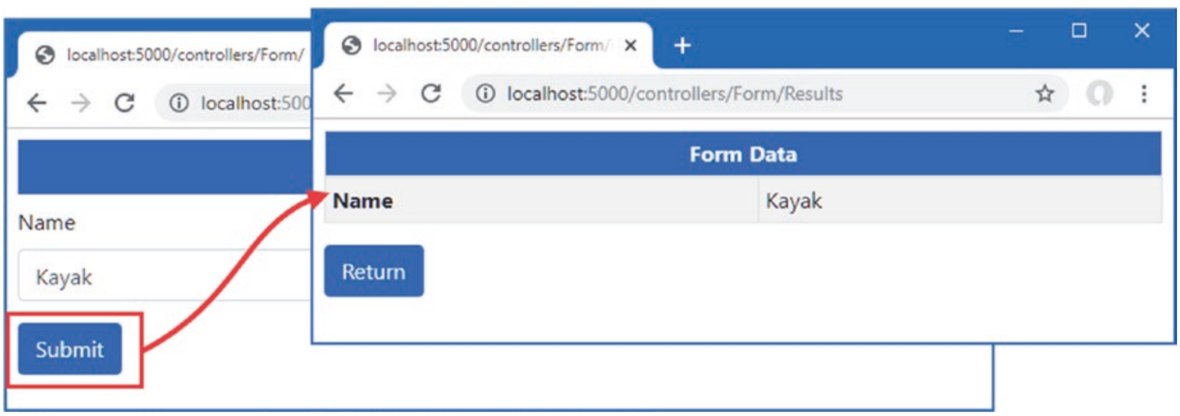

Figure 27-3. *Using a controller to render and process an HTML form*

Creating a Razor Page to Handle Forms

The same pattern can be implemented using Razor Pages. One page is required to render and process the form data, and a second page displays the results. Add a Razor Page named `FormHandler.cshtml` to the Pages folder with the contents shown in Listing 27-8.

Listing 27-8. The Contents of the FormHandler.cshtml File in the Pages Folder

```
@page "/pages/form/{id:long?}"
@model FormHandlerModel
@using Microsoft.AspNetCore.Mvc.RazorPages

<div class="m-2">
    <h5 class="bg-primary text-white text-center p-2">HTML Form</h5>
    <form asp-page="FormHandler" method="post">
        <div class="form-group">
            <label>Name</label>
            <input class="form-control" name="Name" value="@Model.Product.Name" />
        </div>
        <button type="submit" class="btn btn-primary">Submit</button>
    </form>
</div>
```

```
@functions {

    [IgnoreAntiforgeryToken]
    public class FormHandlerModel : PageModel {
        private DataContext context;

        public FormHandlerModel(DataContext dbContext) {
            context = dbContext;
        }

        public Product Product { get; set; }

        public async Task OnGetAsync(long id = 1) {
            Product = await context.Products.FindAsync(id);
        }

        public IActionResult OnPost() {
            foreach (string key in Request.Form.Keys
                    .Where(k => !k.StartsWith("_"))) {
                TempData[key] = string.Join(", ", Request.Form[key]);
            }
            return RedirectToPage("FormResults");
        }
    }
}
```

The `OnGetAsync` handler methods retrieves a `Product` from the database, which is used by the view to set the value for the `input` element in the HTML form. The form is configured to send an HTTP POST request that will be processed by the `OnPost` handler method. The form data is stored as temp data, and the browser is sent a redirection to a form named `FormResults`. To create the page that the browser will be redirected to, add a Razor Page named `FormResults.cshtml` to the `Pages` folder with the content shown in Listing 27-9.

■ **Tip** The page model class in Listing 27-8 is decorated with the `IgnoreAntiforgeryToken` attribute, which is described in the "Using the Anti-forgery Feature" section.

Listing 27-9. The Contents of the FormResults.cshtml File in the Pages Folder

```
@page "/pages/results"

<div class="m-2">
    <table class="table table-striped table-bordered table-sm">
        <thead>
            <tr class="bg-primary text-white text-center">
                <th colspan="2">Form Data</th>
            </tr>
        </thead>
        <tbody>
            @foreach (string key in TempData.Keys) {
                <tr>
                    <th>@key</th>
                    <td>@TempData[key]</td>
                </tr>
            }
        </tbody>
    </table>
    <a class="btn btn-primary" asp-page="FormHandler">Return</a>
</div>
```

No code is required for this page, which accesses temp data directly and displays it in a table. Use a browser to navigate to `http://localhost:5000/pages/form`, enter a value into the text field, and click the Submit button. The form data will be processed by the `OnPost` method defined in Listing 27-9, and the browser will be redirected to `/pages/results`, which displays the form data, as shown in Figure 27-4.

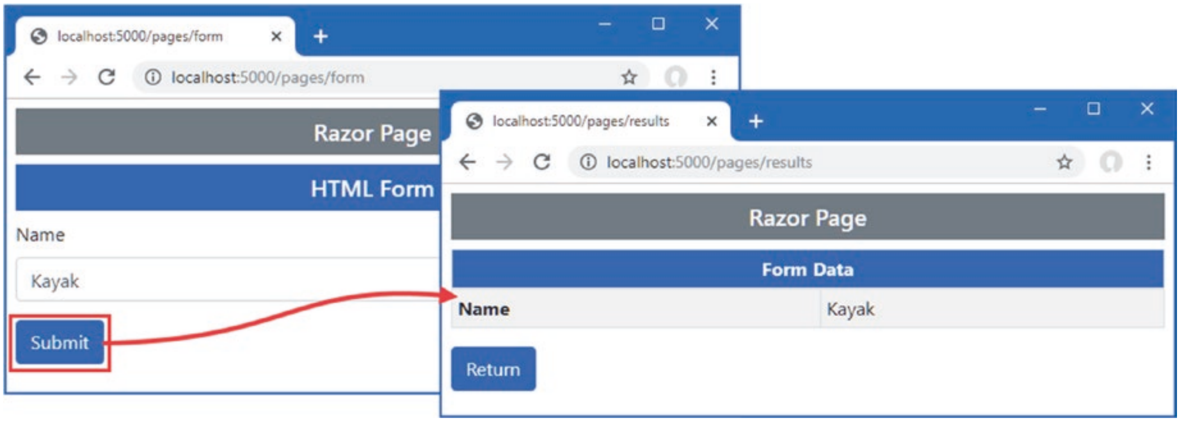

Figure 27-4. *Using Razor Pages to render and process an HTML form*

Using Tag Helpers to Improve HTML Forms

The examples in the previous section show the basic mechanisms for dealing with HTML forms, but ASP.NET Core includes tag helpers that transform form elements. In the sections that follow, I describe the tag helpers and demonstrate their use.

Working with Form Elements

The `FormTagHelper` class is the built-in tag helper for `form` elements and is used to manage the configuration of HTML forms so that they target the right action or page handler without the need to hard-code URLs. This tag helper supports the attributes described in Table 27-3.

Table 27-3. *The Built-in Tag Helper Attributes for Form Elements*

Name	Description
`asp-controller`	This attribute is used to specify the `controller` value to the routing system for the `action` attribute URL. If omitted, then the controller rendering the view will be used.
`asp-action`	This attribute is used to specify the action method for the `action` value to the routing system for the `action` attribute URL. If omitted, then the action rendering the view will be used.
`asp-page`	This attribute is used to specify the name of a Razor Page.
`asp-page-handler`	This attribute is used to specify the name of the handler method that will be used to process the request. You can see an example of this attribute in the SportsStore application in Chapter 9.
`asp-route-*`	Attributes whose name begins with `asp-route-` are used to specify additional values for the action attribute URL so that the `asp-route-id` attribute is used to provide a value for the `id` segment to the routing system.
`asp-route`	This attribute is used to specify the name of the route that will be used to generate the URL for the `action` attribute.
`asp-antiforgery`	This attribute controls whether anti-forgery information is added to the view, as described in the "Using the Anti-forgery Feature" section.
`asp-fragment`	This attribute specifies a fragment for the generated URL.

Setting the Form Target

The FormTagHelper transforms form elements so they target an action method or Razor Page without the need for hard-coded URLs. The attributes supported by this tag helper work in the same way as for anchor elements, described in Chapter 26, and use attributes to provide values that help generate URLs through the ASP.NET Core routing system. Listing 27-10 modifies the form element in the Form view to apply the tag helper.

■ **Note** If a form element is defined without a method attribute, then the tag helper will add one with the post value, meaning that the form will be submitted using an HTTP POST request. This can lead to surprising results if you omitted the method attribute because you expect the browser to follow the HTML5 specification and send the form using an HTTP GET request. It is a good idea to always specify the method attribute so that it is obvious how the form should be submitted.

Listing 27-10. Using a Tag Helper in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<form asp-action="submitform" method="post">
    <div class="form-group">
        <label>Name</label>
        <input class="form-control" name="Name" value="@Model.Name" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

The asp-action attribute is used to specify the name of the action that will receive the HTTP request. The routing system is used to generate the URLs, just as for the anchor elements described in Chapter 26. The asp-controller attribute has not been used in Listing 27-10, which means the controller that rendered the view will be used in the URL.

The asp-page attribute is used to select a Razor Page as the target for the form, as shown in Listing 27-11.

Listing 27-11. Setting the Form Target in the FormHandler.cshtml File in the Pages Folder

```
...
<div class="m-2">
    <h5 class="bg-primary text-white text-center p-2">HTML Form</h5>
    <form asp-page="FormHandler" method="post">
        <div class="form-group">
            <label>Name</label>
            <input class="form-control" name="Name" value="@Model.Product.Name" />
        </div>
        <button type="submit" class="btn btn-primary">Submit</button>
    </form>
</div>
...
```

Use a browser to navigate to http://localhost:5000/controllers/form and examine the HTML received by the browser; you will see that the tag helper as added the action attribute to the form element like this:

```
...
<form method="post" action="controllers/Form/submitform">
...
```

This is the same URL that I defined statically when I created the view but with the advantage that changes to the routing configuration will be reflected automatically in the form URL. Request `http://localhost:5000/pages/form`, and you will see that the `form` element has been transformed to target the page URL, like this:

```
...
<form method="post" action="/pages/form">
...
```

Transforming Form Buttons

The buttons that send forms can be defined outside of the `form` element. In these situations, the button has a `form` attribute whose value corresponds to the `id` attribute of the form element it relates to and a `formaction` attribute that specifies the target URL for the form.

The tag helper will generate the `formaction` attribute through the `asp-action`, `asp-controller`, or `asp-page` attributes, as shown in Listing 27-12.

Listing 27-12. Transforming a Button in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<form asp-action="submitform" method="post" id="htmlform">
    <div class="form-group">
        <label>Name</label>
        <input class="form-control" name="Name" value="@Model.Name" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>

<button form="htmlform" asp-action="submitform" class="btn btn-primary mt-2">
    Sumit (Outside Form)
</button>
```

The value of the `id` attribute added to the `form` element is used by the `button` as the value of the `form` attribute, which tells the browser which form to submit when the button is clicked. The attributes described in Table 27-3 are used to identify the target for the form, and the tag helper will use the routing system to generate a URL when the view is rendered. Listing 27-13 applies the same technique to the Razor Page.

Listing 27-13. Transforming a Button in the FormHandler.cshtml File in the Pages Folder

```
...
<div class="m-2">
    <h5 class="bg-primary text-white text-center p-2">HTML Form</h5>
    <form asp-page="FormHandler" method="post" id="htmlform">
        <div class="form-group">
            <label>Name</label>
            <input class="form-control" name="Name" value="@Model.Product.Name" />
        </div>
        <button type="submit" class="btn btn-primary">Submit</button>
    </form>
    <button form="htmlform" asp-page="FormHandler" class="btn btn-primary mt-2">
        Sumit (Outside Form)
    </button>
</div>
...
```

Use a browser to request `http://localhost:5000/controllers/form` or `http://localhost:5000/pages/form` and inspect the HTML sent to the browser. You will see the `button` element outside of the form has been transformed like this:

```
...
<button form="htmlform" class="btn btn-primary mt-2"
        formaction="/controllers/Form/submitform">
    Sumit (Outside Form)
</button>
...
```

Clicking the button submits the form, just as for a button that is defined within the form element, as shown in Figure 27-5.

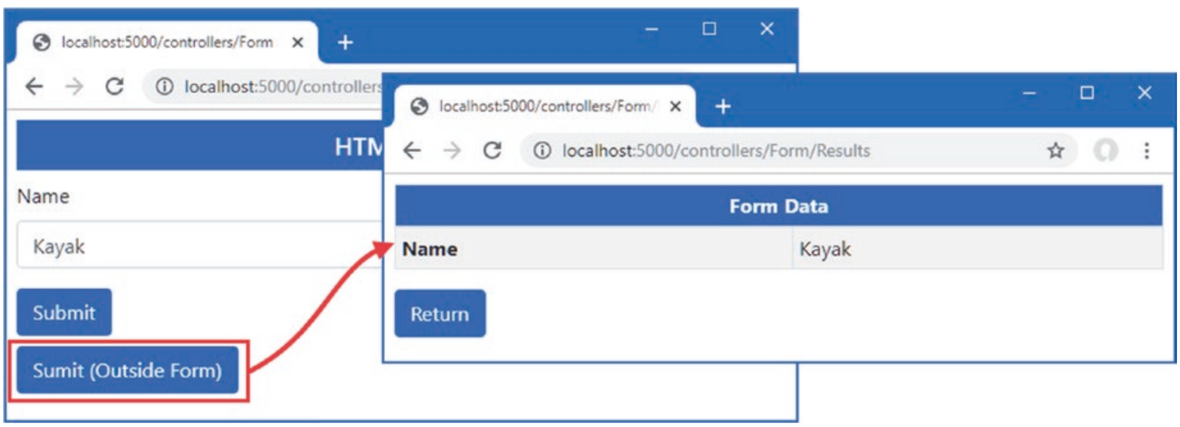

Figure 27-5. *Defining a button outside of a form element*

Working with input Elements

The `input` element is the backbone of HTML forms and provides the main means by which a user can provide an application with unstructured data. The `InputTagHelper` class is used to transform `input` elements so they reflect the data type and format of a view model property they are used to gather, using the attributes described in Table 27-4.

Table 27-4. *The Built-in Tag Helper Attributes for input Elements*

Name	Description
`asp-for`	This attribute is used to specify the view model property that the `input` element represents.
`asp-format`	This attribute is used to specify a format used for the value of the view model property that the `input` element represents.

The `asp-for` attribute is set to the name of a view model property, which is then used to set the `name`, `id`, `type`, and `value` attributes of the `input` element. Listing 27-14 modifies the `input` element in the controller view to use the `asp-for` attribute.

Listing 27-14. Configuring an input Element in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<form asp-action="submitform" method="post" id="htmlform">
    <div class="form-group">
        <label>Name</label>
```

```
        <input class="form-control" asp-for="Name" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>

<button form="htmlform" asp-action="submitform" class="btn btn-primary mt-2">
    Sumit (Outside Form)
</button>
```

This tag helper uses a model expression, described in Listing 27-14, which is why the value for the `asp-for` attribute is specified without the `@` character. If you inspect the HTML the application returns when using a browser to request `http://localhost:5000/controllers/form`, you will see the tag helper has transformed the `input` element like this:

```
...
<div class="form-group">
    <label>Name</label>
    <input class="form-control" type="text" id="Name" name="Name" value="Kayak">
</div>
...
```

The values for the `id` and `name` attributes are obtained through the model expression, ensuring that you don't introduce typos when creating the form. The other attributes are more complex and are described in the sections that follow.

SELECTING MODEL PROPERTIES IN RAZOR PAGES

The `asp-for` attribute for this and the other tag helpers described in this chapter can be used for Razor Pages, but the value for the `name` and `id` attributes in the transformed element includes the name of the page model property. For example, this element selects the `Name` property through the page model's `Product` property:

```
...
<input class="form-control" asp-for="Product.Name" />
...
```

The transformed element will have the following `id` and `name` attributes:

```
...
<input class="form-control" type="text" id="Product_Name" name="Product.Name" >
...
```

This difference is important when using the model binding feature to receive form data, as described in Chapter 28.

Transforming the input Element type Attribute

The `input` element's `type` attribute tells the browser how to display the element and how it should restrict the values the user enters. The `input` element in Listing 27-14 is configured to the `text` type, which is the default `input` element type and offers no restrictions. Listing 27-15 adds another `input` element to the form, which will provide a more useful demonstration of how the `type` attribute is handled.

Listing 27-15. Adding an input Element in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<form asp-action="submitform" method="post" id="htmlform">
    <div class="form-group">
        <label>Id</label>
        <input class="form-control" asp-for="ProductId" />
    </div>
```

```
    <div class="form-group">
        <label>Name</label>
        <input class="form-control" asp-for="Name" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>

<button form="htmlform" asp-action="submitform" class="btn btn-primary mt-2">
    Sumit (Outside Form)
</button>
```

The new element uses the `asp-for` attribute to select the view model's `ProductId` property. Use a browser to request `http://localhost:5000/controllers/form` to see how the tag helper has transformed the element.

```
...
<div class="form-group">
    <label>Id</label>
    <input class="form-control" type="number" data-val="true"
        data-val-required="The ProductId field is required."
        id="ProductId" name="ProductId" value="1">
</div>
...
```

The value of the `type` attribute is determined by the type of the view model property specified by the `asp-for` attribute. The type of the `ProductId` property is the C# `long` type, which has led the tag helper to set the `input` element's type attribute to `number`, which restricts the element so it will accept only numeric characters. The `data-val` and `data-val-required` attributes are added to the `input` element to assist with validation, which is described in Chapter 29. Table 27-5 describes how different C# types are used to set the type attribute of `input` elements.

■ **Note** There is latitude in how the `type` attribute is interpreted by browsers. Not all browsers respond to all the `type` values that are defined in the HTML5 specification, and when they do, there are differences in how they are implemented. The `type` attribute can be a useful hint for the kind of data that you are expecting in a form, but you should use the model validation feature to ensure that users provide usable data, as described in Chapter 29.

Table 27-5. *C# Property Types and the Input Type Elements They Generate*

C# Type	input Element type Attribute
`byte`, `sbyte`, `int`, `uint`, `short`, `ushort`, `long`, `ulong`	`number`
`float`, `double`, `decimal`	`text`, with additional attributes for model validation, as described in Chapter 29
`bool`	`checkbox`
`string`	`text`
`DateTime`	`datetime`

The `float`, `double`, and `decimal` types produce `input` elements whose `type` is `text` because not all browsers allow the full range of characters that can be used to express legal values of this type. To provide feedback to the user, the tag helper adds attributes to the `input` element that are used with the validation features described in Chapter 29.

You can override the default mappings shown in Table 27-5 by explicitly defining the `type` attribute on `input` elements. The tag helper won't override the value you define, which allows you to specify a `type` attribute value.

The drawback of this approach is that you must remember to set the `type` attribute in all the views where `input` elements are generated for a given model property. A more elegant—and reliable approach—is to apply one of the attributes described in Table 27-6 to the property in the C# model class.

■ **Tip** The tag helper will set the `type` attribute of `input` elements to `text` if the model property isn't one of the types in Table 27-5 and has not been decorated with an attribute.

Table 27-6. *The Input Type Elements Attributes*

Attribute	input Element type Attribute
[HiddenInput]	hidden
[Text]	text
[Phone]	tel
[Url]	url
[EmailAddress]	email
[DataType(DataType.Password)]	password
[DataType(DataType.Time)]	time
[DataType(DataType.Date)]	date

Formatting input Element Values

When the action method provides the view with a view model object, the tag helper uses the value of the property given to the `asp-for` attribute to set the `input` element's `value` attribute. The `asp-format` attribute is used to specify how that data value is formatted. To demonstrate the default formatting, Listing 27-16 adds a new `input` element to the `Form` view.

Listing 27-16. Adding an Element in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<form asp-action="submitform" method="post" id="htmlform">
    <div class="form-group">
        <label>Id</label>
        <input class="form-control" asp-for="ProductId" />
    </div>
    <div class="form-group">
        <label>Name</label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label>Price</label>
        <input class="form-control" asp-for="Price" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>

<button form="htmlform" asp-action="submitform" class="btn btn-primary mt-2">
    Sumit (Outside Form)
</button>
```

Use a browser to navigate to `http://localhost:5000/controllers/form/index/5` and examine the HTML the browser receives. By default, the `value` of the `input` element is set using the value of the model property, like this:

```
...
<input class="form-control" type="text" data-val="true"
    data-val-number="The field Price must be a number."
    data-val-required="The Price field is required."
    id="Price" name="Price" value="79500.00">
...
```

This format, with two decimal places, is how the value is stored in the database. In Chapter 26, I used the `Column` attribute to select a SQL type to store `Price` values, like this:

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

This type specifies a maximum precision of eight digits, two of which will appear after the decimal place. This allows a maximum value of 999,999.99, which is enough to represent prices for most online stores. The `asp-format` attribute accepts a format string that will be passed to the standard C# string formatting system, as shown in Listing 27-17.

Listing 27-17. Formatting a Data Value in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<form asp-action="submitform" method="post" id="htmlform">
    <div class="form-group">
        <label>Id</label>
        <input class="form-control" asp-for="ProductId" />
    </div>
    <div class="form-group">
        <label>Name</label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label>Price</label>
        <input class="form-control" asp-for="Price" asp-format="{0:#,###.00}" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>

<button form="htmlform" asp-action="submitform" class="btn btn-primary mt-2">
    Sumit (Outside Form)
</button>
```

The attribute value is used verbatim, which means you must include the curly brace characters and the `0:` reference, as well as the format you require. Refresh the browser, and you will see that the value for the `input` element has been formatted, like this:

```
...
<input class="form-control" type="text" data-val="true"
    data-val-number="The field Price must be a number."
    data-val-required="The Price field is required."
    id="Price" name="Price" value="79,500.00">
...
```

This feature should be used with caution because you must ensure that the rest of the application is configured to support the format you use and that the format you create contains only legal characters for the `input` element type.

Applying Formatting via the Model Class

If you always want to use the same formatting for a model property, then you can decorate the C# class with the DisplayFormat attribute, which is defined in the System.ComponentModel.DataAnnotations namespace. The DisplayFormat attribute requires two arguments to format a data value: the DataFormatString argument specifies the formatting string, and setting the ApplyFormatInEditMode to true specifies that formatting should be used when values are being applied to elements used for editing, including the input element. Listing 27-18 applies the attribute to the Price property of the Product class, specifying a different formatting string from earlier examples.

Listing 27-18. Applying a Formatting Attribute to the Product.cs File in the Models Folder

```
using System.ComponentModel.DataAnnotations.Schema;
using System.ComponentModel.DataAnnotations;

namespace WebApp.Models {
    public class Product {

        public long ProductId { get; set; }

        public string Name { get; set; }

        [Column(TypeName = "decimal(8, 2)")]
        [DisplayFormat(DataFormatString = "{0:c2}", ApplyFormatInEditMode = true)]
        public decimal Price { get; set; }

        public long CategoryId { get; set; }
        public Category Category { get; set; }

        public long SupplierId { get; set; }
        public Supplier Supplier { get; set; }
    }
}
```

The asp-format attribute takes precedence over the DisplayFormat attribute, so I have removed the attribute from the view, as shown in Listing 27-19.

Listing 27-19. Removing an Attribute in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<form asp-action="submitform" method="post" id="htmlform">
    <div class="form-group">
        <label>Id</label>
        <input class="form-control" asp-for="ProductId" />
    </div>
    <div class="form-group">
        <label>Name</label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label>Price</label>
        <input class="form-control" asp-for="Price" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>

<button form="htmlform" asp-action="submitform" class="btn btn-primary mt-2">
    Sumit (Outside Form)
</button>
```

Restart ASP.NET Core and use a browser to request `http://localhost:5000/controllers/form/index/5`, and you will see that the formatting string defined by the attribute has been applied, as shown in Figure 27-6.

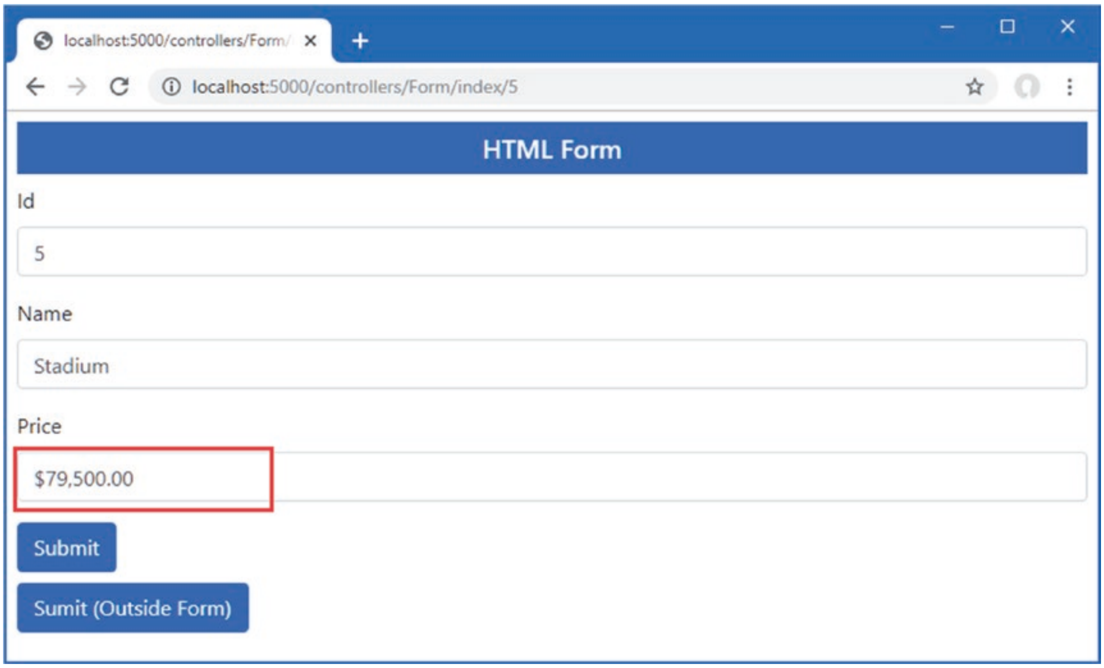

Figure 27-6. *Formatting data values*

I chose this format to demonstrate the way the formatting attribute works, but, as noted previously, care must be taken to ensure that the application is able to process the formatted values using the model binding and validation features described in Chapters 28 and 29.

Displaying Values from Related Data in input Elements

When using Entity Framework Core, you will often need to display data values that are obtained from related data, which is easily done using the `asp-for` attribute because a model expression allows the nested navigation properties to be selected. First, Listing 27-20 includes related data in the view model object provided to the view.

Listing 27-20. Including Related Data in the FormController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using System.Threading.Tasks;
using WebApp.Models;
using Microsoft.EntityFrameworkCore;

namespace WebApp.Controllers {

    public class FormController : Controller {
        private DataContext context;

        public FormController(DataContext dbContext) {
            context = dbContext;
        }

        public async Task<IActionResult> Index(long id = 1) {
            return View("Form", await context.Products.Include(p => p.Category)
                .Include(p => p.Supplier).FirstAsync(p => p.ProductId == id));
        }
```

```
        [HttpPost]
        public IActionResult SubmitForm() {
            foreach (string key in Request.Form.Keys
                    .Where(k => !k.StartsWith("_"))) {
                TempData[key] = string.Join(", ", Request.Form[key]);
            }
            return RedirectToAction(nameof(Results));
        }

        public IActionResult Results() {
            return View(TempData);
        }
    }
}
```

Notice that I don't need to worry about dealing with circular references in the related data because the view model object isn't serialized. The circular reference issue is important only for web service controllers. In Listing 27-21, I have updated the `Form` view to include `input` elements that use the `asp-for` attribute to select related data.

Listing 27-21. Displaying Related Data in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<form asp-action="submitform" method="post" id="htmlform">
    <div class="form-group">
        <label>Id</label>
        <input class="form-control" asp-for="ProductId" />
    </div>
    <div class="form-group">
        <label>Name</label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label>Price</label>
        <input class="form-control" asp-for="Price" />
    </div>
    <div class="form-group">
        <label>Category</label>
        <input class="form-control" asp-for="Category.Name" />
    </div>
    <div class="form-group">
        <label>Supplier</label>
        <input class="form-control" asp-for="Supplier.Name" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>

<button form="htmlform" asp-action="submitform" class="btn btn-primary mt-2">
    Sumit (Outside Form)
</button>
```

The value of the asp-for attribute is expressed relative to the view model object and can include nested properties, allowing me to select the Name properties of the related objects that Entity Framework Core has assigned to the Category and Supplier navigation properties. The same technique is used in Razor Pages, except that the properties are expressed relative to the page model object, as shown in Listing 27-22.

Listing 27-22. Displayed Related Data in the FormHandler.cshtml File in the Pages Folder

```
@page "/pages/form/{id:long?}"
@model FormHandlerModel
@using Microsoft.AspNetCore.Mvc.RazorPages
@using Microsoft.EntityFrameworkCore

<div class="m-2">
    <h5 class="bg-primary text-white text-center p-2">HTML Form</h5>
    <form asp-page="FormHandler" method="post" id="htmlform">
        <div class="form-group">
            <label>Name</label>
            <input class="form-control" asp-for="Product.Name" />
        </div>
        <div class="form-group">
            <label>Price</label>
            <input class="form-control" asp-for="Product.Price" />
        </div>
        <div class="form-group">
            <label>Category</label>
            <input class="form-control" asp-for="Product.Category.Name" />
        </div>
        <div class="form-group">
            <label>Supplier</label>
            <input class="form-control" asp-for="Product.Supplier.Name" />
        </div>
        <button type="submit" class="btn btn-primary">Submit</button>
    </form>
    <button form="htmlform" asp-page="FormHandler" class="btn btn-primary mt-2">
        Sumit (Outside Form)
    </button>
</div>

@functions {

    [IgnoreAntiforgeryToken]
    public class FormHandlerModel : PageModel {
        private DataContext context;

        public FormHandlerModel(DataContext dbContext) {
            context = dbContext;
        }

        public Product Product { get; set; }

        public async Task OnGetAsync(long id = 1) {
            Product = await context.Products.Include(p => p.Category)
                .Include(p => p.Supplier).FirstAsync(p => p.ProductId == id);
        }
```

```
        public IActionResult OnPost() {
            foreach (string key in Request.Form.Keys
                    .Where(k => !k.StartsWith("_"))) {
                TempData[key] = string.Join(", ", Request.Form[key]);
            }
            return RedirectToPage("FormResults");
        }
    }
}
```

To see the effect, restart ASP.NET Core so the changes to the controller take effect, and use a browser to request `http://localhost:5000/controller/form`, which produces the response shown on the left of Figure 27-7. Use the browser to request `http://localhost:5000/pages/form`, and you will see the same features used by the Razor Page, as shown on the right of Figure 27-7.

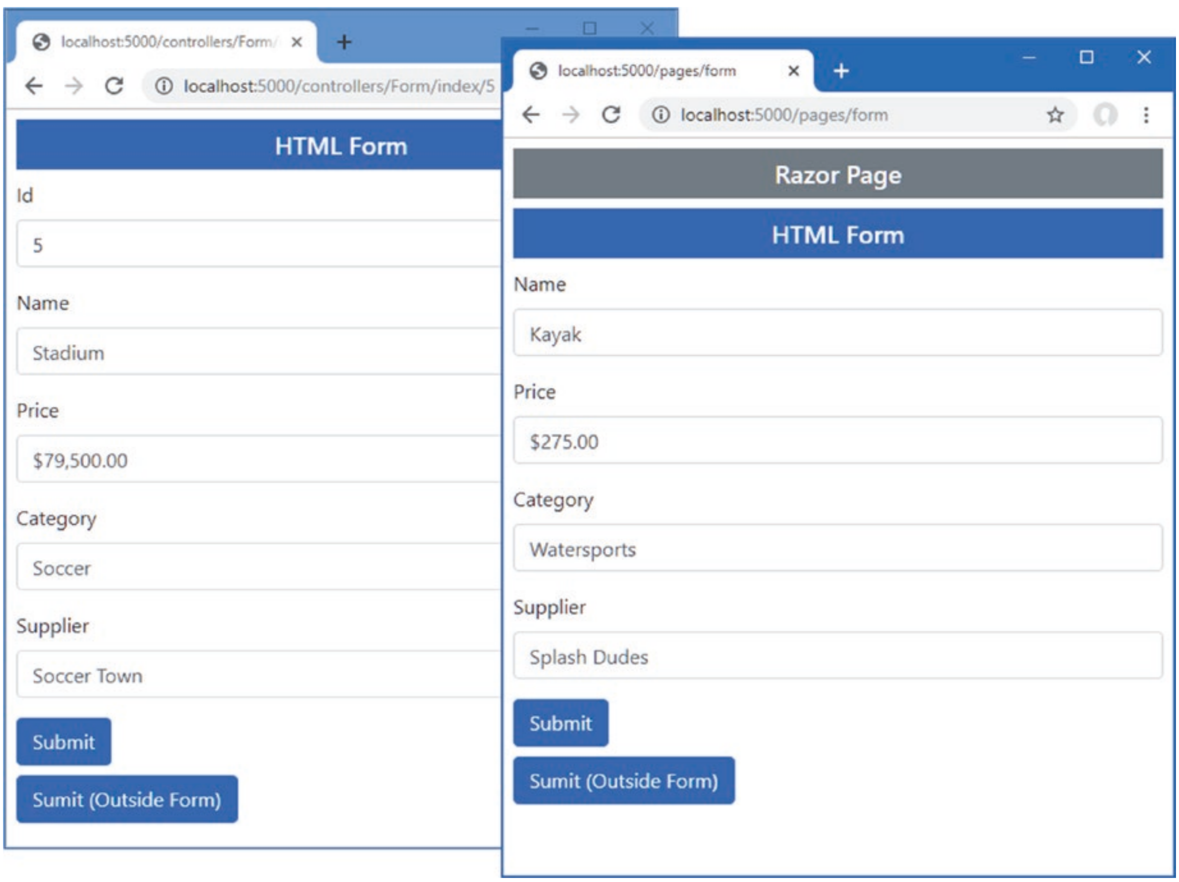

Figure 27-7. *Displaying related data*

Working with label Elements

The `LabelTagHelper` class is used to transform `label` elements so the `for` attribute is set consistently with the approach used to transform `input` elements. Table 27-7 describes the attribute supported by this tag helper.

Table 27-7. *The Built-in Tag Helper Attribute for label Elements*

Name	Description
asp-for	This attribute is used to specify the view model property that the label element describes.

The tag helper sets the content of the label element so that it contains the name of the selected view model property. The tag helper also sets the for attribute, which denotes an association with a specific input element. This aids users who rely on screen readers and allows an input element to gain the focus when its associated label is clicked.

Listing 27-23 applies the asp-for attribute to the Form view to associate each label element with the input element that represents the same view model property.

Listing 27-23. Transforming label Elements in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<form asp-action="submitform" method="post" id="htmlform">
    <div class="form-group">
        <label asp-for="ProductId"></label>
        <input class="form-control" asp-for="ProductId" />
    </div>
    <div class="form-group">
        <label asp-for="Name"></label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label asp-for="Price"></label>
        <input class="form-control" asp-for="Price" />
    </div>
    <div class="form-group">
        <label asp-for="Category.Name">Category</label>
        <input class="form-control" asp-for="Category.Name" />
    </div>
    <div class="form-group">
        <label asp-for="Supplier.Name">Supplier</label>
        <input class="form-control" asp-for="Supplier.Name" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>

<button form="htmlform" asp-action="submitform" class="btn btn-primary mt-2">
    Sumit (Outside Form)
</button>
```

You can override the content for a label element by defining it yourself, which is what I have done for the related data properties in Listing 27-23. The tag helper would have set the content for both these label elements to be Name, which is not a useful description. Defining the element content means the for attribute will be applied, but a more useful name will be displayed to the user. Use a browser to request http://localhost:5000/controllers/form to see the names used for each element, as shown in Figure 27-8.

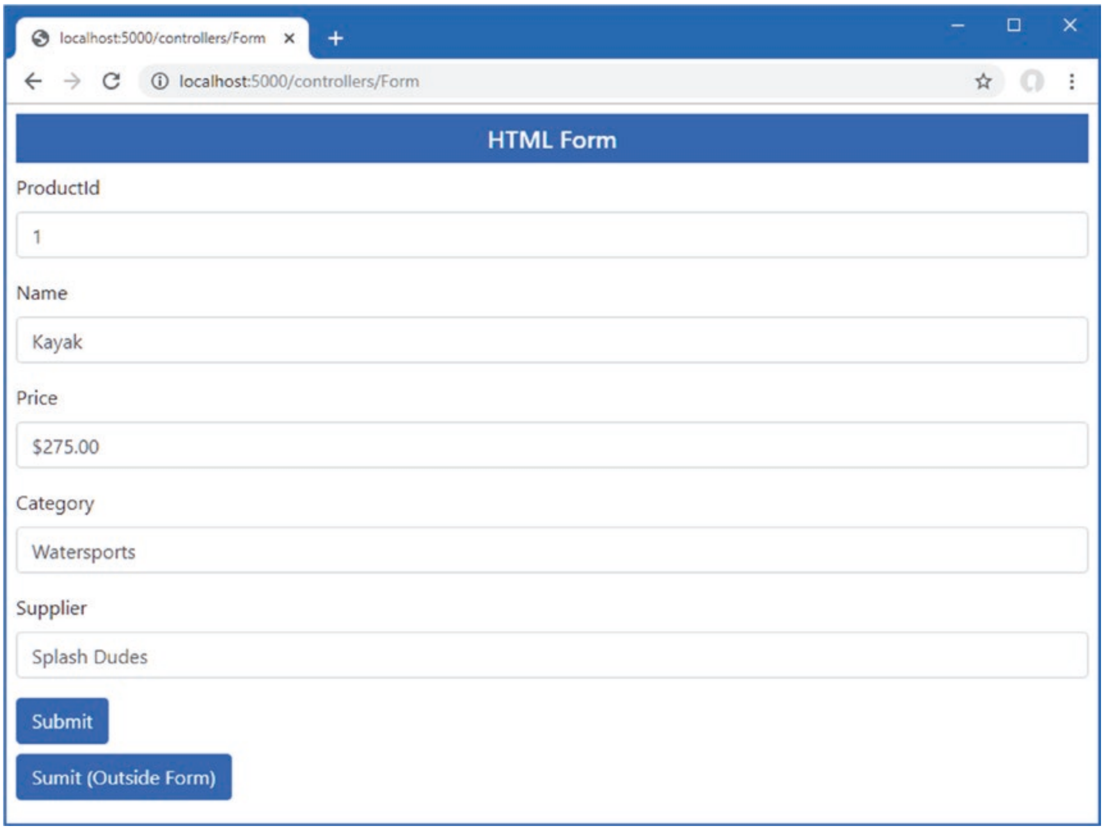

Figure 27-8. *Transforming label elements*

Working with Select and Option Elements

The `select` and `option` elements are used to provide the user with a fixed set of choices, rather than the open data entry that is possible with an `input` element. The `SelectTagHelper` is responsible for transforming `select` elements and supports the attributes described in Table 27-8.

Table 27-8. *The Built-in Tag Helper Attributes for select Elements*

Name	Description
`asp-for`	This attribute is used to specify the view or page model property that the `select` element represents.
`asp-items`	This attribute is used to specify a source of values for the option elements contained within the `select` element.

The `asp-for` attribute sets the value of the `for` and `id` attributes to reflect the model property that it receives. In Listing 27-24, I have replaced the `input` element for the category with a `select` element that presents the user with a fixed range of values.

Listing 27-24. Using a select Element in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<form asp-action="submitform" method="post" id="htmlform">
    <div class="form-group">
        <label asp-for="ProductId"></label>
```

```
        <input class="form-control" asp-for="ProductId" />
    </div>
    <div class="form-group">
        <label asp-for="Name"></label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label asp-for="Price"></label>
        <input class="form-control" asp-for="Price" />
    </div>
    <div class="form-group">
        <label asp-for="Category.Name">Category</label>
        <select class="form-control" asp-for="CategoryId">
            <option value="1">Watersports</option>
            <option value="2">Soccer</option>
            <option value="3">Chess</option>
        </select>
    </div>
    <div class="form-group">
        <label asp-for="Supplier.Name">Supplier</label>
        <input class="form-control" asp-for="Supplier.Name" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>

<button form="htmlform" asp-action="submitform" class="btn btn-primary mt-2">
    Sumit (Outside Form)
</button>
```

I have manually populated the `select` element with `option` elements that provide a range of categories for the user to choose from. If you use a browser to request `http://localhost:5000/controllers/form/index/5` and examine the HTML response, you will see that the tag helper has transformed the `select` element like this:

```
...
<div class="form-group">
    <label for="Category_Name">Category</label>
    <select class="form-control" data-val="true"
          data-val-required="The CategoryId field is required."
          id="CategoryId" name="CategoryId">
        <option value="1">Watersports</option>
        <option value="2" selected="selected">Soccer</option>
        <option value="3">Chess</option>
    </select>
</div>
...
```

Notice that selected attribute has been added to the `option` element that corresponds to the view model's `CategoryId` value, like this:

```
...
<option value="2" selected="selected">Soccer</option>
...
```

The task of selecting an `option` element is performed by the `OptionTagHelper` class, which receives instructions from the `SelectTagHelper` through the `TagHelperContext.Items` collection, described in Chapter 25. The result is that the `select` element displays the name of the category associated with the `Product` object's `CategoryId` value.

Populating a select Element

Explicitly defining the option elements for a select element is a useful approach for choices that always have the same possible values but doesn't help when you need to provide options that are taken from the data model or where you need the same set of options in multiple views and don't want to manually maintain duplicated content.

The asp-items attribute is used to provide the tag helper with a list sequence of SelectListItem objects for which option elements will be generated. Listing 27-25 modifies the Index action of the Form controller to provide the view with a sequence of SelectListItem objects through the view bag.

Listing 27-25. Providing a Data Sequence in the FormController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using System.Threading.Tasks;
using WebApp.Models;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Mvc.Rendering;

namespace WebApp.Controllers {

    public class FormController : Controller {
        private DataContext context;

        public FormController(DataContext dbContext) {
            context = dbContext;
        }

        public async Task<IActionResult> Index(long id = 1) {
            ViewBag.Categories
                = new SelectList(context.Categories, "CategoryId", "Name");
            return View("Form", await context.Products.Include(p => p.Category)
                .Include(p => p.Supplier).FirstAsync(p => p.ProductId == id));
        }

        [HttpPost]
        public IActionResult SubmitForm() {
            foreach (string key in Request.Form.Keys
                    .Where(k => !k.StartsWith("_"))) {
                TempData[key] = string.Join(", ", Request.Form[key]);
            }
            return RedirectToAction(nameof(Results));
        }

        public IActionResult Results() {
            return View(TempData);
        }
    }
}
```

SelectListItem objects can be created directly, but ASP.NET Core provides the SelectList class to adapt existing data sequences. In this case, I pass the sequence of Category objects obtained from the database to the SelectList constructor, along with the names of the properties that should be used as the values and labels for option elements. In Listing 27-26, I have updated the Form view to use the SelectList.

Listing 27-26. Using a SelectList in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<form asp-action="submitform" method="post" id="htmlform">
    <div class="form-group">
        <label asp-for="ProductId"></label>
        <input class="form-control" asp-for="ProductId" />
    </div>
    <div class="form-group">
        <label asp-for="Name"></label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label asp-for="Price"></label>
        <input class="form-control" asp-for="Price" />
    </div>
    <div class="form-group">
        <label asp-for="Category.Name">Category</label>
        <select class="form-control" asp-for="CategoryId"
            asp-items="@ViewBag.Categories">
        </select>
    </div>
    <div class="form-group">
        <label asp-for="Supplier.Name">Supplier</label>
        <input class="form-control" asp-for="Supplier.Name" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>

<button form="htmlform" asp-action="submitform" class="btn btn-primary mt-2">
    Sumit (Outside Form)
</button>
```

Restart ASP.NET Core so the changes to the controller take effect and use a browser to request `http://localhost:5000/controllers/form/index/5`. There is no visual change to the content presented to the user, but the `option` elements used to populate the `select` element have been generated from the database, like this:

```
...
<div class="form-group">
    <label for="Category_Name">Category</label>
    <select class="form-control" data-val="true"
           data-val-required="The CategoryId field is required."
           id="CategoryId" name="CategoryId">
        <option value="1">Watersports</option>
        <option selected="selected" value="2">Soccer</option>
        <option value="3">Chess</option>
    </select>
</div>
...
```

This approach means that the options presented to the user will automatically reflect new categories added to the database.

Working with Text Areas

The textarea element is used to solicit a larger amount of text from the user and is typically used for unstructured data, such as notes or observations. The TextAreaTagHelper is responsible for transforming textarea elements and supports the single attribute described in Table 27-9.

Table 27-9. *The Built-in Tag Helper Attributes for TextArea Elements*

Name	Description
asp-for	This attribute is used to specify the view model property that the textarea element represents.

The TextAreaTagHelper is relatively simple, and the value provided for the asp-for attribute is used to set the id and name attributes on the textarea element. The value of the property selected by the asp-for attribute is used as the content for the textarea element. Listing 27-27 replaces the input element for the Supplier.Name property with a text area to which the asp-for attribute has been applied.

Listing 27-27. Using a Text Area in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<form asp-action="submitform" method="post" id="htmlform">
    <div class="form-group">
        <label asp-for="ProductId"></label>
        <input class="form-control" asp-for="ProductId" />
    </div>
    <div class="form-group">
        <label asp-for="Name"></label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label asp-for="Price"></label>
        <input class="form-control" asp-for="Price" />
    </div>
    <div class="form-group">
        <label asp-for="Category.Name">Category</label>
        <select class="form-control" asp-for="CategoryId"
             asp-items="@ViewBag.Categories">
        </select>
    </div>
    <div class="form-group">
        <label asp-for="Supplier.Name">Supplier</label>
        <textarea class="form-control" asp-for="Supplier.Name"></textarea>
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>

<button form="htmlform" asp-action="submitform" class="btn btn-primary mt-2">
    Sumit (Outside Form)
</button>
```

Use a browser to request http://localhost:5000/controllers/form and examine the HTML received by the browser to see the transformation of the textarea element.

```
...
<div class="form-group">
    <label for="Supplier_Name">Supplier</label>
    <textarea class="form-control" id="Supplier_Name" name="Supplier.Name">
        Soccer Town
    </textarea>
</div>
...
```

The `TextAreaTagHelper` is relatively simple, but it provides consistency with the rest of the form element tag helpers that I have described in this chapter.

Using the Anti-forgery Feature

When I defined the controller action method and page handler methods that process form data, I filtered out form data whose name begins with an underscore, like this:

```
...
[HttpPost]
public IActionResult SubmitForm() {
    foreach (string key in Request.Form.Keys
            .Where(k => !k.StartsWith("_"))) {
        TempData[key] = string.Join(", ", Request.Form[key]);
    }
    return RedirectToAction(nameof(Results));
}
...
```

I applied this filter to hide a feature to focus on the values provided by the HTML elements in the form. Listing 27-28 removes the filter from the action method so that all the data received from the HTML form is stored in temp data.

Listing 27-28. Removing a Filter in the FormController.cs File in the Controllers Folder

```
...
[HttpPost]
public IActionResult SubmitForm() {
    foreach (string key in Request.Form.Keys) {
        TempData[key] = string.Join(", ", Request.Form[key]);
    }
    return RedirectToAction(nameof(Results));
}
...
```

Restart ASP.NET Core and use a browser to request `http://localhost:5000/controllers`. Click the Submit button to send the form to the application, and you will see a new item in the results, as shown in Figure 27-9.

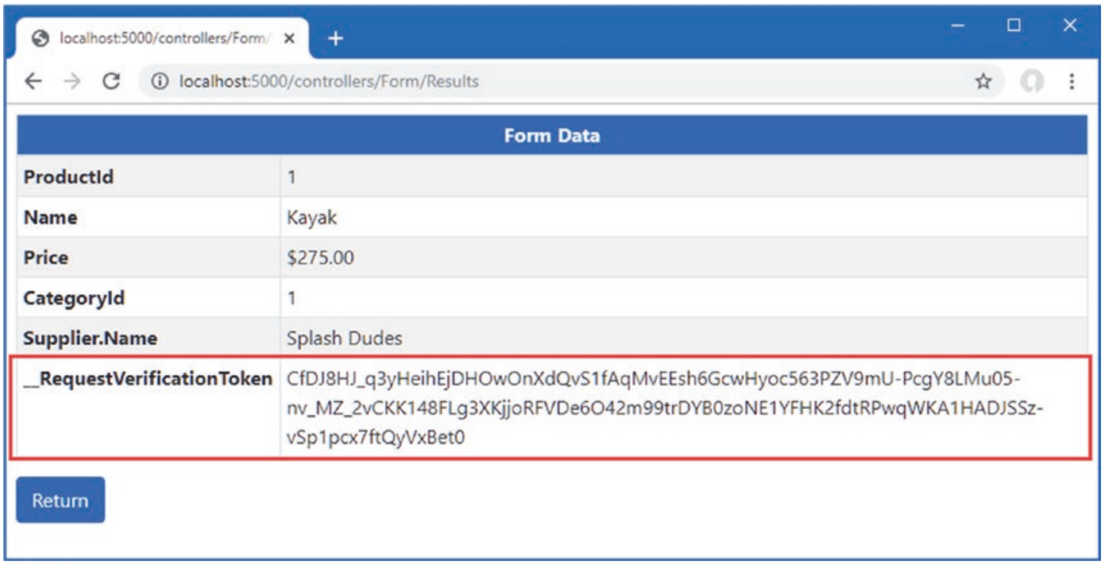

Figure 27-9. *Showing all form data*

The `_RequestVerificationToken` form value displayed in the results is a security feature that is applied by the `FormTagHelper` to guard against cross-site request forgery. Cross-site request forgery (CSRF) exploits web applications by taking advantage of the way that user requests are typically authenticated. Most web applications—including those created using ASP.NET Core—use cookies to identify which requests are related to a specific session, with which a user identity is usually associated.

CSRF—also known as *XSRF*—relies on the user visiting a malicious website after using your web application and without explicitly ending their session. The application still regards the user's session as being active, and the cookie that the browser has stored has not yet expired. The malicious site contains JavaScript code that sends a form request to your application to perform an operation without the user's consent—the exact nature of the operation will depend on the application being attacked. Since the JavaScript code is executed by the user's browser, the request to the application includes the session cookie, and the application performs the operation without the user's knowledge or consent.

■ **Tip** CSRF is described in detail at `http://en.wikipedia.org/wiki/Cross-site_request_forgery`.

If a `form` element doesn't contain an `action` attribute—because it is being generated from the routing system with the `asp-controller`, `asp-action`, and `asp-page` attributes—then the `FormTagHelper` class automatically enables an anti-CSRF feature, whereby a security token is added to the response as a cookie. A hidden `input` element containing the same security token is added to the HTML form, and it is this token that is shown in Figure 27-9.

Enabling the Anti-forgery Feature in a Controller

By default, controllers accept POST requests even when they don't contain the required security tokens. To enable the anti-forgery feature, an attribute is applied to the controller class, as shown in Listing 27-29.

Listing 27-29. Enabling the Anti-forgery Feature in the FormController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using System.Threading.Tasks;
using WebApp.Models;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Mvc.Rendering;
```

```
namespace WebApp.Controllers {

    [AutoValidateAntiforgeryToken]
    public class FormController : Controller {
        private DataContext context;

        public FormController(DataContext dbContext) {
            context = dbContext;
        }

        public async Task<IActionResult> Index(long id = 1) {
            ViewBag.Categories
                = new SelectList(context.Categories, "CategoryId", "Name");
            return View("Form", await context.Products.Include(p => p.Category)
                .Include(p => p.Supplier).FirstAsync(p => p.ProductId == id));
        }

        [HttpPost]
        public IActionResult SubmitForm() {
            foreach (string key in Request.Form.Keys) {
                TempData[key] = string.Join(", ", Request.Form[key]);
            }
            return RedirectToAction(nameof(Results));
        }

        public IActionResult Results() {
            return View(TempData);
        }
    }
}
```

Not all requests require an anti-forgery token, and the `AutoValidateAntiforgeryToken` ensures that checks are performed for all HTTP methods except GET, HEAD, OPTIONS, and TRACE.

■ **Tip** Two other attributes can be used to control token validation. The `IgnoreValidationToken` attribute suppresses validation for an action method or controller. The `ValidateAntiForgeryToken` attribute does the opposite and enforces validation, even for requests that would not normally require validation, such as HTTP GET requests. I recommend using the `AutoValidateAntiforgeryToken` attribute, as shown in the listing.

Testing the anti-CSRF feature is a little tricky. I do it by requesting the URL that contains the form (`http://localhost:5000/controllers/forms` for this example) and then using the browser's F12 developer tools to locate and remove the hidden `input` element from the form (or change the element's value). When I populate and submit the form, it is missing one part of the required data, and the request will fail.

Enabling the Anti-forgery Feature in a Razor Page

The anti-forgery feature is enabled by default in Razor Pages, which is why I applied the `IgnoreAntiforgeryToken` attribute to the page handler method in Listing 27-29 when I created the `FormHandler` page. Listing 27-30 removes the attribute to enable the validation feature.

Listing 27-30. Enabling Request Validation in the FormHandler.cshtml File in the Pages Folder

```
@page "/pages/form/{id:long?}"
@model FormHandlerModel
@using Microsoft.AspNetCore.Mvc.RazorPages
@using Microsoft.EntityFrameworkCore

<div class="m-2">
    <h5 class="bg-primary text-white text-center p-2">HTML Form</h5>
    <form asp-page="FormHandler" method="post" id="htmlform">
        <div class="form-group">
            <label>Name</label>
            <input class="form-control" asp-for="Product.Name" />
        </div>

        <div class="form-group">
            <label>Price</label>
            <input class="form-control" asp-for="Product.Price" />
        </div>
        <div class="form-group">
            <label>Category</label>
            <input class="form-control" asp-for="Product.Category.Name" />
        </div>
        <div class="form-group">
            <label>Supplier</label>
            <input class="form-control" asp-for="Product.Supplier.Name" />
        </div>
        <button type="submit" class="btn btn-primary">Submit</button>
    </form>
    <button form="htmlform" asp-page="FormHandler" class="btn btn-primary mt-2">
        Sumit (Outside Form)
    </button>
</div>

@functions {

    //[IgnoreAntiforgeryToken]
    public class FormHandlerModel : PageModel {
        private DataContext context;

        public FormHandlerModel(DataContext dbContext) {
            context = dbContext;
        }

        public Product Product { get; set; }

        public async Task OnGetAsync(long id = 1) {
            Product = await context.Products.Include(p => p.Category)
                .Include(p => p.Supplier).FirstAsync(p => p.ProductId == id);
        }

        public IActionResult OnPost() {
            foreach (string key in Request.Form.Keys
                    .Where(k => !k.StartsWith("_"))) {
                TempData[key] = string.Join(", ", Request.Form[key]);
            }
```

```
            return RedirectToPage("FormResults");
        }
    }
}
```

Testing the validation feature is done in the same way as for controllers and requires altering the HTML document using the browser's developer tools before submitting the form to the application.

Using Anti-forgery Tokens with JavaScript Clients

By default, the anti-forgery feature relies on the ASP.NET Core application being able to include an element in an HTML form that the browser sends back when the form is submitted. This doesn't work for JavaScript clients because the ASP.NET Core application provides data and not HTML, so there is no way to insert the hidden element and receive it in a future request.

For web services, the anti-forgery token can be sent as a JavaScript-readable cookie, which the JavaScript client code reads and includes as a header in its POST requests. Some JavaScript frameworks, such an Angular, will automatically detect the cookie and include a header in requests. For other frameworks and custom JavaScript code, additional work is required.

Listing 27-31 shows the changes required to the ASP.NET Core application to configure the anti-forgery feature for use with JavaScript clients.

Listing 27-31. Configuring the Anti-forgery Token in the Startup.cs File in the WebApp Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using WebApp.Models;
using Microsoft.AspNetCore.Antiforgery;
using Microsoft.AspNetCore.Http;

namespace WebApp {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        public IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<DataContext>(opts => {
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:ProductConnection"]);
                opts.EnableSensitiveDataLogging(true);
            });
            services.AddControllersWithViews().AddRazorRuntimeCompilation();
            services.AddRazorPages().AddRazorRuntimeCompilation();
            services.AddSingleton<CitiesData>();

            services.Configure<AntiforgeryOptions>(opts => {
                opts.HeaderName = "X-XSRF-TOKEN";
            });
        }

        public void Configure(IApplicationBuilder app, DataContext context,
                IAntiforgery antiforgery) {
```

```
            app.UseRequestLocalization();

            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseRouting();

            app.Use(async (context, next) => {
                if (!context.Request.Path.StartsWithSegments("/api")) {
                    context.Response.Cookies.Append("XSRF-TOKEN",
                       antiforgery.GetAndStoreTokens(context).RequestToken,
                       new CookieOptions { HttpOnly = false });
                }
                await next();
            });

            app.UseEndpoints(endpoints => {
                endpoints.MapControllers();
                endpoints.MapControllerRoute("forms",
                    "controllers/{controller=Home}/{action=Index}/{id?}");
                endpoints.MapDefaultControllerRoute();
                endpoints.MapRazorPages();
            });
            SeedData.SeedDatabase(context);
        }
    }
}
```

The options pattern is used to configure the anti-forgery feature, through the `AntiforgeryOptions` class. The `HeaderName` property is used to specify the name of a header through which anti-forgery tokens will be accepted, which is `X-XSRF-TOKEN` in this case.

A custom middleware component is required to set the cookie, which is named `XSRF-TOKEN` in this example. The value of the cookie is obtained through the `IAntiForgery` service and must be configured with the `HttpOnly` option set to `false` so that the browser will allow JavaScript code to read the cookie.

■ **Tip** I have followed the names that are supported by Angular in this example. Other frameworks follow their own conventions but can usually be configured to use any set of cookie and header names.

To create a simple JavaScript client that uses the cookie and header, add a Razor Page named `JavaScriptForm.cshtml` to the `Pages` folder with the content shown in Listing 27-32.

Listing 27-32. The Contents of the JavaScriptForm.cshtml File in the Pages Folder

```
@page "/pages/jsform"

<script type="text/javascript">
    async function sendRequest() {
        const token = document.cookie
            .replace(/(?:(?:^|.*;\s*)XSRF-TOKEN\s*\=\s*([^;]*).*$)|^.*$/, "$1");

        let form = new FormData();
        form.append("name", "Paddle");
        form.append("price", 100);
        form.append("categoryId", 1);
        form.append("supplierId", 1);
```

```
        let response = await fetch("@Url.Page("FormHandler")", {
            method: "POST",
            headers: { "X-XSRF-TOKEN": token },
            body: form
        });
        document.getElementById("content").innerHTML = await response.text();
    }

    document.addEventListener("DOMContentLoaded",
        () => document.getElementById("submit").onclick = sendRequest);
</script>

<button class="btn btn-primary m-2" id="submit">Submit JavaScript Form</button>
<div id="content"></div>
```

The JavaScript code in this Razor Page responds to a button click by sending an HTTP POST request to the `FormHandler` Razor Page. The value of the `XSRF-TOKEN` cookie is read and included in the `X-XSRF-TOKEN` request header. The response from the `FormHandler` page is a redirection to the `Results` page, which the browser will follow automatically. The response from the `Results` page is read by the JavaScript code and inserted into an element so it can be displayed to the user. To test the JavaScript code, use a browser to request `http://localhost:5000/pages/jsform` and click the button. The JavaScript code will submit the form and display the response, as shown in Figure 27-10.

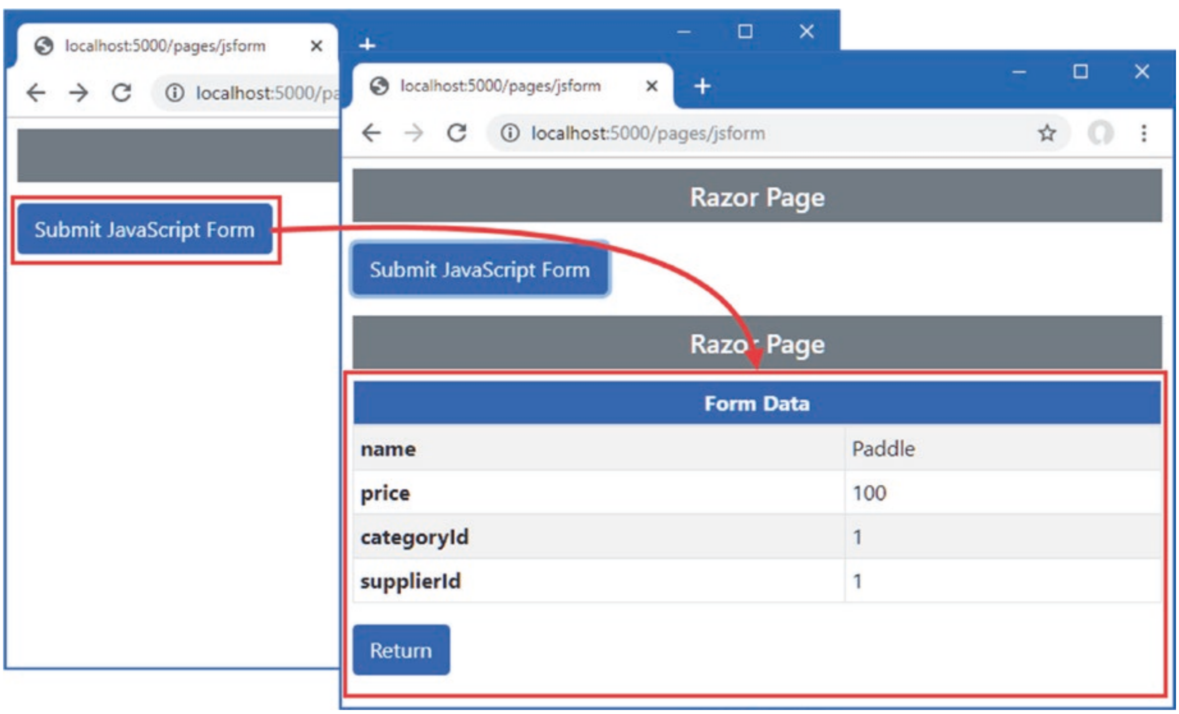

Figure 27-10. Using a security token in JavaScript code

Summary

In this chapter, I explained the features that ASP.NET Core provides for creating HTML forms. I showed you how tag helpers are used to select the form target and associate `input`, `textarea`, and `select` elements with view model or page model properties. In the next chapter, I describe the model binding feature, which extracts data from requests so that it can easily be consumed in action and handler methods.

CHAPTER 28

Using Model Binding

Model binding is the process of creating .NET objects using the values from the HTTP request to provide easy access to the data required by action methods and Razor Pages. In this chapter, I describe the way the model binding system works; show how it binds simple types, complex types, and collections; and demonstrate how you can take control of the process to specify which part of the request provides the data values your application requires. Table 28-1 puts model binding in context.

Table 28-1. *Putting Model Binding in Context*

Question	Answer
What is it?	Model binding is the process of creating the objects that action methods and page handlers require using data values obtained from the HTTP request.
Why is it useful?	Model binding lets controllers or page handlers declare method parameters or properties using C# types and automatically receive data from the request without having to inspect, parse, and process the data directly.
How is it used?	In its simplest form, methods declare parameters or classes define properties whose names are used to retrieve data values from the HTTP request. The part of the request used to obtain the data can be configured by applying attributes to the method parameters or properties.
Are there any pitfalls or limitations?	The main pitfall is getting data from the wrong part of the request. I explain the way that requests are searched for data in the "Understanding Model Binding" section, and the search locations can be specified explicitly using the attributes that I describe in the "Specifying a Model Binding Source" section.
Are there any alternatives?	Data can be obtained without model binding using context objects. However, the result is more complicated code that is hard to read and maintain.

Table 28-2 summarizes the chapter.

Table 28-2. *Chapter Summary*

Problem	Solution	Listing
Binding simple types	Define method parameters with primitive types	5–9
Binding complex types	Define method parameters with class types	10
Binding to a property	Use the `BindProperty` attribute	11, 12
Binding nested types	Ensure the form value types follow the dotted notation	13–17
Selecting properties for binding	Use the `Bind` and `BindNever` attributes	18–19
Binding collections	Follow the sequence binding conventions	20–25
Specifying the source for binding	Use one of the source attributes	26–31
Manually performing binding	Use the `TryUpdateModel` method	32

A. Freeman, *Pro ASP.NET Core 3*, https://doi.org/10.1007/978-1-4842-5440-0_28

Preparing for This Chapter

This chapter uses the WebApp project from Chapter 27. To prepare for this chapter, replace the contents of the Form.cshtml file in the Views/Form folder with the content shown in Listing 28-1.

■ **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-3. See Chapter 1 for how to get help if you have problems running the examples.

Listing 28-1. The Contents of the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<form asp-action="submitform" method="post" id="htmlform">
    <div class="form-group">
        <label asp-for="Name"></label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label asp-for="Price"></label>
        <input class="form-control" asp-for="Price" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

Next, comment out the DisplayFormat attribute that has been applied to the Product model class, as shown in Listing 28-2.

Listing 28-2. Removing an Attribute in the Product.cs File in the Models Folder

```
using System.ComponentModel.DataAnnotations.Schema;
using System.ComponentModel.DataAnnotations;

namespace WebApp.Models {
    public class Product {

        public long ProductId { get; set; }

        public string Name { get; set; }

        [Column(TypeName = "decimal(8, 2)")]
        //[DisplayFormat(DataFormatString = "{0:c2}", ApplyFormatInEditMode = true)]
        public decimal Price { get; set; }

        public long CategoryId { get; set; }
        public Category Category { get; set; }

        public long SupplierId { get; set; }
        public Supplier Supplier { get; set; }
    }
}
```

Dropping the Database

Open a new PowerShell command prompt, navigate to the folder that contains the `WebApp.csproj` file, and run the command shown in Listing 28-3 to drop the database.

Listing 28-3. Dropping the Database

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

Running the Example Application

Select Start Without Debugging or Run Without Debugging from the Debug menu or use the PowerShell command prompt to run the command shown in Listing 28-4.

Listing 28-4. Running the Example Application

```
dotnet run
```

Use a browser to request `http://localhost:5000/controllers/form`, which will display an HTML form. Click the Submit button, and the form data will be displayed, as shown in Figure 28-1.

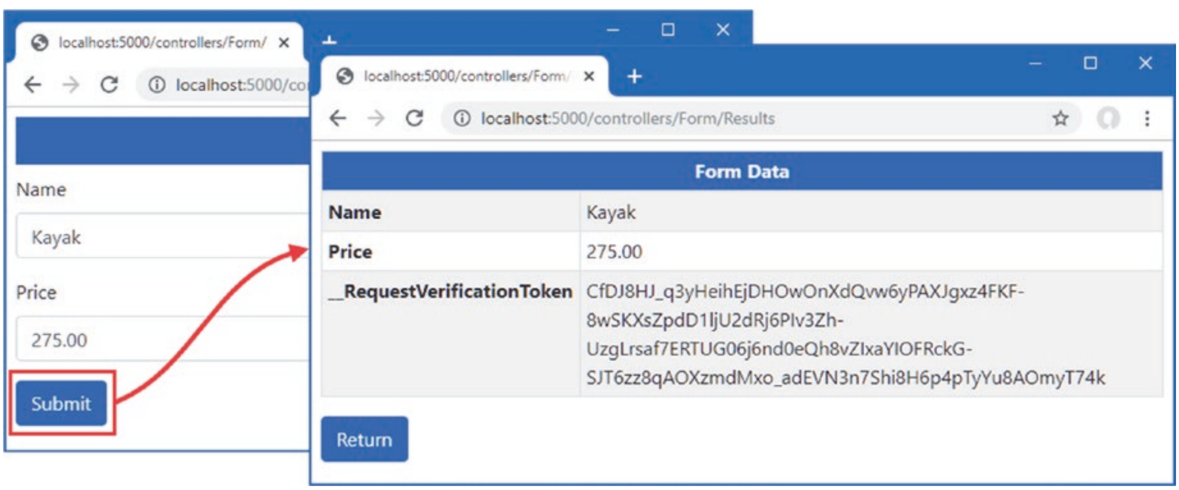

Figure 28-1. *Running the example application*

Understanding Model Binding

Model binding is an elegant bridge between the HTTP request and action or page handler methods. Most ASP.NET Core applications rely on model binding to some extent, including the example application for this chapter.

You can see model binding at work by using the browser to request `http://localhost:5000/controllers/form/index/5`. This URL contains the value of the `ProductId` property of the `Product` object that I want to view, like this:

```
http://localhost:5000/controllers/form/index/5
```

This part of the URL corresponds to the id segment variable defined by the controller routing pattern and matches the name of the parameter defined by the Form controller's Index action:

```
...
public async Task<IActionResult> Index(long id = 1) {
...
```

A value for the id parameter is required before the MVC Framework can invoke the action method, and finding a suitable value is the responsibility of the *model binding* system. The model binding system relies on *model binders*, which are components responsible for providing data values from one part of the request or application. The default model binders look for data values in these four places:

- Form data
- The request body (only for controllers decorated with ApiController)
- Routing segment variables
- Query strings

Each source of data is inspected in order until a value for the argument is found. There is no form data in the example application, so no value will be found there, and the Form controller isn't decorated with the ApiController attribute, so the request body won't be checked. The next step is to check the routing data, which contains a segment variable named id. This allows the model binding system to provide a value that allows the Index action method to be invoked. The search stops after a suitable data value has been found, which means that the query string isn't searched for a data value.

■ **Tip** In the "Specifying a Model Binding Source" section, I explain how you can specify the source of model binding data using attributes. This allows you to specify that a data value is obtained from, for example, the query string, even if there is also suitable data in the routing data.

Knowing the order in which data values are sought is important because a request can contain multiple values, like this URL:

```
http://localhost:5000/controllers/Form/Index/5?id=1
```

The routing system will process the request and match the id segment in the URL template to the value 3, and the query string contains an id value of 1. Since the routing data is searched for data before the query string, the Index action method will receive the value 3, and the query string value will be ignored.

On the other hand, if you request a URL that doesn't have an id segment, then the query string will be examined, which means that a URL like this one will also allow the model binding system to provide a value for the id argument so that it can invoke the Index method.

```
http://localhost:5000/controllers/Form/Index?id=4
```

You can see the effect of both these URLs in Figure 28-2.

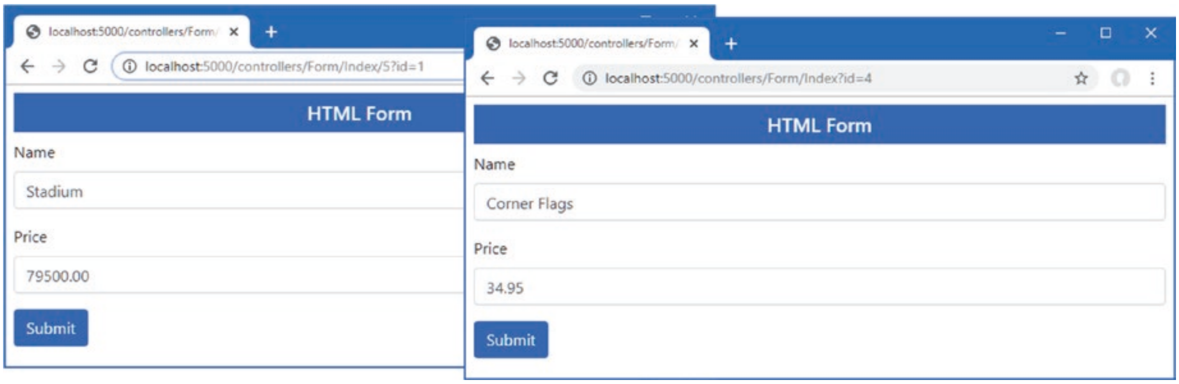

Figure 28-2. *The effect of model binding data source order*

Binding Simple Data Types

Request data values must be converted into C# values so they can be used to invoke action or page handler methods. *Simple types* are values that originate from one item of data in the request that can be parsed from a string. This includes numeric values, `bool` values, dates, and, of course, `string` values.

Data binding for simple types makes it easy to extract single data items from the request without having to work through the context data to find out where it is defined. Listing 28-5 adds parameters to the `SubmitForm` action method defined by the `Form` controller method so that the model binder will be used to provide `name` and `price` values.

Listing 28-5. Adding Method Parameters in the FormController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using System.Threading.Tasks;
using WebApp.Models;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Mvc.Rendering;

namespace WebApp.Controllers {

    [AutoValidateAntiforgeryToken]
    public class FormController : Controller {
        private DataContext context;

        public FormController(DataContext dbContext) {
            context = dbContext;
        }

        public async Task<IActionResult> Index(long id = 1) {
            ViewBag.Categories
                = new SelectList(context.Categories, "CategoryId", "Name");
            return View("Form", await context.Products.Include(p => p.Category)
                .Include(p => p.Supplier).FirstAsync(p => p.ProductId == id));
        }

        [HttpPost]
        public IActionResult SubmitForm(string name, decimal price) {
            TempData["name param"] = name;
            TempData["price param"] = price.ToString();
            return RedirectToAction(nameof(Results));
        }
```

```
        public IActionResult Results() {
            return View(TempData);
        }
    }
}
```

The model binding system will be used to obtain name and price values when ASP.NET Core receives a request that will be processed by the SubmitForm action method. The use of parameters simplifies the action method and takes care of converting the request data into C# data types so that the price value will be converted to the C# decimal type before the action method is invoked. (I had to convert the decimal back to a string to store it as temp data in this example. I demonstrate more useful ways of dealing with form data in Chapter 31.) Restart ASP.NET Core so the change to the controller takes effect and request `http://localhost:5000/controllers/Form`. Click the Submit button, and you will see the values that were extracted from the request by the model binding feature, as shown in Figure 28-3.

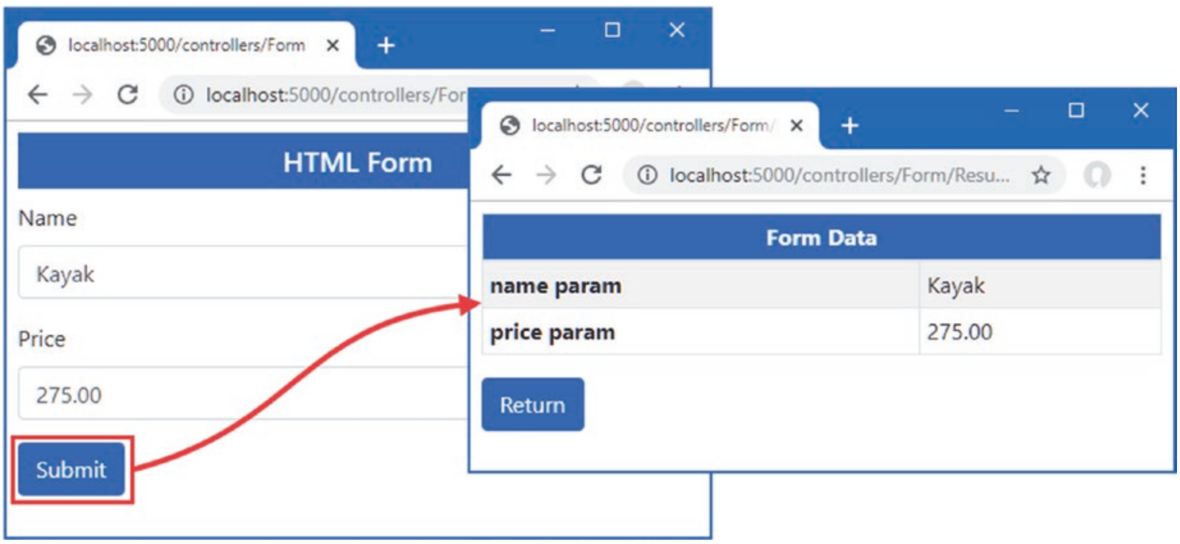

Figure 28-3. *Model binding for simple types*

Binding Simple Data Types in Razor Pages

Razor Pages can use model binding, but care must be taken to ensure that the value of the form element's name attribute matches the name of the handler method parameter, which may not be the case if the asp-for attribute has been used to select a nested property. To ensure the names match, the name attribute can be defined explicitly, as shown in Listing 28-6, which also simplifies the HTML form so that it matches the controller example.

Listing 28-6. Using Model Binding in the FormHandler.cshtml File in the Pages Folder

```
@page "/pages/form/{id:long?}"
@model FormHandlerModel
@using Microsoft.AspNetCore.Mvc.RazorPages
@using Microsoft.EntityFrameworkCore

<div class="m-2">
    <h5 class="bg-primary text-white text-center p-2">HTML Form</h5>
    <form asp-page="FormHandler" method="post" id="htmlform">
        <div class="form-group">
            <label>Name</label>
            <input class="form-control" asp-for="Product.Name" name="name"/>
        </div>
        <div class="form-group">
            <label>Price</label>
```

```
            <input class="form-control" asp-for="Product.Price" name="price" />
        </div>
        <button type="submit" class="btn btn-primary">Submit</button>
    </form>
</div>

@functions {

    public class FormHandlerModel : PageModel {
        private DataContext context;

        public FormHandlerModel(DataContext dbContext) {
            context = dbContext;
        }

        public Product Product { get; set; }

        public async Task OnGetAsync(long id = 1) {
            Product = await context.Products.Include(p => p.Category)
                .Include(p => p.Supplier).FirstAsync(p => p.ProductId == id);
        }

        public IActionResult OnPost(string name, decimal price) {
            TempData["name param"] = name;
            TempData["price param"] = price.ToString();
            return RedirectToPage("FormResults");
        }
    }
}
```

The tag helper would have set the name attributes of the input elements to `Product.Name` and `Product.Price`, which prevents the model binder from matching the values. Explicitly setting the name attribute overrides the tag helper and ensures the model binding process works correctly. Use a browser to request `http://localhost:5000/pages/form` and click the Submit button, and you will see the values found by the model binder, as shown in Figure 28-4.

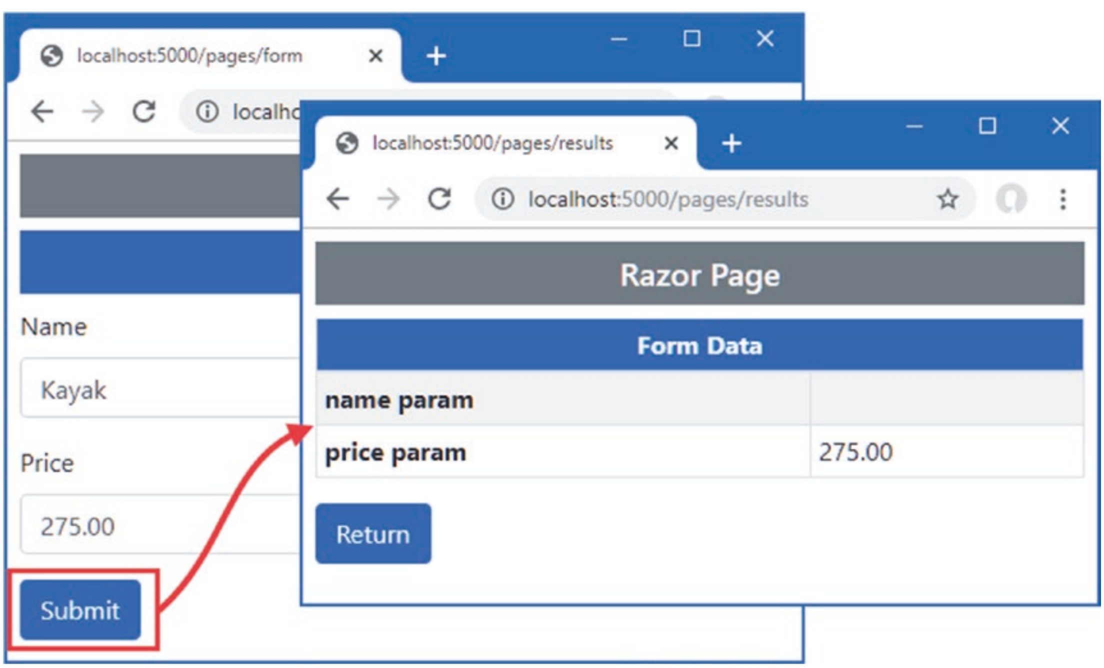

Figure 28-4. *Model binding in a Razor Page*

Understanding Default Binding Values

Model binding is a best-effort feature, which means the model binder will try to get values for method parameters but will still invoke the method if data values cannot be located. You can see how this works by removing the default value for the `id` parameter in the `Form` controller's `Index` action method, as shown in Listing 28-7.

Listing 28-7. Removing a Parameter in the FormController.cs File in the Controllers Folder

```
...
public async Task<IActionResult> Index(long id) {
    ViewBag.Categories
        = new SelectList(context.Categories, "CategoryId", "Name");
    return View("Form", await context.Products.Include(p => p.Category)
        .Include(p => p.Supplier).FirstAsync(p => p.ProductId == id));
}
...
```

Restart ASP.NET Core and request `http://localhost:5000/controllers/Form`. The URL doesn't contain a value that the model binder can use for the `id` parameter, and there is no query string or form data, but the method is still invoked, producing the error shown in Figure 28-5.

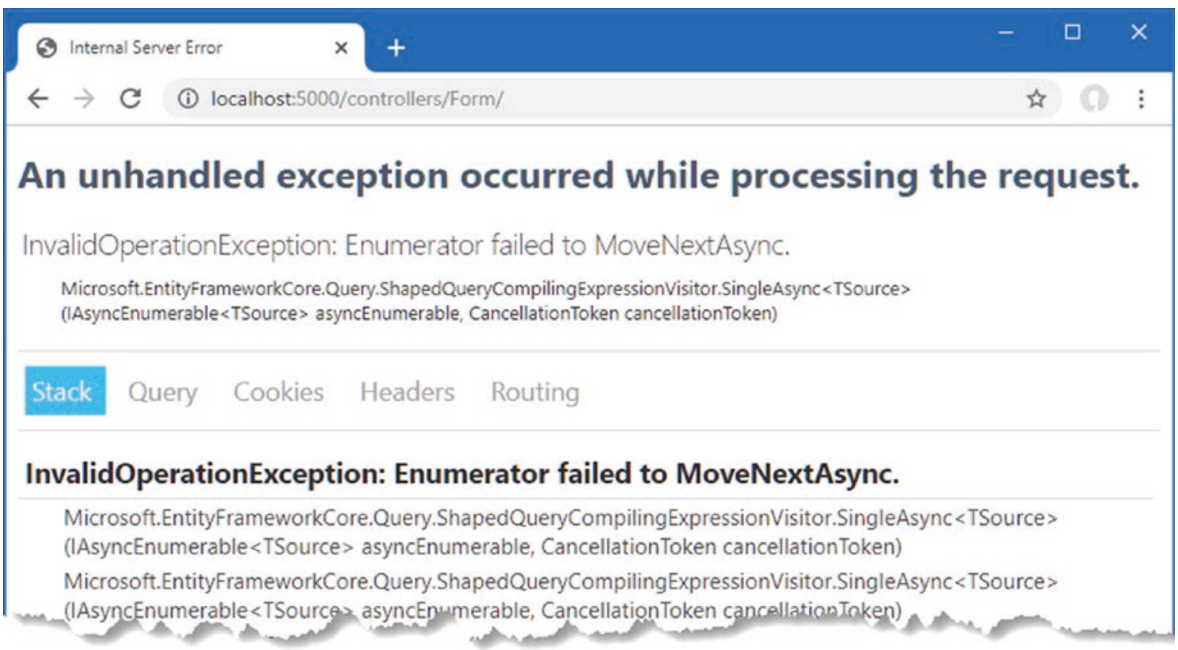

Figure 28-5. *An error caused by a missing data value*

This exception isn't reported by the model binding system. Instead, it occurred when the Entity Framework Core query was executed. The MVC Framework must provide *some* value for the `id` argument to invoke the `Index` action method, so it uses a default value and hopes for the best. For `long` arguments, the default value is `0`, and this is what leads to the exception. The `Index` action method uses the `id` value as the key to query the database for a `Product` object, like this:

```
...
public async Task<IActionResult> Index(long id) {
    ViewBag.Categories = new SelectList(context.Categories, "CategoryId", "Name");
    return View("Form", await context.Products.Include(p => p.Category)
        .Include(p => p.Supplier).FirstAsync(p => p.ProductId == id));
}
...
```

When there is no value available for model binding, the action method tries to query the database with an `id` of zero. There is no such object, which causes the error shown in the figure when Entity Framework Core tries to process the result.

Applications must be written to cope with default argument values, which can be done in several ways. You can add fallback values to the routing URL patterns used by controllers (as shown in Chapter 21) or pages (as shown in Chapter 23). You can assign default values when defining the parameter in the action or page handler method, which is the approach that I have taken so far in this part of the book. Or you can simply write methods that accommodate the default values without causing an error, as shown in Listing 28-8.

Listing 28-8. Avoiding a Query Error in the FormController.cs File in the Controllers Folder

```
...
public async Task<IActionResult> Index(long id) {
    ViewBag.Categories = new SelectList(context.Categories, "CategoryId", "Name");
    return View("Form", await context.Products.Include(p => p.Category)
        .Include(p => p.Supplier).FirstOrDefaultAsync(p => p.ProductId == id));
}
...
```

The Entity Framework Core `FirstOrDefaultAsync` method will return `null` if there is no matching object in the database and won't attempt to load related data. The tag helpers cope with `null` values and display empty fields, which you can see by restarting ASP.NET Core and requesting `http://localhost:5000/controllers/Form`, which produces the result shown in Figure 28-6.

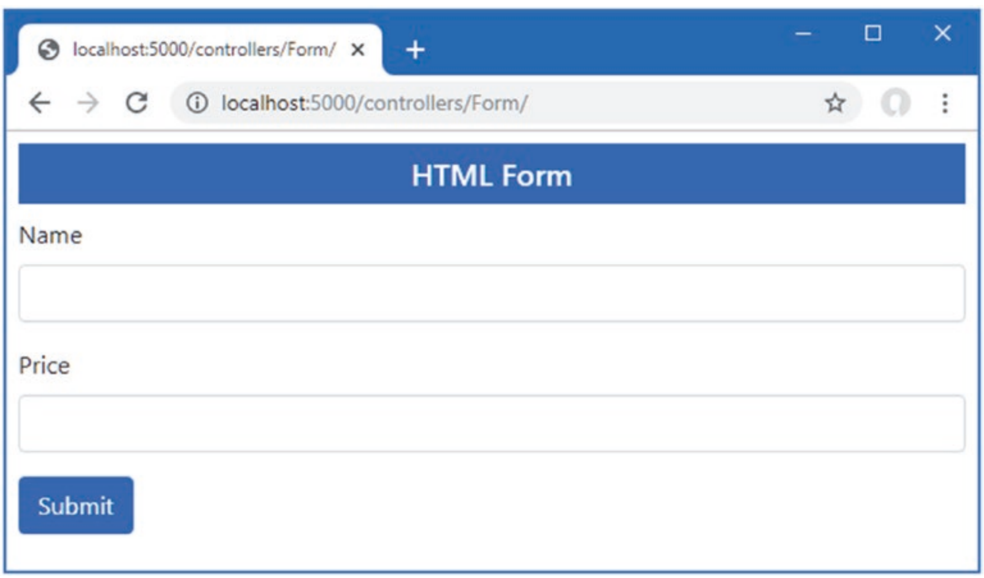

Figure 28-6. *Avoiding an error*

Some applications need to differentiate between a missing value and any value provided by the user. In these situations, a nullable parameter type can be used, as shown in Listing 28-9.

Listing 28-9. Using a Nullable Parameter in the FormController.cs File in the Controllers Folder

```
...
public async Task<IActionResult> Index(long? id) {
    ViewBag.Categories = new SelectList(context.Categories, "CategoryId", "Name");
    return View("Form", await context.Products.Include(p => p.Category)
        .Include(p => p.Supplier)
        .FirstOrDefaultAsync(p => id == null || p.ProductId == id));
}
...
```

The `id` parameter will be `null` only if the request doesn't contain a suitable value, which allows the expression passed to the `FirstOrDefaultAsync` method to default to the first object in the database when there is no value and to query for any other value. To see the effect, restart ASP.NET Core and request `http://localhost:5000/controllers/Form` and `http://localhost:5000/controllers/Form/index/0`. The first URL contains no `id` value, so the first object in the database is selected. The second URL provides an `id` value of zero, which doesn't correspond to any object in the database. Figure 28-7 shows both results.

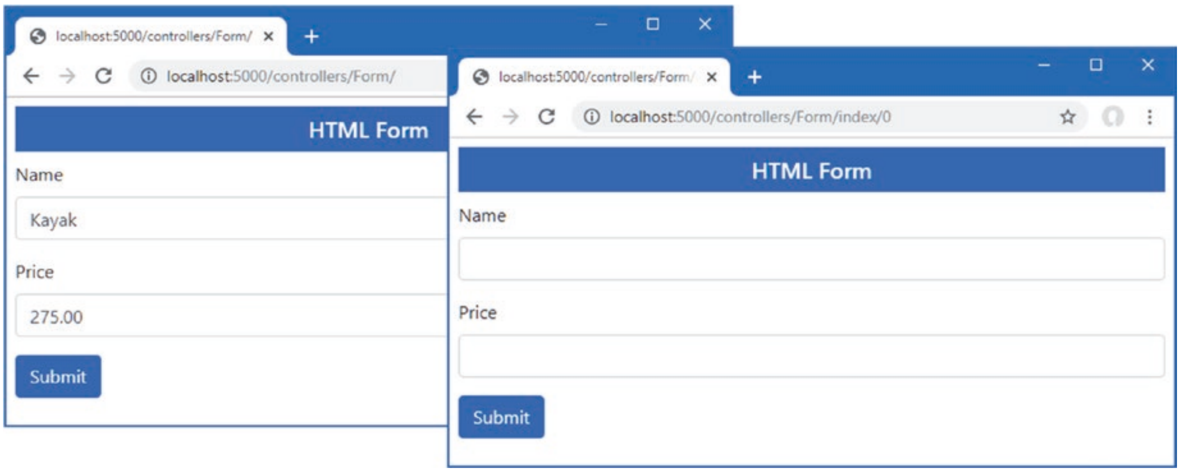

Figure 28-7. *Using a nullable type to determine whether a request contains a value*

Binding Complex Types

The model binding system shines when dealing with complex types, which are any type that cannot be parsed from a single string value. The model binding process inspects the complex type and performs the binding process on each of the `public` properties it defines. This means that instead of dealing with individual values such as `name` and `price`, I can use the binder to create complete `Product` objects, as shown in Listing 28-10.

Listing 28-10. Binding a Complex Type in the FormController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using System.Threading.Tasks;
using WebApp.Models;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Mvc.Rendering;

namespace WebApp.Controllers {

    [AutoValidateAntiforgeryToken]
    public class FormController : Controller {
        private DataContext context;

        public FormController(DataContext dbContext) {
            context = dbContext;
        }

        public async Task<IActionResult> Index(long? id) {
            ViewBag.Categories
                = new SelectList(context.Categories, "CategoryId", "Name");
            return View("Form", await context.Products.Include(p => p.Category)
                .Include(p => p.Supplier)
                .FirstOrDefaultAsync(p => id == null || p.ProductId == id));
        }
```

```
        [HttpPost]
        public IActionResult SubmitForm(Product product) {
            TempData["product"] = System.Text.Json.JsonSerializer.Serialize(product);
            return RedirectToAction(nameof(Results));
        }

        public IActionResult Results() {
            return View(TempData);
        }
    }
}
```

The listing changes the SubmitForm action method so that it defines a Product parameter. Before the action method is invoked, a new Product object is created, and the model binding process is applied to each of its public properties. The SubmitForm method is then invoked, using the Product object as its argument.

To see the model binding process, restart ASP.NET Core, navigate to http://localhost:5000/controllers/Form, and click the Submit button. The model binding process will extract the data values from the request and produce the result shown in Figure 28-8. The Product object created by the model binding process is serialized as JSON data so that it can be stored as temp data, making it easy to see the request data.

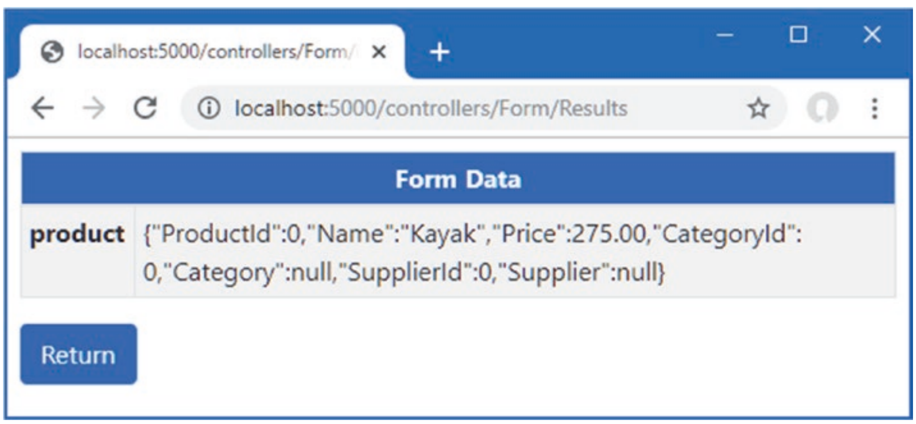

Figure 28-8. *Data binding a complex type*

The data binding process for complex types remains a best-effort feature, meaning that a value will be sought for each public property defined by the Product class, but missing values won't prevent the action method from being invoked. Instead, properties for which no value can be located will be left as the default value for the property type. The example provided values for the Name and Price properties, but the ProductId, CategoryId, and SupplierId properties are zero, and the Category and Supplier properties are null.

Binding to a Property

Using parameters for model binding doesn't fit with the Razor Pages development style because the parameters often duplicate properties defined by the page model class, as shown in Listing 28-11.

Listing 28-11. Binding a Complex Type in the FormHandler.cshtml File in the Pages Folder

```
...
@functions {

    public class FormHandlerModel : PageModel {
        private DataContext context;
```

```
        public FormHandlerModel(DataContext dbContext) {
            context = dbContext;
        }

        public Product Product { get; set; }

        public async Task OnGetAsync(long id = 1) {
            Product = await context.Products.Include(p => p.Category)
                .Include(p => p.Supplier).FirstAsync(p => p.ProductId == id);
        }

        public IActionResult OnPost(Product product) {
            TempData["product"] = System.Text.Json.JsonSerializer.Serialize(product);
            return RedirectToPage("FormResults");
        }
    }
}
...
```

This code works, but the `OnPost` handler method has its own version of the `Product` object, mirroring the property used by the `OnGetAsync` handler. A more elegant approach is to use the existing property for model binding, as shown in Listing 28-12.

Listing 28-12. Using a Property for Model Binding in the FormHandler.cshtml File in the Pages Folder

```
@page "/pages/form/{id:long?}"
@model FormHandlerModel
@using Microsoft.AspNetCore.Mvc.RazorPages
@using Microsoft.EntityFrameworkCore

<div class="m-2">
    <h5 class="bg-primary text-white text-center p-2">HTML Form</h5>
    <form asp-page="FormHandler" method="post" id="htmlform">
        <div class="form-group">
            <label>Name</label>
            <input class="form-control" asp-for="Product.Name" />
        </div>
        <div class="form-group">
            <label>Price</label>
            <input class="form-control" asp-for="Product.Price"  />
        </div>
        <button type="submit" class="btn btn-primary">Submit</button>
    </form>
</div>

@functions {

    public class FormHandlerModel : PageModel {
        private DataContext context;

        public FormHandlerModel(DataContext dbContext) {
            context = dbContext;
        }

        [BindProperty]
        public Product Product { get; set; }

        public async Task OnGetAsync(long id = 1) {
            Product = await context.Products.Include(p => p.Category)
```

```
                .Include(p => p.Supplier).FirstAsync(p => p.ProductId == id);
        }

        public IActionResult OnPost() {
            TempData["product"] = System.Text.Json.JsonSerializer.Serialize(Product);
            return RedirectToPage("FormResults");
        }
    }
}
```

Decorating a property with the BindProperty attribute indicates that its properties should be subject to the model binding process, which means the OnPost handler method can get the data it requires without declaring a parameter. When the BindProperty attribute is used, the model binder uses the property name when locating data values, so the explicit name attributes added to the input element are not required. By default, BindProperty won't bind data for GET requests, but this can be changed by setting the BindProperty attribute's SupportsGet argument to true.

■ **Note** The BindProperties attribute can be applied to classes that require the model binding process for all the public properties they define, which can be more convenient than applying BindProperty to many individual properties. Decorate properties with the BindNever attribute to exclude them from model binding.

Binding Nested Complex Types

If a property that is subject to model binding is defined using a complex type, then the model binding process is repeated using the property name as a prefix. For example, the Product class defines the Category property, whose type is the complex Category type. Listing 28-13 adds elements to the HTML form to provide the model binder with values for the properties defined by the Category class.

Listing 28-13. Adding Nested Form Elements in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<form asp-action="submitform" method="post" id="htmlform">
    <div class="form-group">
        <label asp-for="Name"></label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label asp-for="Price"></label>
        <input class="form-control" asp-for="Price" />
    </div>
    <div class="form-group">
        <label>Category Name</label>
        <input class="form-control" name="Category.Name"
             value="@Model.Category.Name" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

The name attribute combines the property names, separated by periods. In this case, the element is for the Name property of the object assigned to the view model's Category property, so the name attribute is set to Category.Name. The input element tag helper will automatically use this format for the name attribute when the asp-for attribute is applied, as shown in Listing 28-14.

Listing 28-14. Using a Tag Helper in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<form asp-action="submitform" method="post" id="htmlform">
    <div class="form-group">
        <label asp-for="Name"></label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label asp-for="Price"></label>
        <input class="form-control" asp-for="Price" />
    </div>
    <div class="form-group">
        <label>Category Name</label>
        <input class="form-control" asp-for="Category.Name"  />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

The tag helper is a more reliable method of creating elements for nested properties and avoids the risk of typos producing elements that are ignored by the model binding process. To see the effect of the new elements, request `http://localhost:5000/controllers/Form` and click the Submit button, which will produce the response shown in Figure 28-9.

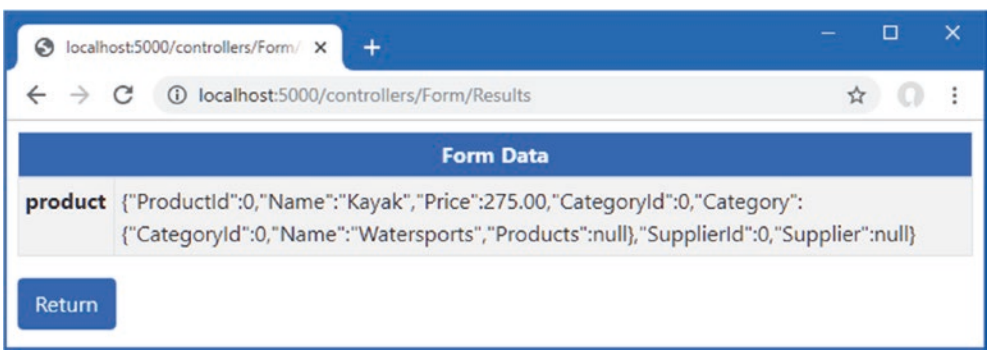

Figure 28-9. *Model binding a nested property*

During the model binding process, a new `Category` object is created and assigned to the `Category` property of the `Product` object. The model binder locates the value for the `Category` object's `Name` property, which can be seen in the figure, but there is no value for the `CategoryId` property, which is left as the default value.

Specifying Custom Prefixes for Nested Complex Types

There are occasions when the HTML you generate relates to one type of object but you want to bind it to another. This means that the prefixes containing the view won't correspond to the structure that the model binder is expecting, and your data won't be properly processed. Listing 28-15 demonstrates this problem by changing the type of the parameter defined by the controller's `SubmitForm` action method.

Listing 28-15. Changing a Parameter in the FormController.cs File in the Controllers Folder

```
...
[HttpPost]
public IActionResult SubmitForm(Category category) {
    TempData["category"] = System.Text.Json.JsonSerializer.Serialize(category);
    return RedirectToAction(nameof(Results));
}
...
```

The new parameter is a `Category`, but the model binding process won't be able to pick out the data values correctly, even though the form data sent by the `Form` view will contain a value for the `Category` object's `Name` property. Instead, the model binder will find the `Name` value for the `Product` object and use that instead, which you can see by restarting ASP.NET Core, requesting `http://localhost:5000/controllers/Form`, and submitting the form data, which will produce the first response shown in Figure 28-10.

This problem is solved by applying the `Bind` attribute to the parameter and using the `Prefix` argument to specify a prefix for the model binder, as shown in Listing 28-16.

Listing 28-16. Setting a Prefix in the FormController.cs File in the Controllers Folder

```
...
[HttpPost]
public IActionResult SubmitForm([Bind(Prefix ="Category")] Category category) {
    TempData["category"] = System.Text.Json.JsonSerializer.Serialize(category);
    return RedirectToAction(nameof(Results));
}
...
```

The syntax is awkward, but the attribute ensures the model binder can locate the data the action method requires. In this case, setting the prefix to `Category` ensures the correct data values are used to bind the `Category` parameter. Restart ASP.NET Core, request `http://localhost:5000/controllers/form`, and submit the form, which produces the second response shown in Figure 28-10.

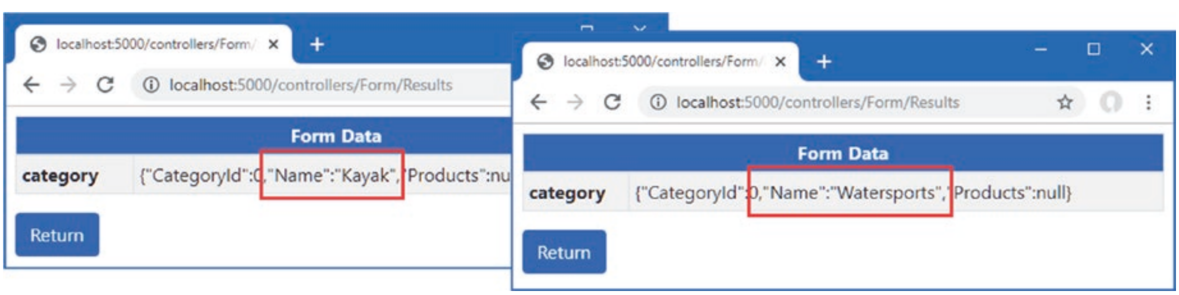

Figure 28-10. *Specifying a model binding prefix*

When using the `BindProperty` attribute, the prefix is specified using the `Name` argument, as shown in Listing 28-17.

Listing 28-17. Specifying a Model Binding Prefix in the FormHandler.cshtml File in the Pages Folder

```
@page "/pages/form/{id:long?}"
@model FormHandlerModel
@using Microsoft.AspNetCore.Mvc.RazorPages
@using Microsoft.EntityFrameworkCore

<div class="m-2">
    <h5 class="bg-primary text-white text-center p-2">HTML Form</h5>
    <form asp-page="FormHandler" method="post" id="htmlform">
        <div class="form-group">
```

```
            <label>Name</label>
            <input class="form-control" asp-for="Product.Name" />
        </div>
        <div class="form-group">
            <label>Price</label>
            <input class="form-control" asp-for="Product.Price"  />
        </div>
        <div class="form-group">
            <label>Category Name</label>
            <input class="form-control" asp-for="Product.Category.Name"  />
        </div>
        <button type="submit" class="btn btn-primary">Submit</button>
    </form>
</div>

@functions {

    public class FormHandlerModel : PageModel {
        private DataContext context;

        public FormHandlerModel(DataContext dbContext) {
            context = dbContext;
        }

        [BindProperty]
        public Product Product { get; set; }

        [BindProperty(Name = "Product.Category")]
        public Category Category { get; set; }

        public async Task OnGetAsync(long id = 1) {
            Product = await context.Products.Include(p => p.Category)
                .Include(p => p.Supplier).FirstAsync(p => p.ProductId == id);
        }

        public IActionResult OnPost() {
            TempData["product"] = System.Text.Json.JsonSerializer.Serialize(Product);
            TempData["category"]
                = System.Text.Json.JsonSerializer.Serialize(Category);
            return RedirectToPage("FormResults");
        }
    }
}
```

This listing adds an input element that uses the asp-for attribute to select the Product.Category property. A page handler class defined a Category property that is decorated with the BindProperty attribute and configured with the Name argument. To see the result of the model binding process, use a browser to request http://localhost:5000/pages/form and click the Submit button. The model binding finds values for both the decorated properties, which produces the response shown in Figure 28-11.

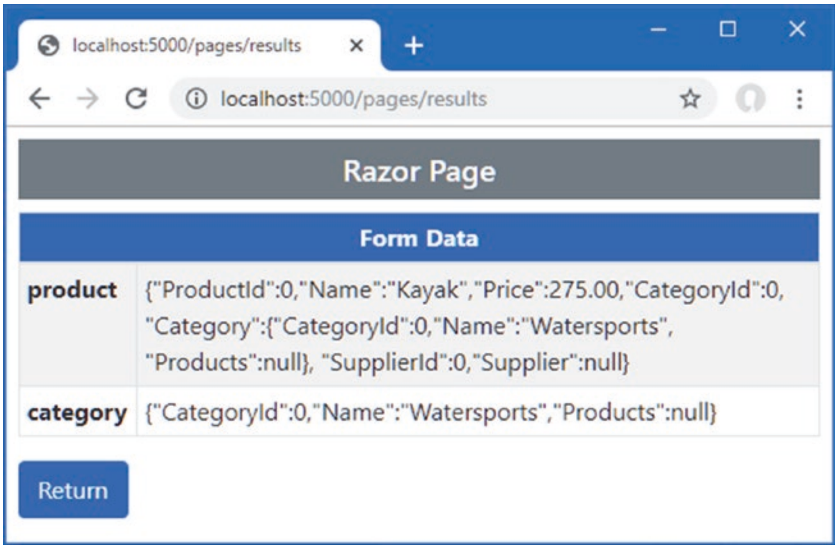

Figure 28-11. *Specifying a model binding prefix in a Razor Page*

Selectively Binding Properties

Some model classes define properties that are sensitive and for which the user should not be able to specify values. A user may be able to change the category for a `Product` object, for example, but should not be able to alter the price.

You might be tempted to simply create views that omit HTML elements for sensitive properties but that won't prevent malicious users from crafting HTTP requests that contain values anyway, which is known as an *over-binding attack*. To prevent the model binder from using values for sensitive properties, the list of properties that should be bound can be specified, as shown in Listing 28-18.

Listing 28-18. Selectively Binding Properties in the FormController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using System.Threading.Tasks;
using WebApp.Models;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Mvc.Rendering;

namespace WebApp.Controllers {

    [AutoValidateAntiforgeryToken]
    public class FormController : Controller {
        private DataContext context;

        public FormController(DataContext dbContext) {
            context = dbContext;
        }

        public async Task<IActionResult> Index(long? id) {
            ViewBag.Categories
                = new SelectList(context.Categories, "CategoryId", "Name");
            return View("Form", await context.Products.Include(p => p.Category)
                .Include(p => p.Supplier)
                .FirstOrDefaultAsync(p => id == null || p.ProductId == id));
        }
```

```
        [HttpPost]
        public IActionResult SubmitForm([Bind("Name", "Category")] Product product) {
            TempData["name"] = product.Name;
            TempData["price"] = product.Price.ToString();
            TempData["category name"] = product.Category.Name;
            return RedirectToAction(nameof(Results));
        }

        public IActionResult Results() {
            return View(TempData);
        }
    }
}
```

I have returned to the Product type for the action method parameter, which has been decorated with the Bind attribute to specify the names of the properties that should be included in the model binding process. This example tells the model binding feature to look for values for the Name and Category properties, which excludes any other property from the process. Restart ASP.NET Core, navigate to http://localhost:5000/controller/Form, and submit the form. Even though the browser sends a value for the Price property as part of the HTTP POST request, it is ignored by the model binder, as shown in Figure 28-12.

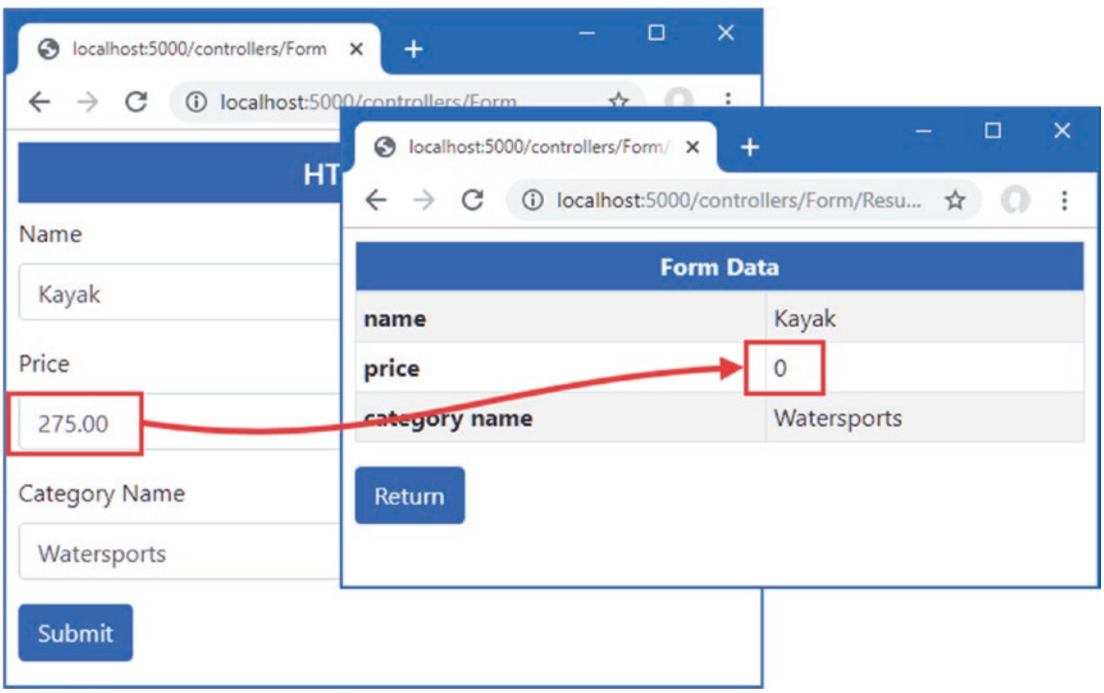

Figure 28-12. *Selectively binding properties*

Selectively Binding in the Model Class

If you are using Razor Pages or you want to use the same set of properties for model binding throughout the application, you can apply the BindNever attribute directly to the model class, as shown in Listing 28-19.

Listing 28-19. Decorating a Property in the Product.cs File in the Models Folder

```
using System.ComponentModel.DataAnnotations.Schema;
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc.ModelBinding;
```

```
namespace WebApp.Models {
    public class Product {

        public long ProductId { get; set; }

        public string Name { get; set; }

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

        public long CategoryId { get; set; }
        public Category Category { get; set; }

        public long SupplierId { get; set; }
        public Supplier Supplier { get; set; }
    }
}
```

The BindNever attribute excludes a property from the model binder, which has the same effect as omitting it from the list used in the previous section. To see the effect, restart ASP.NET Core so the change to the Product class takes effect, request http://localhost:5000/pages/form, and submit the form. Just as with the previous example, the model binder ignores the value for the Price property, as shown in Figure 28-13.

■ **Tip** There is also a BindRequired attribute that tells the model binding process that a request must include a value for a property. If the request doesn't have a required value, then a model validation error is produced, as described in Chapter 29.

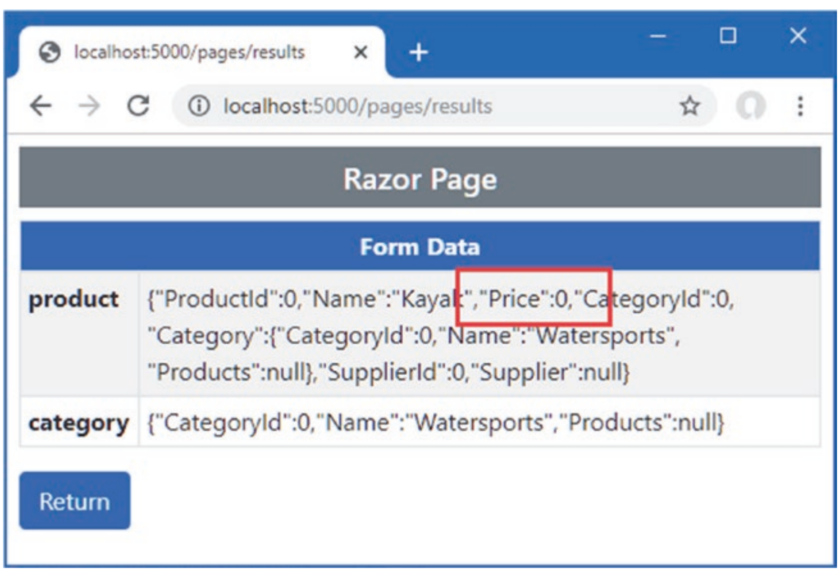

Figure 28-13. *Excluding a property from model binding*

Binding to Arrays and Collections

The model binding process has some nice features for binding request data to arrays and collections, which I demonstrate in the following sections.

Binding to Arrays

One elegant feature of the default model binder is how it supports arrays. To see how this feature works, add a Razor Page named `Bindings.cshtml` to the `Pages` folder with the content shown in Listing 28-20.

Listing 28-20. The Contents of the Bindings.cshtml File in the Pages Folder

```
@page "/pages/bindings"
@model BindingsModel
@using Microsoft.AspNetCore.Mvc
@using Microsoft.AspNetCore.Mvc.RazorPages

<div class="container-fluid">
    <div class="row">
        <div class="col">
            <form asp-page="Bindings" method="post">
                <div class="form-group">
                    <label>Value #1</label>
                    <input class="form-control" name="Data" value="Item 1" />
                </div>
                <div class="form-group">
                    <label>Value #2</label>
                    <input class="form-control" name="Data" value="Item 2" />
                </div>
                <div class="form-group">
                        <label>Value #3</label>
                    <input class="form-control" name="Data" value="Item 3" />
                </div>
                <button type="submit" class="btn btn-primary">Submit</button>
                <a class="btn btn-secondary" asp-page="Bindings">Reset</a>
            </form>
        </div>
        <div class="col">
            <ul class="list-group">
                @foreach (string s in Model.Data.Where(s => s != null)) {
                    <li class="list-group-item">@s</li>
                }
            </ul>
        </div>
    </div>
</div>

@functions {

    public class BindingsModel : PageModel {

        [BindProperty(Name = "Data")]
        public string[] Data { get; set; } = Array.Empty<string>();
    }
}
```

Model binding for an array requires setting the name attribute to the same value for all the elements that will provide an array value. This page displays three input elements, all of which have a name attribute value of Data. To allow the model binder to find the array values, I have decorated the page model's Data property with the BindProperty attribute and used the Name argument.

■ **Tip** Notice that the page model class in Listing 28-20 defines no handler methods. This is unusual, but it works because there is no explicit processing required for any requests since requests only provide values for and display the Data array.

When the HTML form is submitted, a new array is created and populated with the values from all three input elements, which are displayed to the user. To see the binding process, request http://localhost:5000/pages/bindings, edit the form fields, and click the Submit button. The contents of the Data array are displayed in a list using an @foreach expression, as shown in Figure 28-14.

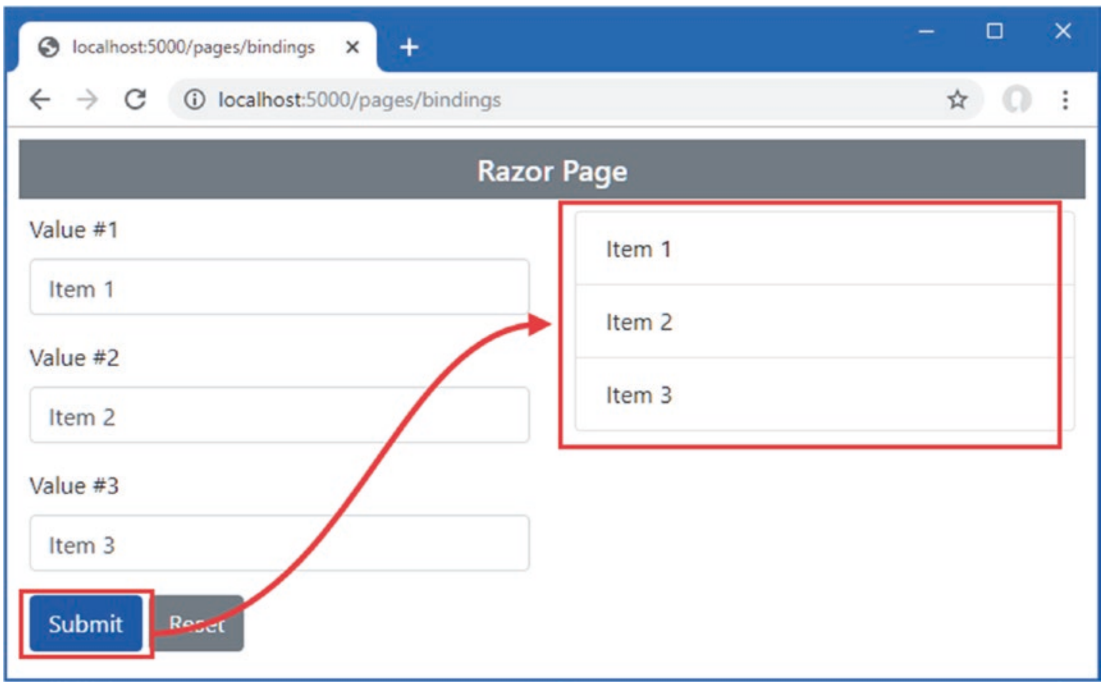

Figure 28-14. *Model binding for array values*

Notice that I filter out null values when displaying the array contents.

```
...
@foreach (string s in Model.Data.Where(s => s != null)) {
    <li class="list-group-item">@s</li>
}
...
```

Empty form fields produce null values in the array, which I don't want to show in the results. In Chapter 29, I show you how to ensure that values are provided for model binding properties.

Specifying Index Positions for Array Values

By default, arrays are populated in the order in which the form values are received from the browser, which will generally be the order in which the HTML elements are defined. The name attribute can be used to specify the position of values in the array if you need to override the default, as shown in Listing 28-21.

Listing 28-21. Specifying Array Position in the Bindings.cshtml File in the Pages Folder

```
@page "/pages/bindings"
@model BindingsModel
@using Microsoft.AspNetCore.Mvc
@using Microsoft.AspNetCore.Mvc.RazorPages

<div class="container-fluid">
    <div class="row">
        <div class="col">
            <form asp-page="Bindings" method="post">
                <div class="form-group">
                    <label>Value #1</label>
                    <input class="form-control" name="Data[1]" value="Item 1" />
                </div>
                <div class="form-group">
                    <label>Value #2</label>
                    <input class="form-control" name="Data[0]" value="Item 2" />
                </div>
                <div class="form-group">
                        <label>Value #3</label>
                    <input class="form-control" name="Data[2]" value="Item 3" />
                </div>
                <button type="submit" class="btn btn-primary">Submit</button>
                <a class="btn btn-secondary" asp-page="Bindings">Reset</a>
            </form>
        </div>
        <div class="col">
            <ul class="list-group">
                @foreach (string s in Model.Data.Where(s => s != null)) {
                    <li class="list-group-item">@s</li>
                }
            </ul>
        </div>
    </div>
</div>

@functions {

    public class BindingsModel : PageModel {

        [BindProperty(Name = "Data")]
        public string[] Data { get; set; } = Array.Empty<string>();
    }
}
```

The array index notation is used to specify the position of a value in the data-bound array. Use a browser to request `http://localhost:5000/pages/bindings` and submit the form, and you will see the items appear in the order dictated by the `name` attributes, as shown in Figure 28-15. The index notation must be applied to all the HTML elements that provide array values, and there must not be any gaps in the numbering sequence.

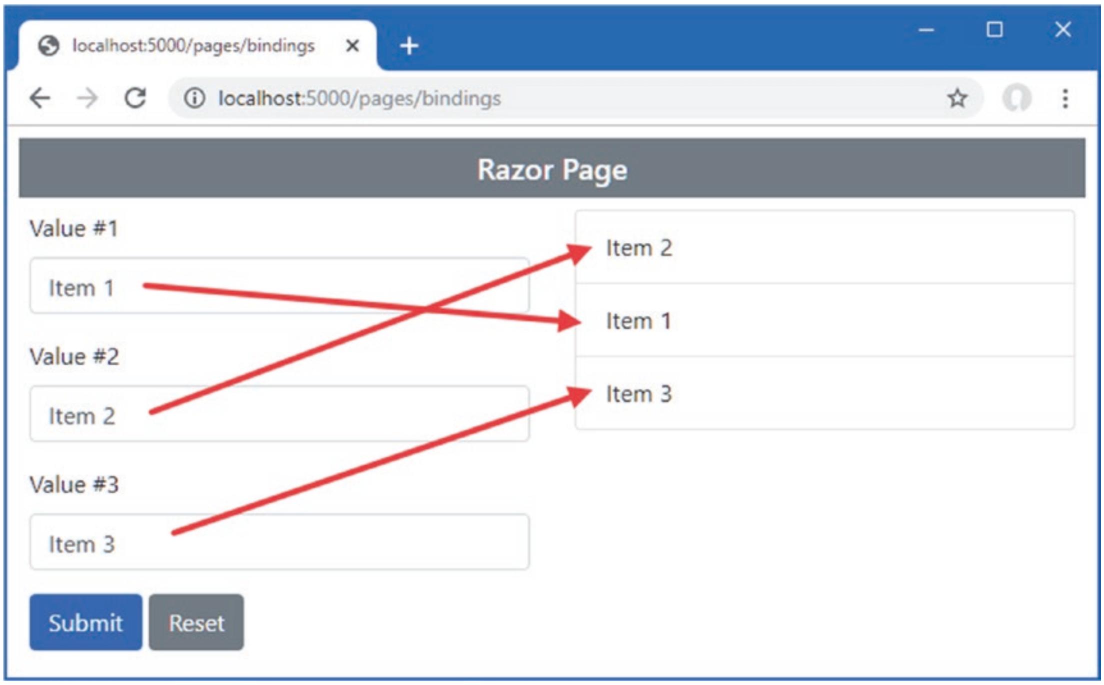

Figure 28-15. *Specifying array position*

Binding to Simple Collections

The model binding process can create collections as well as arrays. For sequence collections, such as lists and sets, only the type of the property or parameter that is used by the model binder is changed, as shown in Listing 28-22.

Listing 28-22. Binding to a List in the Bindings.cshtml File in the Pages Folder

```
@page "/pages/bindings"
@model BindingsModel
@using Microsoft.AspNetCore.Mvc
@using Microsoft.AspNetCore.Mvc.RazorPages

<div class="container-fluid">
    <div class="row">
        <div class="col">
            <form asp-page="Bindings" method="post">
                <div class="form-group">
                    <label>Value #1</label>
                    <input class="form-control" name="Data[1]" value="Item 1" />
                </div>
                <div class="form-group">
                    <label>Value #2</label>
                    <input class="form-control" name="Data[0]" value="Item 2" />
                </div>
                <div class="form-group">
                        <label>Value #3</label>
                    <input class="form-control" name="Data[2]" value="Item 3" />
                </div>
                <button type="submit" class="btn btn-primary">Submit</button>
                <a class="btn btn-secondary" asp-page="Bindings">Reset</a>
            </form>
        </div>
```

```
        <div class="col">
            <ul class="list-group">
                @foreach (string s in Model.Data.Where(s => s != null)) {
                    <li class="list-group-item">@s</li>
                }
            </ul>
        </div>
    </div>
</div>

@functions {

    public class BindingsModel : PageModel {

        [BindProperty(Name = "Data")]
        public SortedSet<string> Data { get; set; } = new SortedSet<string>();
    }
}
```

I changed the type of the `Data` property to `SortedSet<string>`. The model binding process will populate the set with the values from the `input` elements, which will be sorted alphabetically. I have left the index notation on the `input` element `name` attributes, but they have no effect since the collection class will sort its values alphabetically. To see the effect, use a browser to request `http://localhost:5000/pages/bindings`, edit the text fields, and click the Submit button. The model binding process will populate the sorted set with the form values, which will be presented in order, as shown in Figure 28-16.

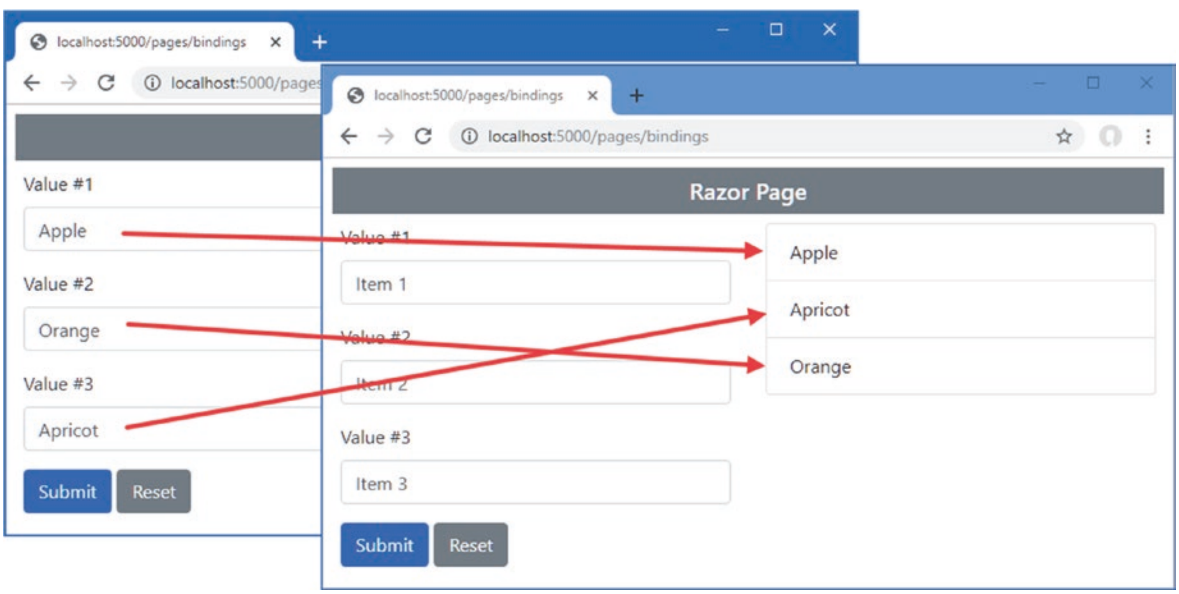

Figure 28-16. *Model binding to a collection*

Binding to Dictionaries

For elements whose `name` attribute is expressed using the index notation, the model binder will use the index as the key when binding to a `Dictionary`, allowing a series of elements to be transformed into key/value pairs, as shown in Listing 28-23.

Listing 28-23. Binding to a Dictionary in the Bindings.cshtml File in the Pages Folder

```
@page "/pages/bindings"
@model BindingsModel
@using Microsoft.AspNetCore.Mvc
@using Microsoft.AspNetCore.Mvc.RazorPages

<div class="container-fluid">
    <div class="row">
        <div class="col">
            <form asp-page="Bindings" method="post">
                <div class="form-group">
                    <label>Value #1</label>
                    <input class="form-control" name="Data[first]" value="Item 1" />
                </div>
                <div class="form-group">
                    <label>Value #2</label>
                    <input class="form-control" name="Data[second]" value="Item 2" />
                </div>
                <div class="form-group">
                        <label>Value #3</label>
                    <input class="form-control" name="Data[third]" value="Item 3" />
                </div>
                <button type="submit" class="btn btn-primary">Submit</button>
                <a class="btn btn-secondary" asp-page="Bindings">Reset</a>
            </form>
        </div>
        <div class="col">
            <table class="table table-sm table-striped">
                <tbody>
                    @foreach (string key in Model.Data.Keys) {
                        <tr>
                            <th>@key</th><td>@Model.Data[key]</td>
                        </tr>
                    }
                </tbody>
            </table>
        </div>
    </div>
</div>

@functions {

    public class BindingsModel : PageModel {

        [BindProperty(Name = "Data")]
        public Dictionary<string, string> Data { get; set; }
            = new Dictionary<string, string>();
    }
}
```

All elements that provide values for the collection must share a common prefix, which is `Data` in this example, followed by the key value in square brackets. The keys for this example are the strings `first`, `second`, and `third`, and will be used as the keys for the content the user provides in the text fields. To see the binding process, request `http://localhost:5000/pages/bindings`, edit the text fields, and submit the form. The keys and values from the form data will be displayed in a table, as shown in Figure 28-17.

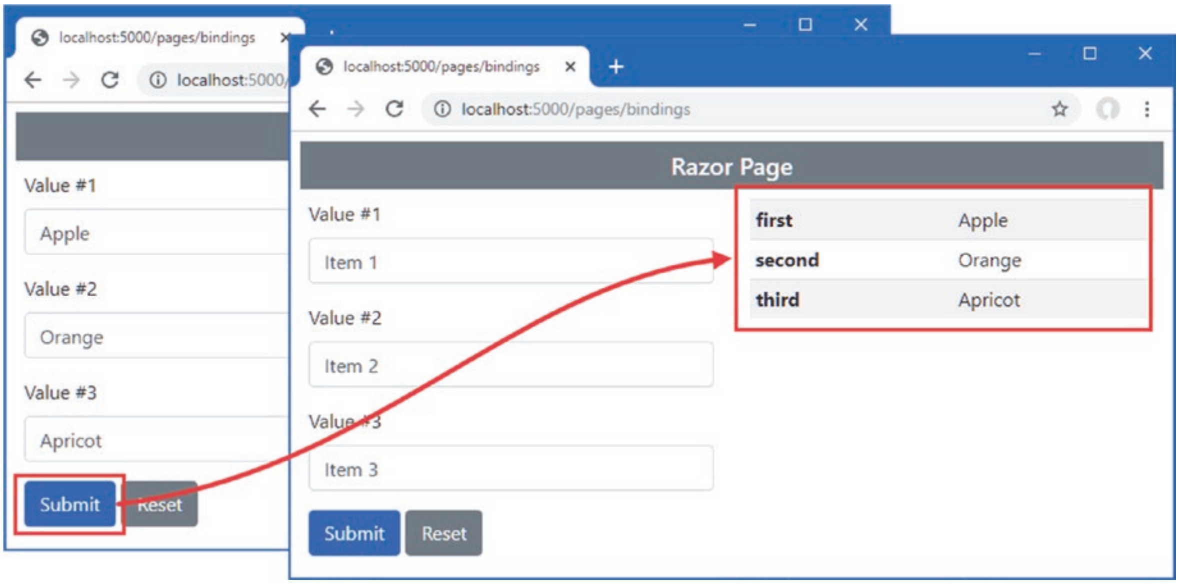

Figure 28-17. *Model binding to a dictionary*

Binding to Collections of Complex Types

The examples in this section have all been collections of simple types, but the same process can be used for complex types, too. To demonstrate, Listing 28-24 revises the Razor Page to gather details used to bind to an array of Product objects.

Listing 28-24. Binding to Complex Types in the Bindings.cshtml File in the Pages Folder

```
@page "/pages/bindings"
@model BindingsModel
@using Microsoft.AspNetCore.Mvc
@using Microsoft.AspNetCore.Mvc.RazorPages

<div class="container-fluid">
    <div class="row">
        <div class="col">
            <form asp-page="Bindings" method="post">
                @for (int i = 0; i < 2; i++) {
                    <div class="form-group">
                        <label>Name #@i</label>
                        <input class="form-control" name="Data[@i].Name"
                            value="Product-@i" />
                    </div>
                    <div class="form-group">
                        <label>Price #@i</label>
                        <input class="form-control" name="Data[@i].Price"
                            value="@(100 + i)" />
                    </div>
                }
                <button type="submit" class="btn btn-primary">Submit</button>
                <a class="btn btn-secondary" asp-page="Bindings">Reset</a>
            </form>
        </div>
```

```
        <div class="col">
            <table class="table table-sm table-striped">
                <tbody>
                    <tr><th>Name</th><th>Price</th></tr>
                    @foreach (Product p in Model.Data) {
                        <tr>
                            <td>@p.Name</td><td>@p.Price</td>
                        </tr>
                    }
                </tbody>
            </table>
        </div>
    </div>
</div>

@functions {

    public class BindingsModel : PageModel {

        [BindProperty(Name = "Data")]
        public Product[] Data { get; set; } = Array.Empty<Product>();
    }
}
```

The name attributes for the input elements use the array notation, followed by a period, followed by the name of the complex type properties they represent. To define elements for the Name and Price properties, this requires elements like this:

```
...
<input class="form-control" name="Data[0].Name" />
...
<input class="form-control" name="Data[0].Price" />
...
```

During the binding process, the model binder will attempt to locate values for all the public properties defined by the target type, repeating the process for each set of values in the form data.

This example relies on model binding for the Price property defined by the Product class, which was excluded from the binding process with the BindNever attribute. Remove the attribute from the property, as shown in Listing 28-25.

Listing 28-25. Removing an Attribute in the Product.cs File in the Models Folder

```
using System.ComponentModel.DataAnnotations.Schema;
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace WebApp.Models {
    public class Product {

        public long ProductId { get; set; }

        public string Name { get; set; }

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

```
        public long CategoryId { get; set; }
        public Category Category { get; set; }

        public long SupplierId { get; set; }
        public Supplier Supplier { get; set; }
    }
}
```

Restart ASP.NET Core so the change to the `Product` class takes effect and use a browser to request `http://localhost:5000/pages/bindings`. Enter names and prices into the text fields and submit the form, and you will see the details of the `Product` objects created from the data displayed in a table, as shown in Figure 28-18.

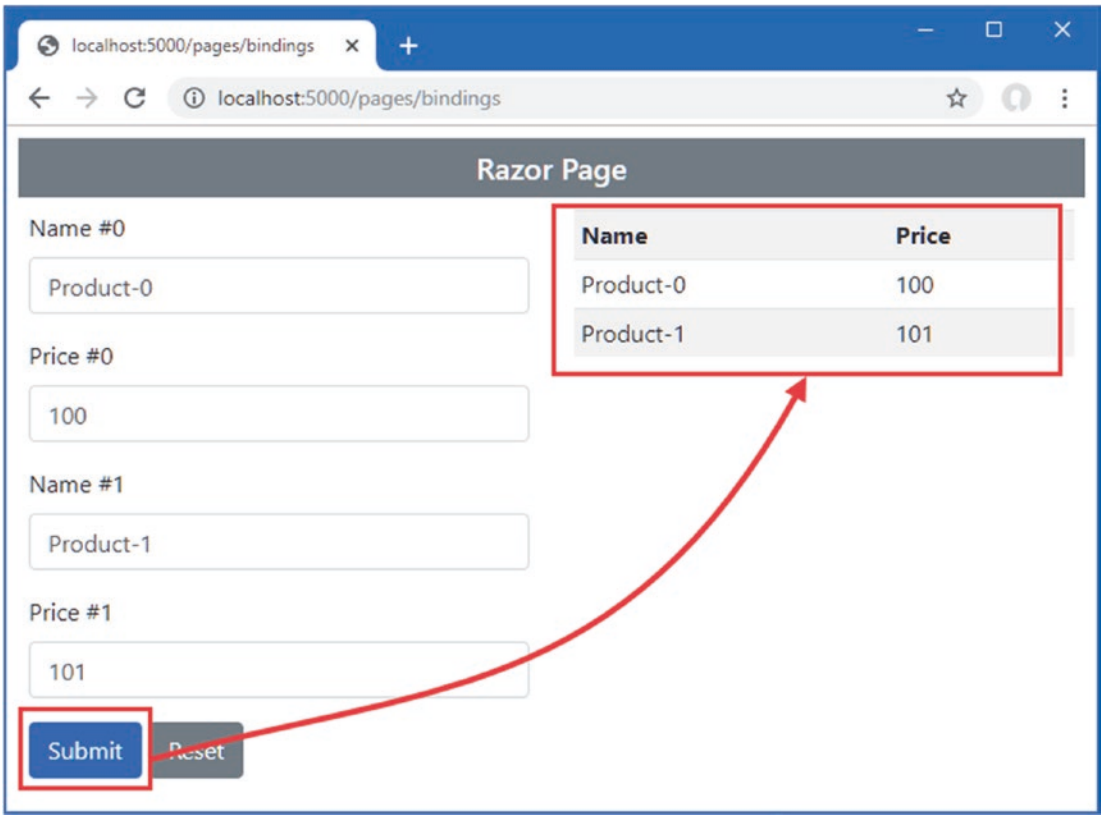

Figure 28-18. *Binding to a collection of complex types*

Specifying a Model Binding Source

As I explained at the start of the chapter, the default model binding process looks for data in four places: the form data values, the request body (for web service controllers only), the routing data, and the request query string.

The default search sequence isn't always helpful, either because you always want data to come from a specific part of the request or because you want to use a data source that isn't searched by default. The model binding feature includes a set of attributes used to override the default search behavior, as described in Table 28-3.

■ **Tip** There is also the `FromService` attribute, which doesn't get a value from the request, but through the dependency injection feature described in Chapter 14.

Table 28-3. The Model Binding Source Attributes

Name	Description
FromForm	This attribute is used to select form data as the source of binding data. The name of the parameter is used to locate a form value by default, but this can be changed using the Name property, which allows a different name to be specified.
FromRoute	This attribute is used to select the routing system as the source of binding data. The name of the parameter is used to locate a route data value by default, but this can be changed using the Name property, which allows a different name to be specified.
FromQuery	This attribute is used to select the query string as the source of binding data. The name of the parameter is used to locate a query string value by default, but this can be changed using the Name property, which allows a different query string key to be specified.
FromHeader	This attribute is used to select a request header as the source of binding data. The name of the parameter is used as the header name by default, but this can be changed using the Name property, which allows a different header name to be specified.
FromBody	This attribute is used to specify that the request body should be used as the source of binding data, which is required when you want to receive data from requests that are not form-encoded, such as in API controllers that provide web services.

The FromForm, FromRoute, and FromQuery attributes allow you to specify that the model binding data will be obtained from one of the standard locations but without the normal search sequence. Earlier in the chapter, I used this URL:

```
http://localhost:5000/controllers/Form/Index/5?id=1
```

This URL contains two possible values that can be used for the id parameter of the Index action method on the Form controller. The routing system will assign the final segment of the URL to a variable called id, which is defined in the default URL pattern for controllers, and the query string also contains an id value. The default search pattern means that the model binding data will be taken from the route data and the query string will be ignored.

In Listing 28-26, I have applied the FromQuery attribute to the id parameter defined by the Index action method, which overrides the default search sequence.

Listing 28-26. Selecting the Query String in the FormController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using System.Threading.Tasks;
using WebApp.Models;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Mvc.Rendering;

namespace WebApp.Controllers {

    [AutoValidateAntiforgeryToken]
    public class FormController : Controller {
        private DataContext context;

        public FormController(DataContext dbContext) {
            context = dbContext;
        }
```

```
        public async Task<IActionResult> Index([FromQuery] long? id) {
            ViewBag.Categories
                = new SelectList(context.Categories, "CategoryId", "Name");
            return View("Form", await context.Products.Include(p => p.Category)
                .Include(p => p.Supplier)
                .FirstOrDefaultAsync(p => id == null || p.ProductId == id));
        }

        [HttpPost]
        public IActionResult SubmitForm([Bind("Name", "Category")] Product product) {
            TempData["name"] = product.Name;
            TempData["price"] = product.Price.ToString();
            TempData["category name"] = product.Category.Name;
            return RedirectToAction(nameof(Results));
        }

        public IActionResult Results() {
            return View(TempData);
        }
    }
}
```

The attribute specifies the source for the model binding process, which you can see by restarting ASP.NET Core and using a browser to request `http://localhost:5000/controllers/Form/Index/5?id=1`. Instead of using the value that has been matched by the routing system, the query string will be used instead, producing the response shown in Figure 28-19. No other location will be used if the query string doesn't contain a suitable value for the model binding process.

■ **Tip** You can still bind complex types when specifying a model binding source such as the query string. For each simple property in the parameter type, the model binding process will look for a query string key with the same name.

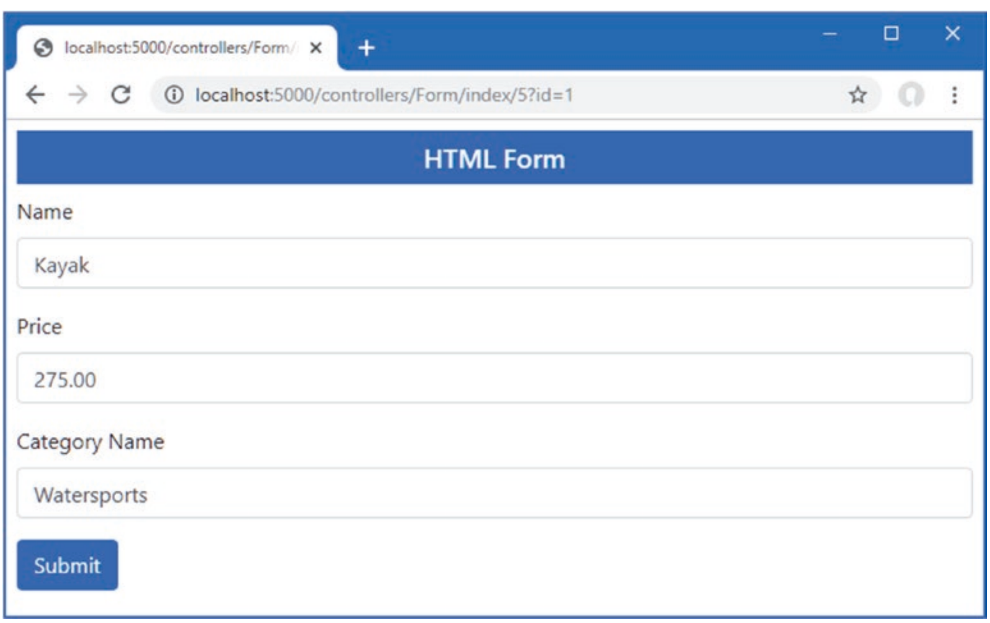

Figure 28-19. *Specifying a model binding data source*

Selecting a Binding Source for a Property

The same attributes can be used to model bind properties defined by a page model or a controller, as shown in Listing 28-27.

Listing 28-27. Selecting the Query String in the Bindings.cshtml File in the Pages Folder

```
...
@functions {

    public class BindingsModel : PageModel {

        //[BindProperty(Name = "Data")]
        [FromQuery(Name = "Data")]
        public Product[] Data { get; set; } = Array.Empty<Product>();
    }
}
...
```

The use of the `FromQuery` attribute means the query string is used as the source of values for the model binder as it creates the `Product` array, which you can see by requesting `http://localhost:5000/pages/bindings?data[0].name=Skis&data[0].price=500`, which produces the response shown in Figure 28-20.

■ **Note** In this example, I have used a GET request because it allows the query string to be easily set. Although it is harmless in such a simple example, care must be taken when sending GET requests that modify the state of the application. As noted previously, making changes in GET requests can lead to problems.

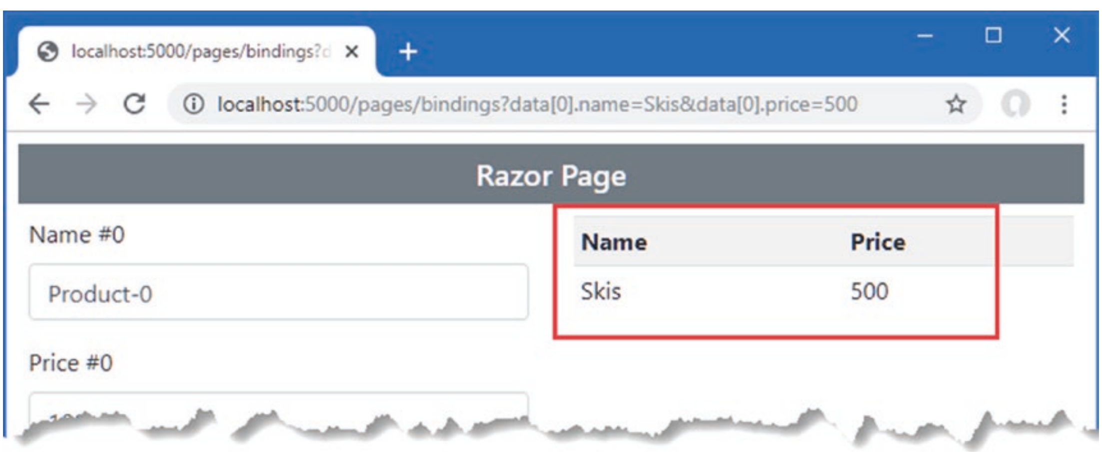

Figure 28-20. *Specifying a model binding data source in a Razor Page*

■ **Tip** Although it is rarely used, you can bind complex types using header values by applying the `FromHeader` attribute to the properties of a model class.

Using Headers for Model Binding

The `FromHeader` attribute allows HTTP request headers to be used as the source for binding data. In Listing 28-28, I have added a simple action method to the `Form` controller that defines a parameter that will be model bound from a standard HTTP request header.

Listing 28-28. Model Binding from a Header in the FormController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using System.Threading.Tasks;
using WebApp.Models;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Mvc.Rendering;

namespace WebApp.Controllers {

    [AutoValidateAntiforgeryToken]
    public class FormController : Controller {
        private DataContext context;

        // ...other action methods omitted for brevity...

        public string Header([FromHeader]string accept) {
            return $"Header: {accept}";
        }
    }
}
```

The Header action method defines an accept parameter, the value for which will be taken from the Accept header in the current request and returned as the method result. Restart ASP.NET Core and request http://localhost:5000/controllers/form/header, and you will see a result like this:

```
Header: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,
    image/apng,*/*;q=0.8,application/signed-exchange;v=b3
```

Not all HTTP header names can be easily selected by relying on the name of the action method parameter because the model binding system doesn't convert from C# naming conventions to those used by HTTP headers. In these situations, you must configure the FromHeader attribute using the Name property to specify the name of the header, as shown in Listing 28-29.

Listing 28-29. Selecting a Header by Name in the FormController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using System.Threading.Tasks;
using WebApp.Models;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Mvc.Rendering;

namespace WebApp.Controllers {

    [AutoValidateAntiforgeryToken]
    public class FormController : Controller {
        private DataContext context;

        // ...other action methods omitted for brevity...

        public string Header([FromHeader(Name = "Accept-Language")] string accept) {
            return $"Header: {accept}";
        }
    }
}
```

I can't use Accept-Language as the name of a C# parameter, and the model binder won't automatically convert a name like AcceptLanguage into Accept-Language so that it matches the header. Instead, I used the Name property to configure the attribute so that it matches the right header. If you restart ASP.NET Core and request http://localhost:5000/controllers/form/header, you will see a result like this, which will vary based on your locale settings:

```
Header: en-US;q=0.9,en;q=0.8
```

Using Request Bodies as Binding Sources

Not all data sent by clients is sent as form data, such as when a JavaScript client sends JSON data to an API controller. The FromBody attribute specifies that the request body should be decoded and used as a source of model binding data. In Listing 28-30, I have added a new action method to the Form controller with a parameter that is decorated with the FromBody attribute.

■ **Tip** The FromBody attribute isn't required for controllers that are decorated with the ApiController attribute.

Listing 28-30. Adding an Action Method in the FormController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using System.Threading.Tasks;
using WebApp.Models;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Mvc.Rendering;

namespace WebApp.Controllers {

    [AutoValidateAntiforgeryToken]
    public class FormController : Controller {
        private DataContext context;

        public FormController(DataContext dbContext) {
            context = dbContext;
        }

        // ...other action methods omitted for brevity...

        [HttpPost]
        [IgnoreAntiforgeryToken]
        public Product Body([FromBody] Product model) {
            return model;
        }
    }
}
```

To test the model binding process, restart ASP.NET Core, open a new PowerShell command prompt, and run the command in Listing 28-31 to send a request to the application.

■ **Note** I added the IgnoreAntiforgeryToken to the action method in Listing 28-31 because the request that I am going to send won't include an anti-forgery token, which I described in Chapter 27.

Listing 28-31. Sending a Request

```
Invoke-RestMethod http://localhost:5000/controllers/form/body -Method POST -Body  (@{ Name="Soccer Boots";
Price=89.99} | ConvertTo-Json) -ContentType "application/json"
```

The JSON-encoded request body is used to model bind the action method parameter, which produces the following response:

```
productId  : 0
name       : Soccer Boots
price      : 89.99
categoryId : 0
category   :
supplierId : 0
supplier   :
```

Manually Model Binding

Model binding is applied automatically when you define a parameter for an action or handler method or apply the `BindProperty` attribute. Automatic model binding works well if you can consistently follow the name conventions and you always want the process to be applied. If you need to take control of the binding process or you want to perform binding selectively, then you can perform model binding manually, as shown in Listing 28-32.

Listing 28-32. Manually Binding in the Bindings.cshtml File in the Pages Folder

```
@page "/pages/bindings"
@model BindingsModel
@using Microsoft.AspNetCore.Mvc
@using Microsoft.AspNetCore.Mvc.RazorPages

<div class="container-fluid">
    <div class="row">
        <div class="col">
            <form asp-page="Bindings" method="post">
                <div class="form-group">
                    <label>Name</label>
                    <input class="form-control" asp-for="Data.Name" />
                </div>
                <div class="form-group">
                    <label>Price</label>
                    <input class="form-control" asp-for="Data.Price"
                           value="@(Model.Data.Price + 1)" />
                </div>
                <div class="form-check m-2">
                    <input class="form-check-input" type="checkbox" name="bind"
                        value="true" checked />
                    <label class="form-check-label">Model Bind?</label>
                </div>
                <button type="submit" class="btn btn-primary">Submit</button>
                <a class="btn btn-secondary" asp-page="Bindings">Reset</a>
            </form>
        </div>
```

```
        <div class="col">
            <table class="table table-sm table-striped">
                <tbody>
                    <tr><th>Name</th><th>Price</th></tr>
                    <tr>
                        <td>@Model.Data.Name</td><td>@Model.Data.Price</td>
                    </tr>
                </tbody>
            </table>
        </div>
    </div>
</div>

@functions {

    public class BindingsModel : PageModel {

        public Product Data { get; set; }
            = new Product() { Name = "Skis", Price = 500 };

        public async Task OnPostAsync([FromForm] bool bind) {
            if (bind) {
               await TryUpdateModelAsync<Product>(Data,
                   "data", p => p.Name, p => p.Price);
            }
        }
    }
}
```

Manual model binding is performed using the TryUpdateModelAsync method, which is provided by the PageModel and ControllerBase classes, which means it is available for both Razor Pages and MVC controllers.

This example mixes automatic and manual model binding. The OnPostAsync method uses automatic model binding to receive a value for its bind parameter, which has been decorated with the FromForm attribute. If the value of the parameter is true, the TryUpdateModelAsync method is used to apply model binding. The arguments to the TryUpdateModelAsync method are the object that will be model bound, the prefix for the values, and a series of expressions that select the properties that will be included in the process, although there are other versions of the TryUpdateModelAsync method available.

The result is that the model binding process for the Data property is performed only when the user checks the checkbox added to the form in Listing 28-32. If the checkbox is unchecked, then no model binding occurs, and the form data is ignored. To make it obvious when model binding is used, the value of the Price property is incremented when the form is rendered. To see the effect, request http://localhost:5000/pages/bindings and submit the form with the checkbox checked and then unchecked, as shown in Figure 28-21.

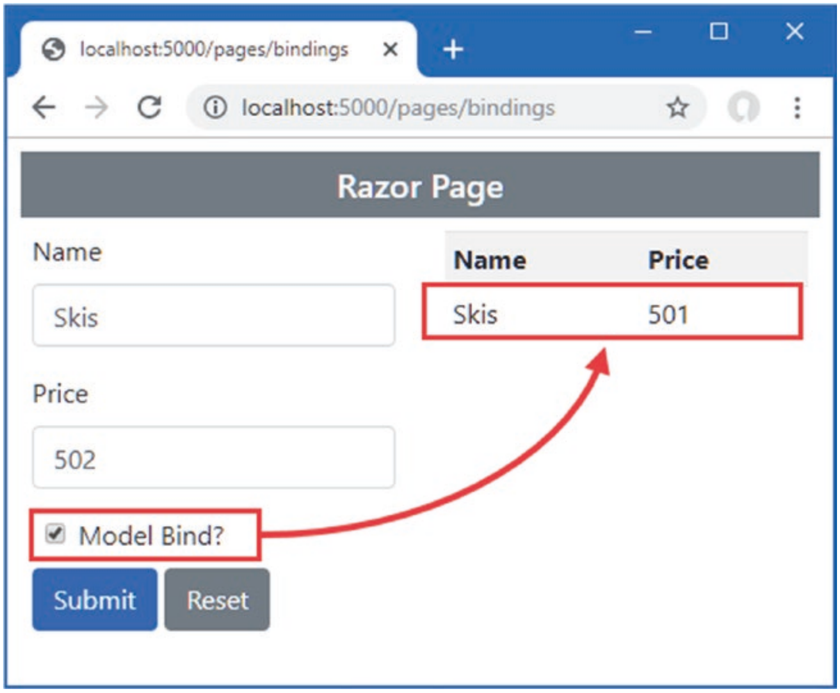

Figure 28-21. *Using manual model binding*

Summary

In this chapter, I introduced the model binding feature, which makes it easy to work with request data. I showed you how to use model binding with parameters and properties, how to bind simple and complex types, and the conventions required to bind to arrays and collections. I also explained how to control which part of the request is used for model binding and how to take control of when model binding is performed. In the next chapter, I describe the features that ASP.NET Core provides to validate form data.

CHAPTER 29

Using Model Validation

In the previous chapter, I showed you how the model binding process creates objects from HTTP requests. Throughout that chapter, I simply displayed the data that the application received. That's because the data that users provide should not be used until it has been inspected to ensure that the application is able to use it. The reality is that users will often enter data that isn't valid and cannot be used, which leads me to the topic of this chapter: *model validation*.

Model validation is the process of ensuring the data received by the application is suitable for binding to the model and, when this is not the case, providing useful information to the user that will help explain the problem.

The first part of the process, checking the data received, is one of the most important ways to preserve the integrity of an application's data. Rejecting data that cannot be used can prevent odd and unwanted states from arising in the application. The second part of the validation process is helping the user correct the problem and is equally important. Without the feedback needed to correct the problem, users become frustrated and confused. In public-facing applications, this means users will simply stop using the application. In corporate applications, this means the user's workflow will be hindered. Neither outcome is desirable, but fortunately, ASP.NET Core provides extensive support for model validation. Table 29-1 puts model validation in context.

Table 29-1. *Putting Model Validation in Context*

Question	Answer
What is it?	Model validation is the process of ensuring that the data provided in a request is valid for use in the application.
Why is it useful?	Users do not always enter valid data, and using it in the application can produce unexpected and undesirable errors.
How is it used?	Controllers and Razor Pages check the outcome of the validation process, and tag helpers are used to include validation feedback in views displayed to the user. Validation can be performed automatically during the model binding process and can be supplemented with custom validation.
Are there any pitfalls or limitations?	It is important to test the efficacy of your validation code to ensure that it covers the full range of values that the application can receive.
Are there any alternatives?	Model validation is optional, but it is a good idea to use it whenever using model binding.

Table 29-2 summarizes the chapter.

Table 29-2. *Chapter Summary*

Problem	Solution	Listing
Validating data	Manually use the `ModelState` features or apply validation attributes	5, 13–20
Displaying validation messages	Use the classes to which form elements are assigned and the validation tag helpers	6–12
Validating data before the form is submitted	Use client-side and remote validation	21–25

A. Freeman, *Pro ASP.NET Core 3*, https://doi.org/10.1007/978-1-4842-5440-0_29

Preparing for This Chapter

This chapter uses the WebApp project from Chapter 28. To prepare for this chapter, change the contents of the Form controller's Form view so it contains input elements for each of the properties defined by the Product class, excluding the navigation properties used by Entity Framework Core, as shown in Listing 29-1.

■ **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-3. See Chapter 1 for how to get help if you have problems running the examples.

Listing 29-1. Changing Elements in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<form asp-action="submitform" method="post" id="htmlform">
    <div class="form-group">
        <label asp-for="Name"></label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label asp-for="Price"></label>
        <input class="form-control" asp-for="Price" />
    </div>
    <div class="form-group">
        <label>CategoryId</label>
        <input class="form-control" asp-for="CategoryId"  />
    </div>
    <div class="form-group">
        <label>SupplierId</label>
        <input class="form-control" asp-for="SupplierId"  />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

Replace the contents of the FormController.cs file with those shown in Listing 29-2, which adds support for displaying the properties defined in Listing 29-1 and removes model binding attributes and action methods that are no longer required.

Listing 29-2. Replacing the Contents of the FormController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using System.Threading.Tasks;
using WebApp.Models;
using Microsoft.EntityFrameworkCore;

namespace WebApp.Controllers {

    [AutoValidateAntiforgeryToken]
    public class FormController : Controller {
        private DataContext context;

        public FormController(DataContext dbContext) {
            context = dbContext;
        }
```

```
        public async Task<IActionResult> Index(long? id) {
            return View("Form", await context.Products
                .FirstOrDefaultAsync(p => id == null || p.ProductId == id));
        }

        [HttpPost]
        public IActionResult SubmitForm(Product product) {
            TempData["name"] = product.Name;
            TempData["price"] = product.Price.ToString();
            TempData["categoryId"] = product.CategoryId.ToString();
            TempData["supplierId"] = product.SupplierId.ToString();
            return RedirectToAction(nameof(Results));
        }

        public IActionResult Results() {
            return View(TempData);
        }
    }
}
```

Dropping the Database

Open a new PowerShell command prompt, navigate to the folder that contains the `WebApp.csproj` file, and run the command shown in Listing 29-3 to drop the database.

Listing 29-3. Dropping the Database

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

Running the Example Application

Select Start Without Debugging or Run Without Debugging from the Debug menu or use the PowerShell command prompt to run the command shown in Listing 29-4.

Listing 29-4. Running the Example Application

```
dotnet run
```

Use a browser to request `http://localhost:5000/controllers/Form`, which will display an HTML form. Click the Submit button, and the form data will be displayed, as shown in Figure 29-1.

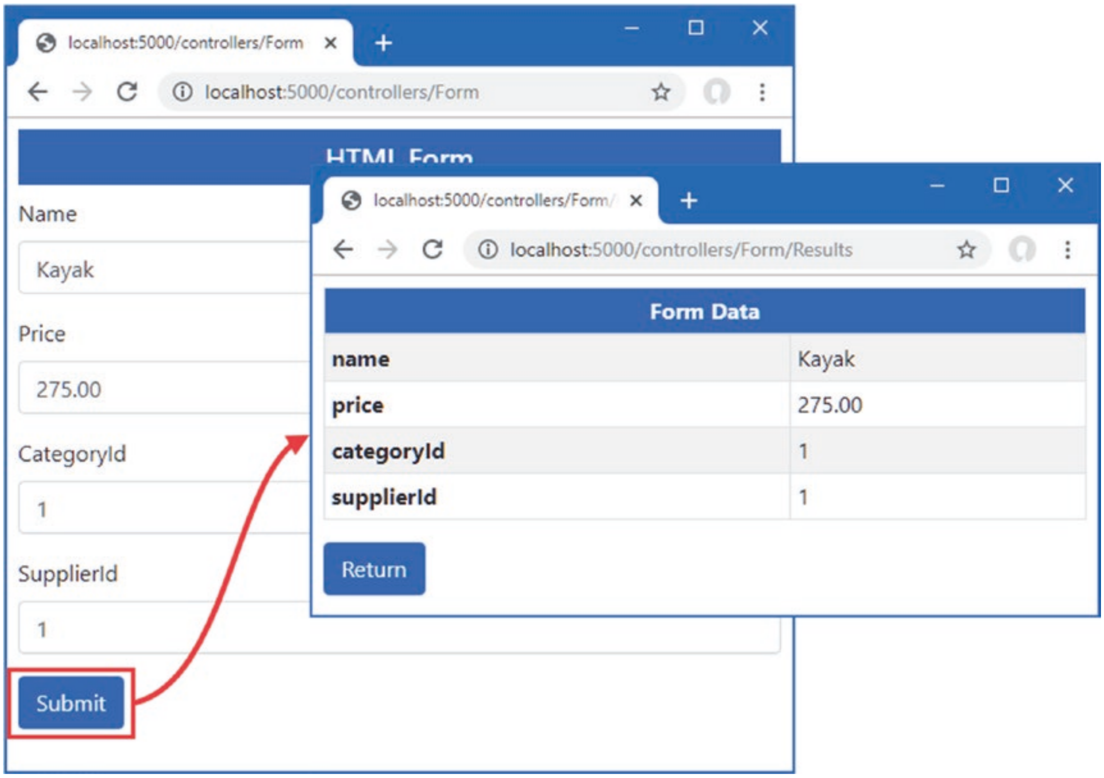

Figure 29-1. *Running the example application*

Understanding the Need for Model Validation

Model validation is the process of enforcing the requirements that an application has for the data it receives from clients. Without validation, an application will try to operate on any data it receives, which can lead to exceptions and unexpected behavior that appear immediately or long-term problems that appear gradually as the database is populated with bad, incomplete, or malicious data.

Currently, the action and handler methods that receive form data will accept any data that the user submits, which is why the examples just display the form data and don't store it in the database.

Most data values have constraints of some sort. This can involve requiring a value to be provided, requiring the value to be a specific type, and requiring the value to fall within a specific range.

As an example, before I can safely store a `Product` object in the database, for example, I need to make sure that the user provides values for the `Name`, `Price`, `CategoryId`, and `SupplierId` properties. The `Name` value can be any valid string, the `Price` property must be a valid currency amount, and the `CategoryId` and `SupplierId` properties must correspond to existing `Supplier` and `Category` products in the database. In the following sections, I demonstrate how model validation can be used to enforce these requirements by checking the data that the application receives and providing feedback to the user when the application cannot use the data the user has submitted.

Explicitly Validating Data in a Controller

The most direct way of validating data is to do so in an action or handler method, as shown in Listing 29-5, recording details of any problems so they can be displayed to the user.

Listing 29-5. Explicitly Validating Data in the FormController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using System.Threading.Tasks;
using WebApp.Models;
```

```
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace WebApp.Controllers {

    [AutoValidateAntiforgeryToken]
    public class FormController : Controller {
        private DataContext context;

        public FormController(DataContext dbContext) {
            context = dbContext;
        }

        public async Task<IActionResult> Index(long? id) {
            return View("Form", await context.Products
                .FirstOrDefaultAsync(p => id == null || p.ProductId == id));
        }

        [HttpPost]
        public IActionResult SubmitForm(Product product) {

            if (string.IsNullOrEmpty(product.Name)) {
                ModelState.AddModelError(nameof(Product.Name), "Enter a name");
            }

            if (ModelState.GetValidationState(nameof(Product.Price))
                    == ModelValidationState.Valid && product.Price < 1) {
                ModelState.AddModelError(nameof(Product.Price),
                    "Enter a positive price");
            }

            if (!context.Categories.Any(c => c.CategoryId == product.CategoryId)) {
                ModelState.AddModelError(nameof(Product.CategoryId),
                    "Enter an existing category ID");
            }

            if (!context.Suppliers.Any(s => s.SupplierId == product.SupplierId)) {
                ModelState.AddModelError(nameof(Product.SupplierId),
                    "Enter an existing supplier ID");
            }

            if (ModelState.IsValid) {
                TempData["name"] = product.Name;
                TempData["price"] = product.Price.ToString();
                TempData["categoryId"] = product.CategoryId.ToString();
                TempData["supplierId"] = product.SupplierId.ToString();
                return RedirectToAction(nameof(Results));
            } else {
                return View("Form");
            }
        }

        public IActionResult Results() {
            return View(TempData);
        }
    }
}
```

For each of the properties of the Product parameter created, I check to see the value provided by the user and record any errors I find using the ModelStateDictionary object that is returned by the ModelState property inherited from the ControllerBase class.

As its name suggests, the ModelStateDictionary class is a dictionary used to track details of the state of the model object, with an emphasis on validation errors. Table 29-3 describes the most important ModelStateDictionary members.

Table 29-3. *Selected ModelStateDictionary Members*

Name	Description
AddModelError(property, message)	This method is used to record a model validation error for the specified property.
GetValidationState(property)	This method is used to determine whether there are model validation errors for a specific property, expressed as a value from the ModelValidationState enumeration.
IsValid	This property returns true if all the model properties are valid and returns false otherwise.
Clear()	This property clears the validation state.

As an example of using the ModelStateDictionary, consider how the Name property was validated.

```
...
if (string.IsNullOrEmpty(product.Name)) {
    ModelState.AddModelError(nameof(Product.Name), "Enter a name");
}
...
```

One of the validation requirements for the Product class is to ensure the user provides a value for the Name property, so I use the static string.IsNullOrEmpty method to test the property value that the model binding process has extracted from the request. If the Name property is null or an empty string, then I know that the value cannot be used by the application, and I use the ModelState.AddModelError method to register a validation error, specifying the name of the property (Name) and a message that will be displayed to the user to explain the nature of the problem (Enter a name).

The ModelStateDictionary is also used during the model binding process to record any problems with finding and assigning values to model properties. The GetValidationState method is used to see whether there have been any errors recorded for a model property, either from the model binding process or because the AddModelError method has been called during explicit validation in the action method. The GetValidationState method returns a value from the ModelValidationState enumeration, which defines the values described in Table 29-4.

Table 29-4. *The ModelValidationState Values*

Name	Description
Unvalidated	This value means that no validation has been performed on the model property, usually because there was no value in the request that corresponded to the property name.
Valid	This value means that the request value associated with the property is valid.
Invalid	This value means that the request value associated with the property is invalid and should not be used.
Skipped	This value means that the model property has not been processed, which usually means that there have been so many validation errors that there is no point continuing to perform validation checks.

For the Price property, I check to see whether the model binding process has reported a problem parsing the value sent by the browser into a decimal value, like this:

```
...
if (ModelState.GetValidationState(nameof(Product.Price))
        == ModelValidationState.Valid && product.Price < 1) {
    ModelState.AddModelError(nameof(Product.Price), "Enter a positive price");
}
...
```

I want to make sure that the user provides a Price value that is equal to or greater than 1, but there is no point in recording an error about zero or negative values if the user has provided a value that the model binder cannot convert into a decimal value. I use the GetValidationState method to determine the validation status of the Price property before performing my own validation check.

After I have validated all the properties in the Product object, I check the ModelState.IsValid property to see whether there were errors. This method returns true if the Model.State.AddModelError method was called during the checks or if the model binder had any problems creating the object.

```
...
if (ModelState.IsValid) {
    TempData["name"] = product.Name;
    TempData["price"] = product.Price.ToString();
    TempData["categoryId"] = product.CategoryId.ToString();
    TempData["supplierId"] = product.SupplierId.ToString();
    return RedirectToAction(nameof(Results));
} else {
    return View("Form");
}
...
```

The Product object is valid if the IsValid property returns true, in which case the action method redirects the browser to the Results action, where the validated form values will be displayed. There is a validation problem if the IsValue property returns false, which is dealt with by calling the View method to render the Form view again.

Displaying Validation Errors to the User

It may seem odd to deal with a validation error by calling the View method, but the context data provided to the view contains details of the model validation errors; these details are used by the tag helper to transform the input elements.

To see how this works, restart ASP.NET Core so the changes to the controller take effect and use a browser to request http://localhost:5000/controllers/form. Clear the contents of the Name field and click the Submit button. There won't be any visible change in the content displayed by the browser, but if you examine the input element for the Name field, you will see the element has been transformed. Here is the input element before the form was submitted:

```
<input class="form-control" type="text" id="Name" name="Name" value="Kayak">
```

Here is the input element after the form has been submitted:

```
<input class="form-control input-validation-error" type="text" id="Name"
    name="Name" value="">
```

The tag helper adds elements whose values have failed validation to the input-validation-error class, which can then be styled to highlight the problem to the user.

You can do this by defining custom CSS styles in a stylesheet, but a little extra work is required if you want to use the built-in validation styles that CSS libraries like Bootstrap provides. The name of the class added to the input elements cannot be changed, which means that some JavaScript code is required to map between the name used by ASP.NET Core and the CSS error classes provided by Bootstrap.

■ **Tip** Using JavaScript code like this can be awkward, and it can be tempting to use custom CSS styles, even when working with a CSS library like Bootstrap. However, the colors used for validation classes in Bootstrap can be overridden by using themes or by customizing the package and defining your own styles, which means you have to ensure that any changes to the theme are matched by corresponding changes to any custom styles you define. Ideally, Microsoft will make the validation class names configurable in a future release of ASP.NET Core, but until then, using JavaScript to apply Bootstrap styles is a more robust approach than creating custom stylesheets.

To define the JavaScript code so that it can be used by both controllers and Razor Pages, use the Visual Studio JavaScript File template to add a file named `_Validation.cshtml` to the `Views/Shared` folder with the content shown in Listing 29-6. Visual Studio Code doesn't require templates, and you can just add a file named `_Validation.cshtml` in the `Views/Shared` folder with the code shown in the listing.

Listing 29-6. The Contents of the _Validation.cshtml File in the Views/Shared Folder

```
<script src="/lib/jquery/jquery.min.js"></script>
<script type="text/javascript">
    $(document).ready(function () {
        $("input.input-validation-error").addClass("is-invalid");
    });
</script>
```

I will use the new file as a partial view, which contains a script element that loads the jQuery library and contains a custom script that locates `input` elements that are members of the `input-validation-error` class and adds them to the `is-invalid` class (which Bootstrap uses to set the error color for form elements). Listing 29-7 uses the `partial` tag helper to incorporate the new partial view into the HTML form so that fields with validation errors are highlighted.

Listing 29-7. Including a Partial View in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<partial name="_Validation" />

<form asp-action="submitform" method="post" id="htmlform">
    <div class="form-group">
        <label asp-for="Name"></label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label asp-for="Price"></label>
        <input class="form-control" asp-for="Price" />
    </div>
    <div class="form-group">
        <label>CategoryId</label>
        <input class="form-control" asp-for="CategoryId"  />
    </div>
    <div class="form-group">
        <label>SupplierId</label>
        <input class="form-control" asp-for="SupplierId"  />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

The jQuery code runs when the browser has finished parsing all the elements in the HTML document, and the effect is to highlight the `input` elements that have been assigned to the `input-validaton-error` class. You can see the effect by navigating to `http://localhost:5000/controllers/form`, clearing the contents of the Name field, and submitting the form, which produces the response shown in Figure 29-2.

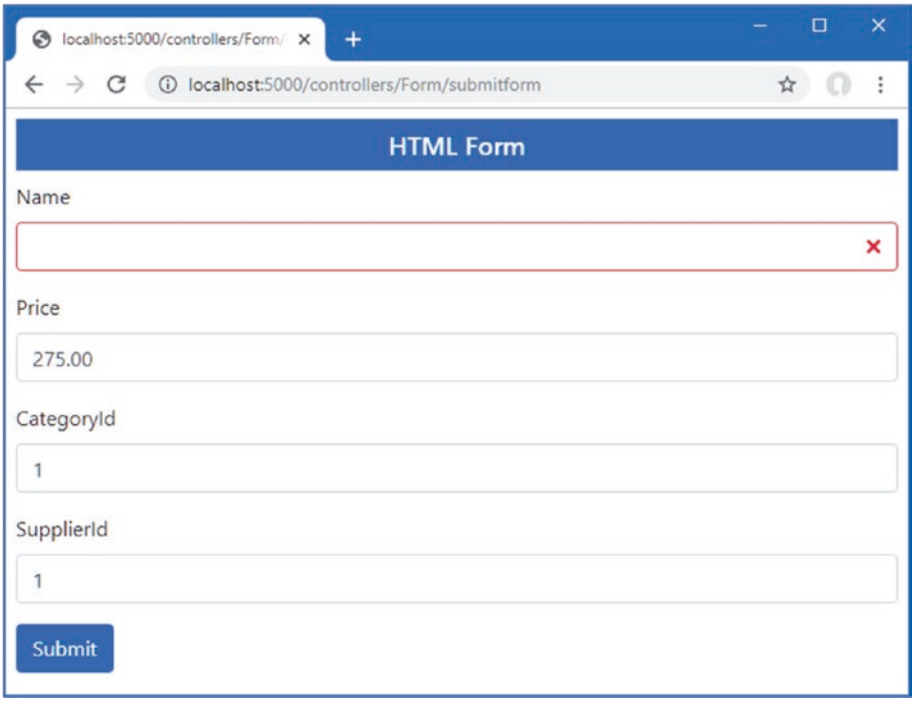

Figure 29-2. *Highlighting a validation error*

The user will not be shown the `Results` view until the form is submitted with data that can be parsed by the model browser and that passes the explicit validation checks in the action method. Until that happens, submitting the form will cause the `Form` view to be rendered with the highlighted validation errors.

Displaying Validation Messages

The CSS classes that the tag helpers apply to `input` elements indicate that there are problems with a form field, but they do not tell the user what the problem is. Providing the user with more information requires the use of a different tag helper, which adds a summary of the problems to the view, as shown in Listing 29-8.

Listing 29-8. Displaying a Summary in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<partial name="_Validation" />

<form asp-action="submitform" method="post" id="htmlform">
    <div asp-validation-summary="All" class="text-danger"></div>
    <div class="form-group">
        <label asp-for="Name"></label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label asp-for="Price"></label>
        <input class="form-control" asp-for="Price" />
    </div>
    <div class="form-group">
        <label>CategoryId</label>
```

```
        <input class="form-control" asp-for="CategoryId"  />
    </div>
    <div class="form-group">
        <label>SupplierId</label>
        <input class="form-control" asp-for="SupplierId"  />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

The `ValidationSummaryTagHelper` class detects the `asp-validation-summary` attribute on `div` elements and responds by adding messages that describe any validation errors that have been recorded. The value of the `asp-validation-summary` attribute is a value from the `ValidationSummary` enumeration, which defines the values shown in Table 29-5 and which I demonstrate shortly.

Table 29-5. *The ValidationSummary Values*

Name	Description
`All`	This value is used to display all the validation errors that have been recorded.
`ModelOnly`	This value is used to display only the validation errors for the entire model, excluding those that have been recorded for individual properties, as described in the "Displaying Model-Level Messages" section.
`None`	This value is used to disable the tag helper so that it does not transform the HTML element.

Presenting error messages helps the user understand why the form cannot be processed. As an example, try submitting the form with a negative value in the Price field, such as **-10**, and with a value that cannot be converted into a `decimal` value, such as **ten**. Each value results in a different error message, as shown in Figure 29-3.

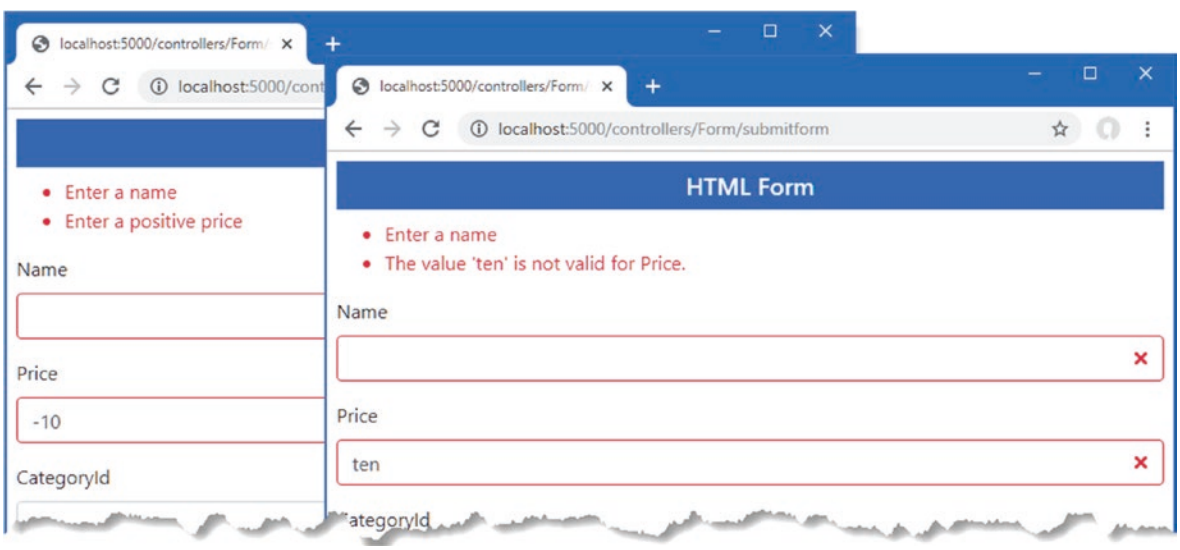

Figure 29-3. *Displaying validation messages*

Configuring the Default Validation Error Messages

The model binding process performs its own validation when it tries to provide the data values required to invoke an action method, which is why you see a validation message when the `Price` value cannot be converted to a `decimal`, for example. Not all the validation messages produced by the model binder are helpful to the user, which you can see by clearing the Price field and submitting the form. The empty field produces the following message:

```
The value '' is invalid
```

This message is added to the ModelStateDictionary by the model binding process when it can't find a value for a property or does find a value but can't parse it. In this case, the error has arisen because the empty string sent in the form data can't be parsed into a decimal value for the Price property of the Product class.

The model binder has a set of predefined messages that it uses for validation errors. These can be replaced with custom messages using the methods defined by the DefaultModelBindingMessageProvider class, as described in Table 29-6.

Table 29-6. *The DefaultModelBindingMessageProvider Methods*

Name	Description
SetValueMustNotBeNullAccessor	The function assigned to this property is used to generate a validation error message when a value is null for a model property that is non-nullable.
SetMissingBindRequiredValueAccessor	The function assigned to this property is used to generate a validation error message when the request does not contain a value for a required property.
SetMissingKeyOrValueAccessor	The function assigned to this property is used to generate a validation error message when the data required for dictionary model object contains null keys or values.
SetAttemptedValueIsInvalidAccessor	The function assigned to this property is used to generate a validation error message when the model binding system cannot convert the data value into the required C# type.
SetUnknownValueIsInvalidAccessor	The function assigned to this property is used to generate a validation error message when the model binding system cannot convert the data value into the required C# type.
SetValueMustBeANumberAccessor	The function assigned to this property is used to generate a validation error message when the data value cannot be parsed into a C# numeric type.
SetValueIsInvalidAccessor	The function assigned to this property is used to generate a fallback validation error message that is used as a last resort.

Each of the methods described in the table accepts a function that is invoked to get the validation message to display to the user. These methods are applied through the options pattern in the Startup class, as shown in Listing 29-9, in which I have replaced the default message that is displayed when a value is null or cannot be converted.

Listing 29-9. Changing a Validation Message in the Startup.cs File in the WebApp Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using WebApp.Models;
using Microsoft.AspNetCore.Antiforgery;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;

namespace WebApp {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        public IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<DataContext>(opts => {
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:ProductConnection"]);
                opts.EnableSensitiveDataLogging(true);
            });
```

```
            services.AddControllersWithViews().AddRazorRuntimeCompilation();
            services.AddRazorPages().AddRazorRuntimeCompilation();
            services.AddSingleton<CitiesData>();

            services.Configure<AntiforgeryOptions>(opts => {
                opts.HeaderName = "X-XSRF-TOKEN";
            });

            services.Configure<MvcOptions>(opts => opts.ModelBindingMessageProvider
                .SetValueMustNotBeNullAccessor(value => "Please enter a value"));
        }

        public void Configure(IApplicationBuilder app, DataContext context,
                IAntiforgery antiforgery) {

            app.UseRequestLocalization();

            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseRouting();

            app.Use(async (context, next) => {
                if (!context.Request.Path.StartsWithSegments("/api")) {
                    context.Response.Cookies.Append("XSRF-TOKEN",
                       antiforgery.GetAndStoreTokens(context).RequestToken,
                       new CookieOptions { HttpOnly = false });
                }
                await next();
            });

            app.UseEndpoints(endpoints => {
                endpoints.MapControllers();
                endpoints.MapControllerRoute("forms",
                    "controllers/{controller=Home}/{action=Index}/{id?}");
                endpoints.MapDefaultControllerRoute();
                endpoints.MapRazorPages();
            });
            SeedData.SeedDatabase(context);
        }
    }
}
```

The function that you specify receives the value that the user has supplied, although that is not especially useful when dealing with `null` values. To see the custom message, restart ASP.NET Core, use the browser to request `http://localhost:5000/controllers/form`, and submit the form with an empty Price field. The response will include the custom error message, as shown in Figure 29-4.

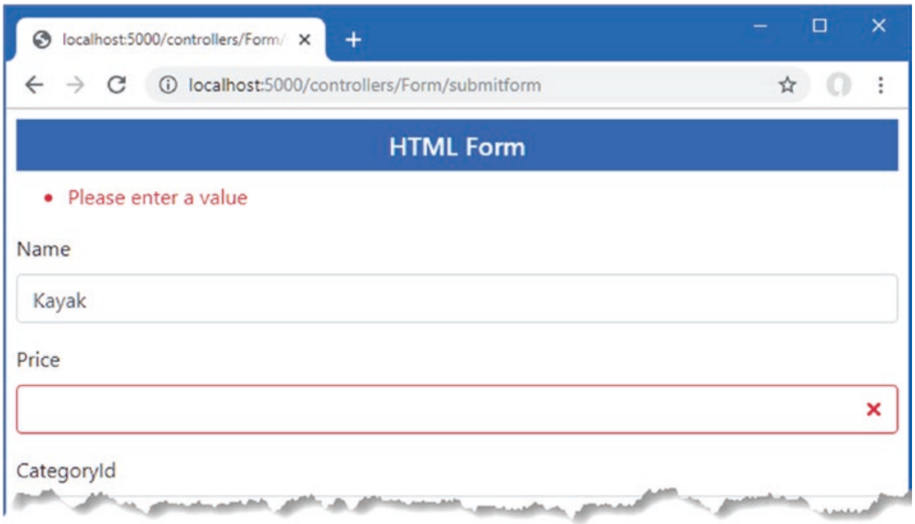

Figure 29-4. *Changing the default validation messages*

Displaying Property-Level Validation Messages

Although the custom error message is more meaningful than the default one, it still isn't that helpful because it doesn't clearly indicate which field the problem relates to. For this kind of error, it is more useful to display the validation error messages alongside the HTML elements that contain the problem data. This can be done using the `ValidationMessageTag` tag helper, which looks for `span` elements that have the `asp-validation-for` attribute, which is used to specify the property for which error messages should be displayed.

In Listing 29-10, I have added property-level validation message elements for each of the `input` elements in the form.

Listing 29-10. Adding Property-Level Messages in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<partial name="_Validation" />

<form asp-action="submitform" method="post" id="htmlform">
    <div asp-validation-summary="All" class="text-danger"></div>
    <div class="form-group">
        <label asp-for="Name"></label>
        <div><span asp-validation-for="Name" class="text-danger"></span></div>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label asp-for="Price"></label>
        <div><span asp-validation-for="Price" class="text-danger"></span></div>
        <input class="form-control" asp-for="Price" />
    </div>
    <div class="form-group">
        <label>CategoryId</label>
        <div><span asp-validation-for="CategoryId" class="text-danger"></span></div>
        <input class="form-control" asp-for="CategoryId"  />
    </div>
    <div class="form-group">
        <label>SupplierId</label>
```

```
        <div><span asp-validation-for="SupplierId" class="text-danger"></span></div>
        <input class="form-control" asp-for="SupplierId"  />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

Since span elements are displayed inline, care must be taken to present the validation messages to make it obvious which element the message relates to. You can see the effect of the new validation messages by requesting `http://localhost:5000/controllers/form`, clearing the Name and Price fields, and submitting the form. The response, shown in Figure 29-5, includes validation messages alongside the text fields.

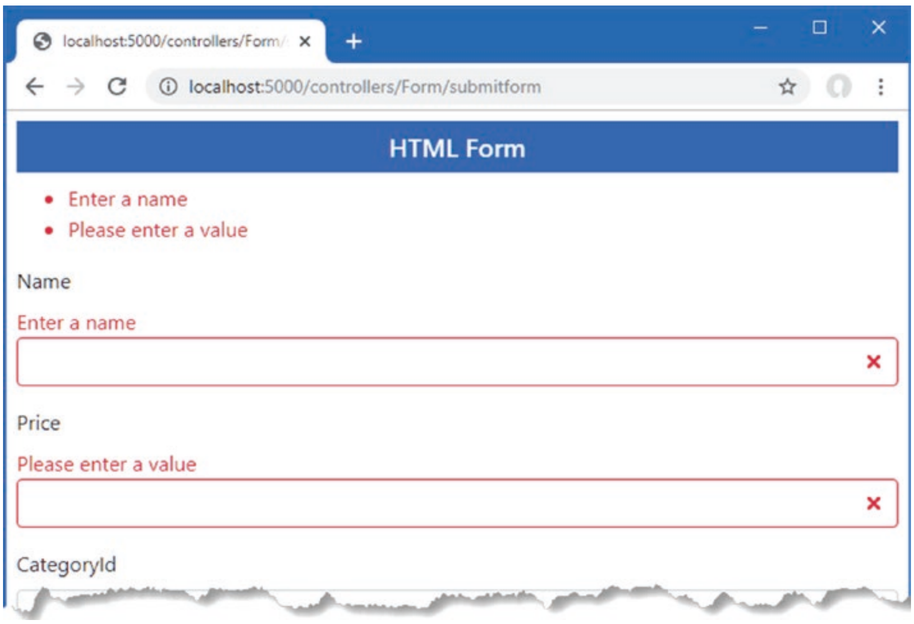

Figure 29-5. *Displaying property-level validation messages*

Displaying Model-Level Messages

It may seem that the validation summary message is superfluous because it duplicates the property-level messages. But the summary has a useful trick, which is the ability to display messages that apply to the entire model and not just individual properties. This means you can report errors that arise from a combination of individual properties, which would otherwise be hard to express with a property-level message.

In Listing 29-11, I have added a check to the `FormController.SubmitForm` action that records a validation error when the `Price` value exceeds 100 at the time that the `Name` value starts with `Small`.

Listing 29-11. Performing Model-Level Validation in the FormController.cs File in the Controllers Folder

```
...
[HttpPost]
public IActionResult SubmitForm(Product product) {

    if (string.IsNullOrEmpty(product.Name)) {
        ModelState.AddModelError(nameof(Product.Name), "Enter a name");
    }

    if (ModelState.GetValidationState(nameof(Product.Price))
            == ModelValidationState.Valid && product.Price < 1) {
        ModelState.AddModelError(nameof(Product.Price), "Enter a positive price");
    }
```

```
    if (ModelState.GetValidationState(nameof(Product.Name))
            == ModelValidationState.Valid
            && ModelState.GetValidationState(nameof(Product.Price))
            == ModelValidationState.Valid
            && product.Name.ToLower().StartsWith("small") && product.Price > 100) {
        ModelState.AddModelError("", "Small products cannot cost more than $100");
    }

    if (!context.Categories.Any(c => c.CategoryId == product.CategoryId)) {
        ModelState.AddModelError(nameof(Product.CategoryId),
            "Enter an existing category ID");
    }

    if (!context.Suppliers.Any(s => s.SupplierId == product.SupplierId)) {
        ModelState.AddModelError(nameof(Product.SupplierId),
            "Enter an existing supplier ID");
    }

    if (ModelState.IsValid) {
        TempData["name"] = product.Name;
        TempData["price"] = product.Price.ToString();
        TempData["categoryId"] = product.CategoryId.ToString();
        TempData["supplierId"] = product.SupplierId.ToString();
        return RedirectToAction(nameof(Results));
    } else {
        return View("Form");
    }
}
...
```

If the user enters a `Name` value that starts with `Small` and a `Price` value that is greater than `100`, then a model-level validation error is recorded. I check for the combination of values only if there are no validation problems with the individual property values, which ensures the user doesn't get conflicting messages. Validation errors that relate to the entire model are recorded using the `AddModelError` with the empty string as the first argument.

Listing 29-12 changes the value of the `asp-validation-summary` attribute to `ModelOnly`, which excludes property-level errors, meaning that the summary will display only those errors that apply to the entire model.

Listing 29-12. Configuring the Validation Summary in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<partial name="_Validation" />

<form asp-action="submitform" method="post" id="htmlform">
    <div asp-validation-summary="ModelOnly" class="text-danger"></div>
    <div class="form-group">
        <label asp-for="Name"></label>
        <div><span asp-validation-for="Name" class="text-danger"></span></div>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label asp-for="Price"></label>
        <div><span asp-validation-for="Price" class="text-danger"></span></div>
        <input class="form-control" asp-for="Price" />
    </div>
```

```
    <div class="form-group">
        <label>CategoryId</label>
        <div><span asp-validation-for="CategoryId" class="text-danger"></span></div>
        <input class="form-control" asp-for="CategoryId"  />
    </div>
    <div class="form-group">
        <label>SupplierId</label>
        <div><span asp-validation-for="SupplierId" class="text-danger"></span></div>
        <input class="form-control" asp-for="SupplierId"  />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

Restart ASP.NET Core and request `http://localhost:5000/controllers/form`. Enter **Small Kayak** into the Name field and **150** into the Price field and submit the form. The response will include the model-level error message, as shown in Figure 29-6.

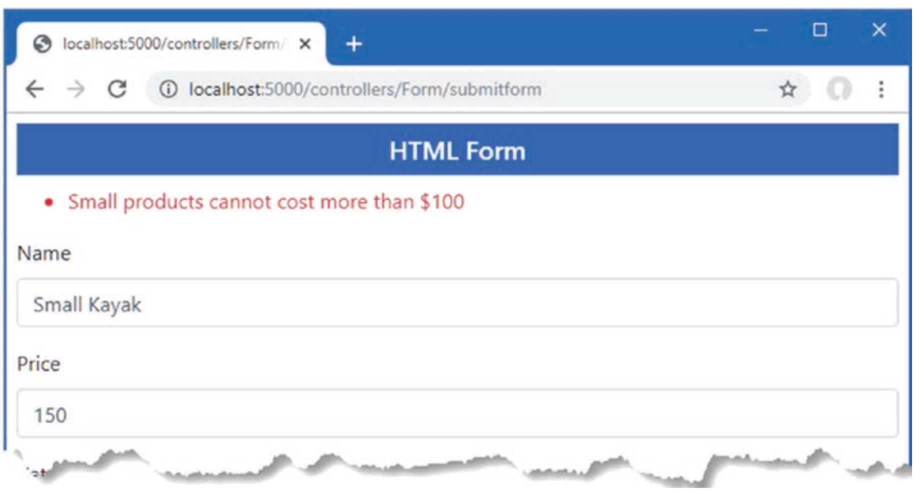

Figure 29-6. *Displaying a model-level validation message*

Explicitly Validating Data in a Razor Page

Razor Page validation relies on the features used in the controller in the previous section. Listing 29-13 adds explicit validation checks and error summaries to the `FormHandler` page.

Listing 29-13. Validating Data in the FormHandler.cshtml File in the Pages Folder

```
@page "/pages/form/{id:long?}"
@model FormHandlerModel
@using Microsoft.AspNetCore.Mvc.RazorPages
@using Microsoft.EntityFrameworkCore
@using Microsoft.AspNetCore.Mvc.ModelBinding

<partial name="_Validation" />

<div class="m-2">
    <h5 class="bg-primary text-white text-center p-2">HTML Form</h5>
    <form asp-page="FormHandler" method="post" id="htmlform">
        <div asp-validation-summary="ModelOnly" class="text-danger"></div>
        <div class="form-group">
            <label>Name</label>
            <div>
```

```
                <span asp-validation-for="Product.Name" class="text-danger">
                </span>
            </div>
            <input class="form-control" asp-for="Product.Name" />
        </div>
        <div class="form-group">
            <label>Price</label>
            <div>
                 <span asp-validation-for="Product.Price" class="text-danger">
                 </span>
            </div>
            <input class="form-control" asp-for="Product.Price"  />
        </div>
        <div class="form-group">
            <label>CategoryId</label>
            <div>
                <span asp-validation-for="Product.CategoryId" class="text-danger">
               </span>
           </div>
            <input class="form-control" asp-for="Product.CategoryId"  />
        </div>
        <div class="form-group">
            <label>SupplierId</label>
            <div>
                <span asp-validation-for="Product.SupplierId" class="text-danger">
                </span>
            </div>
            <input class="form-control" asp-for="Product.SupplierId"  />
        </div>
        <button type="submit" class="btn btn-primary">Submit</button>
    </form>
</div>

@functions {

    public class FormHandlerModel : PageModel {
        private DataContext context;

        public FormHandlerModel(DataContext dbContext) {
            context = dbContext;
        }

        [BindProperty]
        public Product Product { get; set; }

        //[BindProperty(Name = "Product.Category")]
        //public Category Category { get; set; }

        public async Task OnGetAsync(long id = 1) {
            Product = await context.Products.FirstAsync(p => p.ProductId == id);
        }

        public IActionResult OnPost() {

            if (string.IsNullOrEmpty(Product.Name)) {
                ModelState.AddModelError("Product.Name", "Enter a name");
            }
```

```
            if (ModelState.GetValidationState("Product.Price")
                    == ModelValidationState.Valid && Product.Price < 1) {
                ModelState.AddModelError("Product.Price", "Enter a positive price");
            }

            if (ModelState.GetValidationState("Product.Name")
                    == ModelValidationState.Valid
                && ModelState.GetValidationState("Product.Price")
                    == ModelValidationState.Valid
                && Product.Name.ToLower().StartsWith("small")
                && Product.Price > 100) {
                    ModelState.AddModelError("",
                        "Small products cannot cost more than $100");
            }

            if (!context.Categories.Any(c => c.CategoryId == Product.CategoryId)) {
                ModelState.AddModelError("Product.CategoryId",
                    "Enter an existing category ID");
            }

            if (!context.Suppliers.Any(s => s.SupplierId == Product.SupplierId)) {
                    ModelState.AddModelError("Product.SupplierId",
                        "Enter an existing supplier ID");
            }

            if (ModelState.IsValid) {
                TempData["name"] = Product.Name;
                TempData["price"] = Product.Price.ToString();
                TempData["categoryId"] = Product.CategoryId.ToString();
                TempData["supplierId"] = Product.SupplierId.ToString();
                return RedirectToPage("FormResults");
            } else {
                return Page();
            }
        }
    }
}
```

The PageModel class defines a ModelState property that is the equivalent of the one I used in the controller and allows validation errors to be recorded. The process for validation is the same, but you must take care when recording errors to ensure the names match the pattern used by Razor Pages. When I recorded an error, I used the nameof keyword to select the property to which the error relates, like this:

```
...
ModelState.AddModelError(nameof(Product.Name), "Enter a name");
...
```

This is a common convention because it ensures that a typo won't cause errors to be recorded incorrectly. This expression won't work in the Razor Page, where the error must be recorded against Product.Name, rather than Name, to reflect that @Model expressions in Razor Pages return the page model object, like this:

```
...
ModelState.AddModelError("Product.Name", "Enter a name");
...
```

To test the validation process, use a browser to request `http://localhost:5000/pages/form` and submit the form with empty fields or with values that cannot be converted into the C# types required by the `Product` class. The error messages are displayed just as they are for controllers, as shown in Figure 29-7. (The values `1`, `2`, and `3` are valid for both the `CategoryId` and `SupplierId` fields.)

■ **Tip** The methods described in Table 29-6 that change the default validation messages affect Razor Pages as well as controllers.

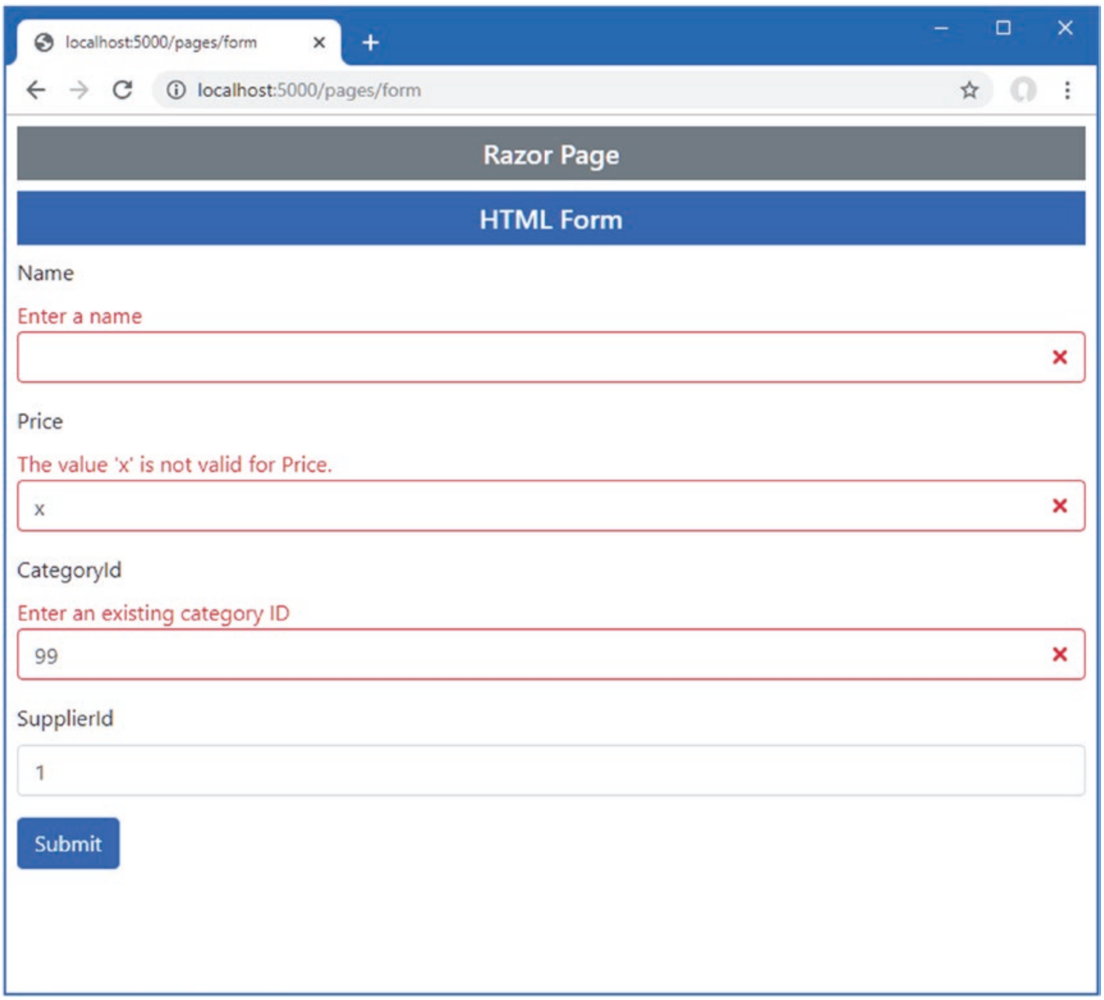

Figure 29-7. *Validating data in a Razor Page*

Specifying Validation Rules Using Metadata

One problem with putting validation logic into an action method is that it ends up being duplicated in every action or handler method that receives data from the user. To help reduce duplication, the validation process supports the use of attributes to express model validation rules directly in the model class, ensuring that the same set of validation rules will be applied regardless of which action method is used to process a request. In Listing 29-14, I have applied attributes to the `Product` class to describe the validation required for the `Name` and `Price` properties.

Listing 29-14. Applying Validation Attributes in the Product.cs File in the Models Folder

```
using System.ComponentModel.DataAnnotations.Schema;
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace WebApp.Models {
    public class Product {

        public long ProductId { get; set; }

        [Required]
        [Display(Name = "Name")]
        public string Name { get; set; }

        [Column(TypeName = "decimal(8, 2)")]
        [Required(ErrorMessage = "Please enter a price")]
        [Range(1, 999999, ErrorMessage = "Please enter a positive price")]
        public decimal Price { get; set; }

        public long CategoryId { get; set; }
        public Category Category { get; set; }

        public long SupplierId { get; set; }
        public Supplier Supplier { get; set; }
    }
}
```

I used two validation attributes in the listing: Required and Range. The Required attribute specifies that it is a validation error if the user doesn't submit a value for a property. The Range attribute specifies a subset of acceptable values. Table 29-7 shows the set of built-in validation attributes available.

Table 29-7. *The Built-in Validation Attributes*

Attribute	Example	Description
Compare	[Compare ("OtherProperty")]	This attribute ensures that properties must have the same value, which is useful when you ask the user to provide the same information twice, such as an e-mail address or a password.
Range	[Range(10, 20)]	This attribute ensures that a numeric value (or any property type that implements IComparable) is not outside the range of specified minimum and maximum values. To specify a boundary on only one side, use a MinValue or MaxValue constant.
RegularExpression	[RegularExpression ("pattern")]	This attribute ensures that a string value matches the specified regular expression pattern. Note that the pattern must match the *entire* user-supplied value, not just a substring within it. By default, it matches case sensitively, but you can make it case insensitive by applying the (?i) modifier—that is, [RegularExpression("(?i)mypattern")].
Required	[Required]	This attribute ensures that the value is not empty or a string consisting only of spaces. If you want to treat whitespace as valid, use [Required(AllowEmptyStrings = true)].
StringLength	[StringLength(10)]	This attribute ensures that a string value is no longer than a specified maximum length. You can also specify a minimum length: [StringLength(10, MinimumLength=2)].

All the validation attributes support specifying a custom error message by setting a value for the ErrorMessage property, like this:

```
...
[Column(TypeName = "decimal(8, 2)")]
[Required(ErrorMessage = "Please enter a price")]
[Range(1, 999999, ErrorMessage = "Please enter a positive price")]
public decimal Price { get; set; }
...
```

If there is no custom error message, then the default messages will be used, but they tend to reveal details of the model class that will make no sense to the user unless you also use the Display attribute, like this:

```
...
[Required]
[Display(Name = "Name")]
public string Name { get; set; }
...
```

The default message generated by the Required attribute reflects the name specified with the Display attribute and so doesn't reveal the name of the property to the user.

VALIDATION WORK AROUNDS

Getting the validation results you require can take some care when using the validation attributes. For example, you cannot use the Required attribute if you want to ensure that a user has checked a checkbox because the browser will send a false value when the checkbox is unchecked, which will always pass the checks applied by the Required attribute. Instead, use the Range attribute and specify the minimum and maximum values as true, like this:

```
...
[Range(typeof(bool), "true", "true", ErrorMessage="You must check the box")]
...
```

If this sort of workaround feels uncomfortable, then you can create custom validation attributes, as described in the next section.

The use of the validation attributes on the Product class allows me to remove the explicit validation checks for the Name and Price properties, as shown in Listing 29-15.

Listing 29-15. Removing Explicit Validation in the FormController.cs File in the Controllers Folder

```
...
[HttpPost]
public IActionResult SubmitForm(Product product) {

    //if (string.IsNullOrEmpty(product.Name)) {
    //    ModelState.AddModelError(nameof(Product.Name), "Enter a name");
    //}

    //if (ModelState.GetValidationState(nameof(Product.Price))
    //        == ModelValidationState.Valid && product.Price < 1) {
    //    ModelState.AddModelError(nameof(Product.Price), "Enter a positive price");
    //}

    if (ModelState.GetValidationState(nameof(Product.Name))
            == ModelValidationState.Valid
            && ModelState.GetValidationState(nameof(Product.Price))
            == ModelValidationState.Valid
```

```
            && product.Name.ToLower().StartsWith("small") && product.Price > 100) {
        ModelState.AddModelError("", "Small products cannot cost more than $100");
    }

    if (!context.Categories.Any(c => c.CategoryId == product.CategoryId)) {
        ModelState.AddModelError(nameof(Product.CategoryId),
            "Enter an existing category ID");
    }

    if (!context.Suppliers.Any(s => s.SupplierId == product.SupplierId)) {
        ModelState.AddModelError(nameof(Product.SupplierId),
            "Enter an existing supplier ID");
    }

    if (ModelState.IsValid) {
        TempData["name"] = product.Name;
        TempData["price"] = product.Price.ToString();
        TempData["categoryId"] = product.CategoryId.ToString();
        TempData["supplierId"] = product.SupplierId.ToString();
        return RedirectToAction(nameof(Results));
    } else {
        return View("Form");
    }
}
...
```

The validation attributes are applied before the action method is called, which means that I can still rely on the model state to determine whether individual properties are valid when performing model-level validation. To see the validation attributes in action, restart ASP.NET Core MVC, request `http://localhost:5000/controllers/form`, clear the Name and Price fields, and submit the form. The response will include the validation errors produced by the attributes, as shown in Figure 29-8.

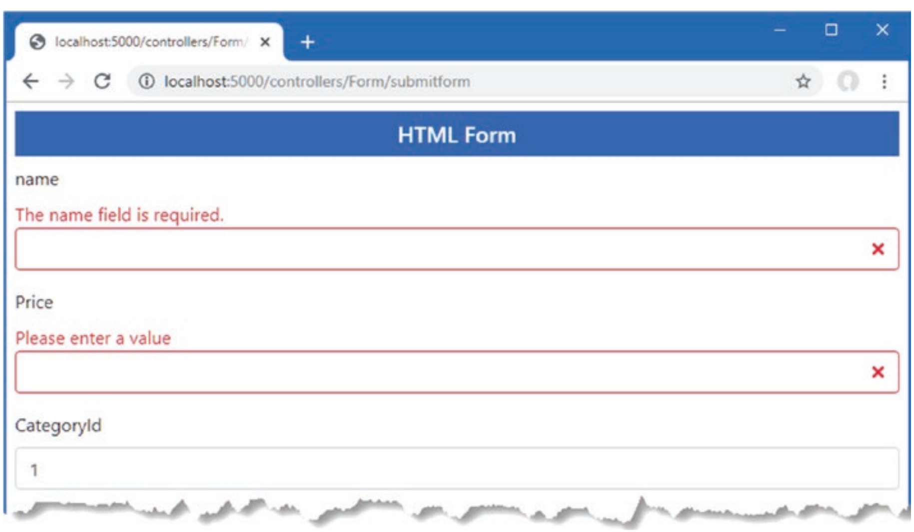

Figure 29-8. *Using validation attributes*

UNDERSTANDING WEB SERVICE CONTROLLER VALIDATION

Controllers that have been decorated with the `ApiController` attribute do not need to check the `ModelState.IsValid` property. Instead, the action method is invoked only if there are no validation errors, which means you can always rely on receiving validated objects through the model binding feature. If any validation errors are detected, then the request is terminated, and an error response is sent to the browser.

Creating a Custom Property Validation Attribute

The validation process can be extended by creating an attribute that extends the `ValidationAttribute` class. To demonstrate, I created the `WebApp/Validation` folder and added to it a class file named `PrimaryKeyAttribute.cs`, which I used to define the class shown in Listing 29-16.

Listing 29-16. The Contents of the PrimaryKeyAttribute.cs File in the Validation Folder

```
using Microsoft.EntityFrameworkCore;
using System;
using System.ComponentModel.DataAnnotations;

namespace WebApp.Validation {
    public class PrimaryKeyAttribute : ValidationAttribute {

        public Type ContextType { get; set; }

        public Type DataType { get; set; }

        protected override ValidationResult IsValid(object value,
                ValidationContext validationContext) {
            DbContext context
                = validationContext.GetService(ContextType) as DbContext;
            if (context.Find(DataType, value) == null) {
                return new ValidationResult(ErrorMessage
                    ?? "Enter an existing key value");
            } else {
                return ValidationResult.Success;
            }
        }
    }
}
```

Custom attributes override the `IsValid` method, which is called with the value to check, and a `ValidationContext` object that provides context about the validation process and provides access to the application's services through its `GetService` method.

In Listing 29-16, the custom attribute receives the type of an Entity Framework Core database context class and the type of a model class. In the `IsValid` method, the attribute obtains an instance of the context class and uses it to query the database to determine whether the value has been used as a primary key value.

REVALIDATING DATA

You may need to perform the validation process again if you modify the object received from the model binder. For these situations, use the `ModelState.Clear` method to clear any existing validation errors and call the `TryValidateModel` method.

Custom validation attributes can also be used to perform model-level validation. To demonstrate, I added a class file named `PhraseAndPriceAttribute.cs` to the `Validation` folder and used it to define the class shown in Listing 29-17.

Listing 29-17. The Contents of the PhraseAndPriceAttribute.cs File in the Validation Folder

```
using System;
using System.ComponentModel.DataAnnotations;
using WebApp.Models;

namespace WebApp.Validation {
    public class PhraseAndPriceAttribute: ValidationAttribute {

        public string Phrase { get; set; }

        public string Price { get; set; }

        protected override ValidationResult IsValid(object value,
                ValidationContext validationContext) {
            Product product = value as Product;
            if (product != null
                && product.Name.StartsWith(Phrase,
                    StringComparison.OrdinalIgnoreCase)
                && product.Price > decimal.Parse(Price)) {
                    return new ValidationResult(ErrorMessage
                        ?? $"{Phrase} products cannot cost more than ${Price}");
            }
            return ValidationResult.Success;
        }
    }
}
```

This attribute is configured with `Phrase` and `Price` properties, which are used in the `IsValid` method to check the `Name` and `Price` properties of the model object. Property-level custom validation attributes are applied directly to the properties they validate, and model-level attributes are applied to the entire class, as shown in Listing 29-18.

Listing 29-18. Applying Custom Validation Attributes in the Product.cs File in the Models Folder

```
using System.ComponentModel.DataAnnotations.Schema;
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using WebApp.Validation;

namespace WebApp.Models {

    [PhraseAndPrice(Phrase ="Small", Price = "100")]
    public class Product {

        public long ProductId { get; set; }

        [Required]
        [Display(Name = "Name")]
        public string Name { get; set; }

        [Column(TypeName = "decimal(8, 2)")]
        [Required(ErrorMessage = "Please enter a price")]
        [Range(1, 999999, ErrorMessage = "Please enter a positive price")]
        public decimal Price { get; set; }

        [PrimaryKey(ContextType= typeof(DataContext), DataType = typeof(Category))]
        public long CategoryId { get; set; }
        public Category Category { get; set; }
```

```
        [PrimaryKey(ContextType = typeof(DataContext), DataType = typeof(Category))]
        public long SupplierId { get; set; }
        public Supplier Supplier { get; set; }
    }
}
```

The custom attributes allow the remaining explicit validation statements to be removed from the `Form` controller's action method, as shown in Listing 29-19.

Listing 29-19. Removing Explicit Validation in the FormController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using System.Threading.Tasks;
using WebApp.Models;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace WebApp.Controllers {

    [AutoValidateAntiforgeryToken]
    public class FormController : Controller {
        private DataContext context;

        public FormController(DataContext dbContext) {
            context = dbContext;
        }

        public async Task<IActionResult> Index(long? id) {
            return View("Form", await context.Products
                .FirstOrDefaultAsync(p => id == null || p.ProductId == id));
        }

        [HttpPost]
        public IActionResult SubmitForm(Product product) {
            if (ModelState.IsValid) {
                TempData["name"] = product.Name;
                TempData["price"] = product.Price.ToString();
                TempData["categoryId"] = product.CategoryId.ToString();
                TempData["supplierId"] = product.SupplierId.ToString();
                return RedirectToAction(nameof(Results));
            } else {
                return View("Form");
            }
        }

        public IActionResult Results() {
            return View(TempData);
        }
    }
}
```

The validation attributes are applied automatically before the action method is invoked, which means that the validation outcome can be determined simply by reading the `ModelState.IsValid` property. The same simplification can be applied to the Razor Page, as shown in Listing 29-20.

Listing 29-20. Removing Explicit Validation in the FormHandler.cshtml File in the Pages Folder

```
...
@functions {

    public class FormHandlerModel : PageModel {
        private DataContext context;

        public FormHandlerModel(DataContext dbContext) {
            context = dbContext;
        }

        [BindProperty]
        public Product Product { get; set; }

        public async Task OnGetAsync(long id = 1) {
            Product = await context.Products.FirstAsync(p => p.ProductId == id);
        }

        public IActionResult OnPost() {
            if (ModelState.IsValid) {
                TempData["name"] = Product.Name;
                TempData["price"] = Product.Price.ToString();
                TempData["categoryId"] = Product.CategoryId.ToString();
                TempData["supplierId"] = Product.SupplierId.ToString();
                return RedirectToPage("FormResults");
            } else {
                return Page();
            }
        }
    }
}
...
```

Expressing the validation through the custom attributes removes the code duplication between the controller and the Razor Page and ensures that validation is applied consistently wherever model binding is used for `Product` objects. To test the validation attributes, restart ASP.NET Core and navigate to `http://localhost:5000/controllers/form` or `http://localhost:5000/pages/form`. Clear the form fields or enter bad key values and submit the form, and you will see the error messages produced by the attributes, some of which are shown in Figure 29-9. (The values `1`, `2`, and `3` are valid for both the `CategoryId` and `SupplierId` fields.)

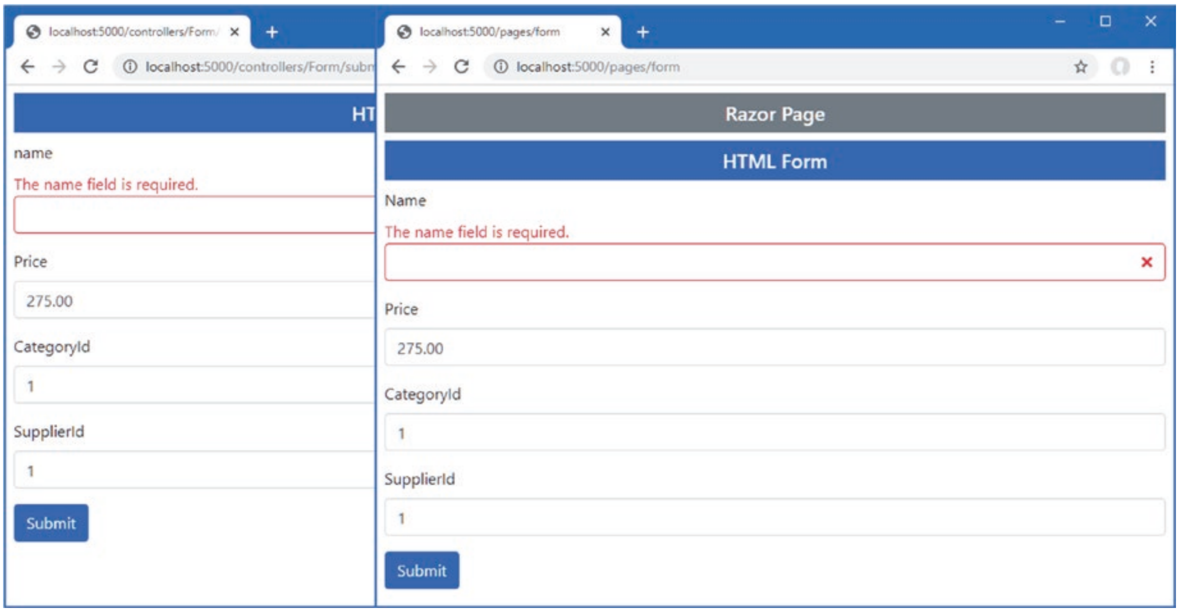

Figure 29-9. *Using custom validation attributes*

Performing Client-Side Validation

The validation techniques I have demonstrated so far have all been examples of *server-side validation*. This means the user submits their data to the server, and the server validates the data and sends back the results of the validation (either success in processing the data or a list of errors that need to be corrected).

In web applications, users typically expect immediate validation feedback—without having to submit anything to the server. This is known as *client-side validation* and is implemented using JavaScript. The data that the user has entered is validated before being sent to the server, providing the user with immediate feedback and an opportunity to correct any problems.

ASP.NET Core supports *unobtrusive client-side validation*. The term *unobtrusive* means that validation rules are expressed using attributes added to the HTML elements that views generate. These attributes are interpreted by a JavaScript library distributed by Microsoft that, in turn, configures the jQuery Validation library, which does the actual validation work. In the following sections, I will show you how the built-in validation support works and demonstrate how I can extend the functionality to provide custom client-side validation.

The first step is to install the JavaScript packages that deal with validation. Open a new PowerShell command prompt, navigate to the `WebApp` project folder, and run the command shown in Listing 29-21.

■ **Tip** The core jQuery command was added to the project in Chapter 26. Run the following command if you need to install it again: `libman install jquery@3.4.1 -d wwwroot/lib/jquery`.

Listing 29-21. Installing the Validation Packages

```
libman install jquery-validate@1.19.1 -d wwwroot/lib/jquery-validate
libman install jquery-validation-unobtrusive@3.2.11 -d wwwroot/lib/jquery-validation-unobtrusive
```

Once the packages are installed, add the elements shown in Listing 29-22 to the `_Validation.cshtml` file in the `Views/Shared` folder, which provides a convenient way to introduce the validation alongside the existing jQuery code in the application.

■ **Tip** The elements must be defined in the order in which they are shown.

Listing 29-22. Adding Elements in the _Validation.cshtml File in the Views/Shared Folder

```
<script src="/lib/jquery/jquery.min.js"></script>
<script src="~/lib/jquery-validate/jquery.validate.min.js"></script>
<script
    src="~/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.min.js">
</script>
<script type="text/javascript">
    $(document).ready(function () {
        $("input.input-validation-error").addClass("is-invalid");
    });
</script>
```

The tag helpers add `data-val*` attributes to `input` elements that describe validation constraints for fields. Here are the attributes added to the `input` element for the `Name` field, for example:

```
...
<input class="form-control valid" type="text" data-val="true" data-val-required="The name field is required."
id="Name" name="Name" value="Kayak" aria-describedby="Name-error" aria-invalid="false">
...
```

The unobtrusive validation JavaScript code looks for these attributes and performs validation in the browser when the user attempts to submit the form. The form won't be submitted, and an error will be displayed if there are validation problems. The data won't be sent to the application until there are no outstanding validation issues.

The JavaScript code looks for elements with the `data-val` attribute and performs local validation in the browser when the user submits the form, without sending an HTTP request to the server. You can see the effect by running the application and submitting the form while using the F12 tools to note that validation error messages are displayed even though no HTTP request is sent to the server.

AVOIDING CONFLICTS WITH BROWSER VALIDATION

Some of the current generation of HTML5 browsers support simple client-side validation based on the attributes applied to `input` elements. The general idea is that, say, an `input` element to which the `required` attribute has been applied, for example, will cause the browser to display a validation error when the user tries to submit the form without providing a value.

If you are generating form elements using tag helpers, as I have been doing in this chapter, then you won't have any problems with browser validation because the elements that are assigned `data` attributes are ignored by the browser.

However, you may run into problems if you are unable to completely control the markup in your application, something that often happens when you are passing on content generated elsewhere. The result is that the jQuery validation and the browser validation can both operate on the form, which is just confusing to the user. To avoid this problem, you can add the `novalidate` attribute to the `form` element to disable browser validation.

One of the nice client-side validation features is that the same attributes that specify validation rules are applied at the client *and* at the server. This means that data from browsers that do not support JavaScript are subject to the same validation as those that do, without requiring any additional effort.

To test the client-side validation feature, request `http://localhost:5000/controllers/form` or `http://localhost:5000/pages/form`, clear the Name field, and click the Submit button.

The error message looks like the ones generated by server-side validation, but if you enter text into the field, you will see the error message disappear immediately as the JavaScript code responds to the user interaction, as shown in Figure 29-10.

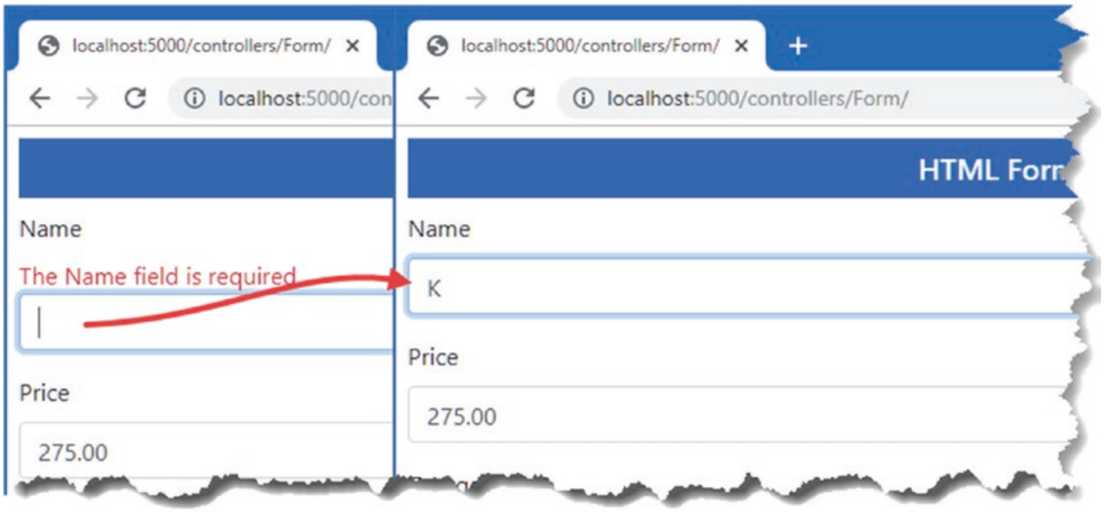

***Figure 29-10.** Performing client-side validation*

EXTENDING CLIENT-SIDE VALIDATION

The client-side validation feature supports the built-in property-level attributes. The feature can be extended but requires fluency in JavaScript and requires working directly with the jQuery Validation package. See `https://jqueryvalidation.org/documentation` for details.

If you don't want to start writing JavaScript code, then you can follow the common pattern of using client-side validation for the built-in validation checks and server-side validation for custom validation.

Performing Remote Validation

Remote validation blurs the line between client- and server-side validation: the validation checks are enforced by the client-side JavaScript code, but the validation checking is performed by sending an asynchronous HTTP request to the application to test the value entered into the form by the user.

A common example of remote validation is to check whether a username is available in applications when such names must be unique, the user submits the data, and the client-side validation is performed. As part of this process, an asynchronous HTTP request is made to the server to validate the username that has been requested. If the username has been taken, a validation error is displayed so that the user can enter another value.

This may seem like regular server-side validation, but there are some benefits to this approach. First, only some properties will be remotely validated; the client-side validation benefits still apply to all the other data values that the user has entered. Second, the request is relatively lightweight and is focused on validation, rather than processing an entire model object.

The third difference is that the remote validation is performed in the background. The user doesn't have to click the submit button and then wait for a new view to be rendered and returned. It makes for a more responsive user experience, especially when there is a slow network between the browser and the server.

That said, remote validation is a compromise. It strikes a balance between client-side and server-side validation, but it does require requests to the application server, and it is not as quick to validate as normal client-side validation.

For the example application, I am going to use remote validation to ensure the user enters existing key values for the `CategoryId` and `SupplierId` properties. The first step is to create a web service controller whose action methods will perform the validation checks. I added a class file named `ValidationController.cs` to the `Controllers` folder with the code shown in Listing 29-23.

Listing 29-23. The Contents of the ValidationController.cs File in the Controllers Folder

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

namespace WebApp.Controllers {

    [ApiController]
    [Route("api/[controller]")]
    public class ValidationController: ControllerBase {
        private DataContext dataContext;

        public ValidationController(DataContext context) {
            dataContext = context;
        }

        [HttpGet("categorykey")]
        public bool CategoryKey(string categoryId) {
            long keyVal;
            return long.TryParse(categoryId, out keyVal)
                && dataContext.Categories.Find(keyVal) != null;
        }

        [HttpGet("supplierkey")]
        public bool SupplierKey(string supplierId) {
            long keyVal;
            return long.TryParse(supplierId, out keyVal)
                && dataContext.Suppliers.Find(keyVal) != null;
        }
    }
}
```

Validation action methods must define a parameter whose name matches the field they will validate, which allows the model binding process to extract the value to test from the request query string. The response from the action method must be JSON and can be only true or false, indicating whether a value is acceptable. The action methods in Listing 29-23 receive candidate values and check they have been used as database keys for `Category` or `Supplier` objects.

■ **Tip** I could have taken advantage of model binding so that the parameter to the action methods would be converted to a `long` value, but doing so would mean that the validation method wouldn't be called if the user entered a value that cannot be converted to the `long` type. If the model binder cannot convert a value, then the MVC Framework is unable to invoke the action method and validation can't be performed. As a rule, the best approach to remote validation is to accept a `string` parameter in the action method and perform any type conversion, parsing, or model binding explicitly.

To use the remote validation method, I apply the `Remote` attribute to the `CategoryId` and `SupplierId` properties in the `Product` class, as shown in Listing 29-24.

Listing 29-24. Using the Remote Attribute in the Product.cs File in the Models Folder

```
using System.ComponentModel.DataAnnotations.Schema;
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using WebApp.Validation;
using Microsoft.AspNetCore.Mvc;
```

```
namespace WebApp.Models {

    [PhraseAndPrice(Phrase ="Small", Price = "100")]
    public class Product {

        public long ProductId { get; set; }

        [Required]
        [Display(Name = "Name")]
        public string Name { get; set; }

        [Column(TypeName = "decimal(8, 2)")]
        [Required(ErrorMessage = "Please enter a price")]
        [Range(1, 999999, ErrorMessage = "Please enter a positive price")]
        public decimal Price { get; set; }

        [PrimaryKey(ContextType= typeof(DataContext),
            DataType = typeof(Category))]
        [Remote("CategoryKey", "Validation", ErrorMessage = "Enter an existing key")]
        public long CategoryId { get; set; }
        public Category Category { get; set; }

        [PrimaryKey(ContextType = typeof(DataContext),
            DataType = typeof(Category))]
        [Remote("SupplierKey", "Validation", ErrorMessage = "Enter an existing key")]
        public long SupplierId { get; set; }
        public Supplier Supplier { get; set; }
    }
}
```

The arguments to the Remote attribute specify the name of the validation controller and its action method. I have also used the optional ErrorMessage argument to specify the error message that will be displayed when validation fails. To see the remote validation, restart ASP.NET Core and navigate to http://localhost:5000/controllers/form, enter an invalid key value, and submit the form. You will see an error message, and the value of the input element will be validated after each key press, as shown in Figure 29-11. (Only the values 1, 2, and 3 are valid for both the CategoryId and SupplierId fields.)

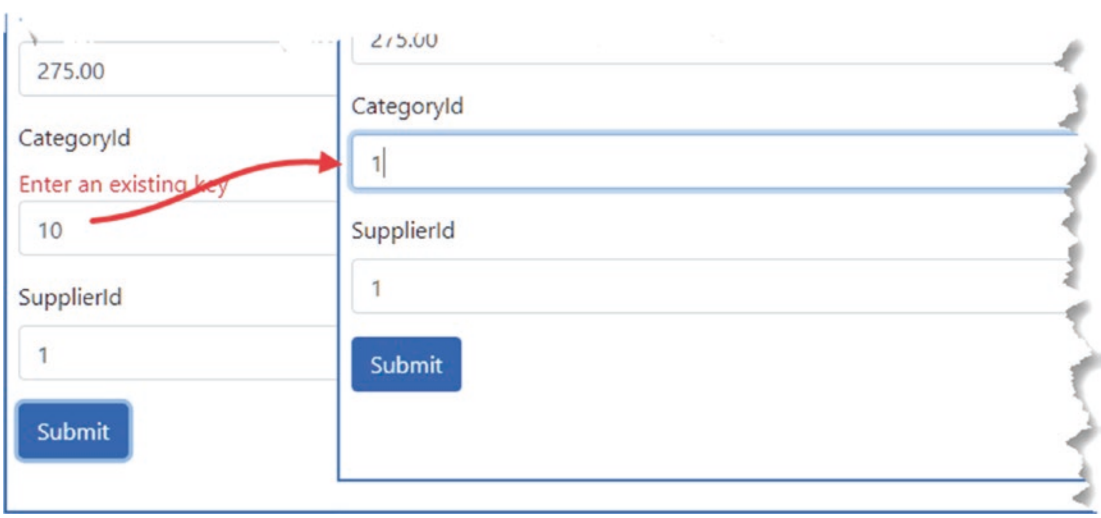

***Figure 29-11.** Performing remote validation*

■ **Caution** The validation action method will be called when the user first submits the form and again each time the data is edited. For text input elements, every keystroke will lead to a call to the server. For some applications, this can be a significant number of requests and must be accounted for when specifying the server capacity and bandwidth that an application requires in production. Also, you might choose *not* to use remote validation for properties that are expensive to validate (the example repeatedly queries the database for key values, which may not be sensible for all applications or databases).

Performing Remote Validation in Razor Pages

Remote validation works in Razor Pages, but attention must be paid to the names used in the asynchronous HTTP request used to validate values. For the controller example in the previous section, the browser will send requests to URLs like this:

```
http://localhost:5000/api/Validation/categorykey?CategoryId=1
```

But for the example Razor Page, the URL will be like this, reflecting the use of the page model:

```
http://localhost:5000/api/Validation/categorykey?Product.CategoryId=1
```

The way I prefer to address this difference is by adding parameters to the validation action methods that will accept both types of request, which is easy to do using the model binding features described in previous chapters, as shown in Listing 29-25.

Listing 29-25. Adding Parameters in the ValidationController.cs File in the Controllers Folder

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

namespace WebApp.Controllers {

    [ApiController]
    [Route("api/[controller]")]
    public class ValidationController: ControllerBase {
        private DataContext dataContext;

        public ValidationController(DataContext context) {
            dataContext = context;
        }

        [HttpGet("categorykey")]
        public bool CategoryKey(string categoryId, [FromQuery] KeyTarget target) {
            long keyVal;
            return long.TryParse(categoryId ?? target.CategoryId, out keyVal)
                && dataContext.Categories.Find(keyVal) != null;
        }

        [HttpGet("supplierkey")]
        public bool SupplierKey(string supplierId, [FromQuery] KeyTarget target) {
            long keyVal;
            return long.TryParse(supplierId ?? target.SupplierId, out keyVal)
                && dataContext.Suppliers.Find(keyVal) != null;
        }
    }
```

```
    [Bind(Prefix = "Product")]
    public class KeyTarget {
        public string CategoryId { get; set; }
        public string SupplierId{ get; set; }
    }
}
```

The KeyTarget class is configured to bind to the Product part of the request, with properties that will match the two types of remote validation request. Each action method has been given a KeyTarget parameter, which is used if no value is received for existing parameters. This allows the same action method to accommodate both types of request, which you can see by restarting ASP.NET Core, navigating to http://localhost:5000/pages/form, entering a nonexistent key value, and clicking the Submit button, which will produce the response shown in Figure 29-12.

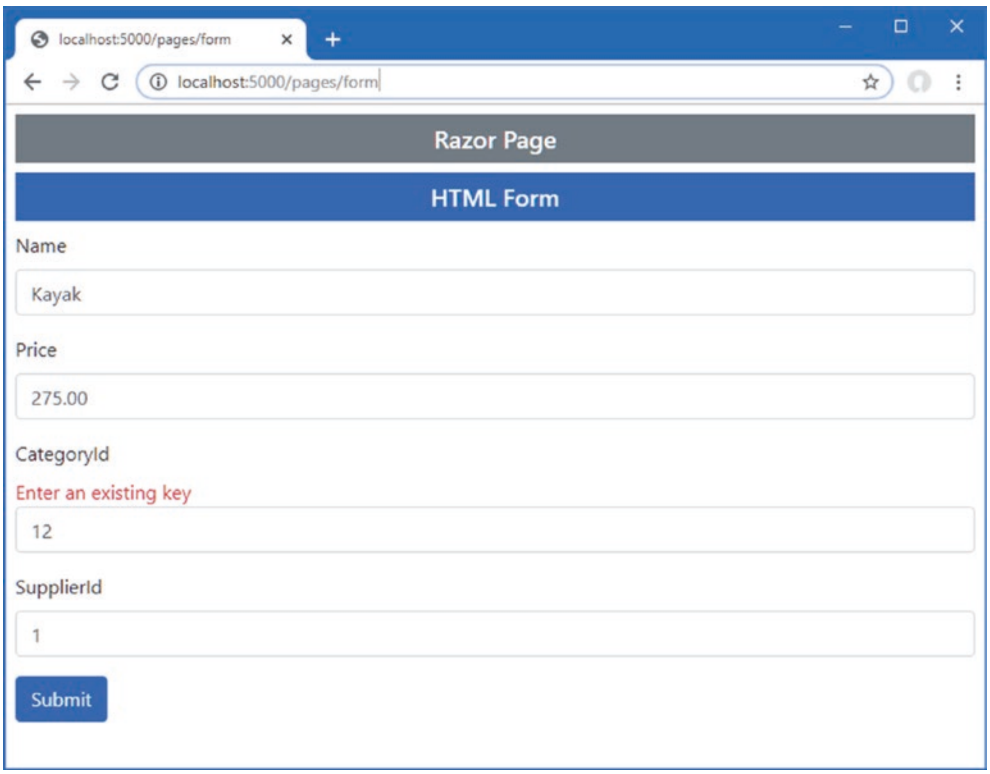

Figure 29-12. *Performing remote validation using a Razor Page*

Summary

In this chapter, I described the ASP.NET Core data validation features. I explained how to explicitly perform validation, how to use attributes to describe validation constraints, and how to validate individual properties and entire objects. I showed you how to display validation messages to the user and how to improve the user's experience of validation with client-side and remote validation. In the next chapter, I describe the ASP.NET Core filters feature.

CHAPTER 30

Using Filters

Filters inject extra logic into request processing. Filters are like middleware that is applied to a single endpoint, which can be an action or a page handler method, and they provide an elegant way to manage a specific set of requests. In this chapter, I explain how filters work, describe the different types of filter that ASP.NET Core supports, and demonstrate the use of custom filters and the filters provided by ASP.NET Core. Table 30-1 summarizes the chapter.

Table 30-1. *Chapter Summary*

Problem	Solution	Listing
Implementing a security policy	Use an authorization filter	15, 16
Implementing a resource policy, such as caching	Use a resource filter	17–19
Altering the request or response for an action method	Use an action filter	20–23
Altering the request or response for a page handler method	Use a page filter	24–26
Inspecting or altering the result produced by an endpoint	Use a result filter	27–29
Inspecting or altering uncaught exceptions	Use an exception filter	30–31
Altering the filter lifecycle	Use a filter factory or define a service	32–35
Applying filters throughout an application	Use a global filter	36, 37
Changing the order in which filters are applied	Implement the `IOrderedFilter` interface	38–42

Preparing for This Chapter

This chapter uses the WebApp project from Chapter 29. To prepare for this chapter, open a new PowerShell command prompt, navigate to the `WebApp` project folder, and run the command shown in Listing 30-1 to remove the files that are no longer required.

Listing 30-1. Removing Files from the Project

```
Remove-Item -Path Controllers,Views,Pages -Recurse -Exclude _*,Shared
```

This command removes the controllers, views, and Razor Pages, leaving behind the shared layouts, data model, and configuration files.

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-3`. See Chapter 1 for how to get help if you have problems running the examples.

A. Freeman, *Pro ASP.NET Core 3*, https://doi.org/10.1007/978-1-4842-5440-0_30

Create the WebApp/Controllers folder and add a class file named HomeController.cs to the Controllers folder with the code shown in Listing 30-2.

Listing 30-2. The Contents of the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;

namespace WebApp.Controllers {

    public class HomeController : Controller {

        public IActionResult Index() {
            return View("Message",
                "This is the Index action on the Home controller");
        }
    }
}
```

The action method renders a view called Message and passes a string as the view data. I added a Razor view named Message.cshtml with the content shown in Listing 30-3.

Listing 30-3. The Contents of the Message.cshtml File in the Views/Shared Folder

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

@if (Model is string) {
    @Model
} else if (Model is IDictionary<string, string>) {
    var dict = Model as IDictionary<string, string>;
    <table class="table table-sm table-striped table-bordered">
        <thead><tr><th>Name</th><th>Value</th></tr></thead>
        <tbody>
            @foreach (var kvp in dict) {
                <tr><td>@kvp.Key</td><td>@kvp.Value</td></tr>
            }
        </tbody>
    </table>
}
```

Add a Razor Page named Message.cshtml to the Pages folder and add the content shown in Listing 30-4.

Listing 30-4. The Contents of the Message.cshtml File in the Pages Folder

```
@page "/pages/message"
@model MessageModel
@using Microsoft.AspNetCore.Mvc.RazorPages
@using System.Collections.Generic

@if (Model.Message is string) {
    @Model.Message
} else if (Model.Message is IDictionary<string, string>) {
    var dict = Model.Message as IDictionary<string, string>;
    <table class="table table-sm table-striped table-bordered">
        <thead><tr><th>Name</th><th>Value</th></tr></thead>
        <tbody>
            @foreach (var kvp in dict) {
                <tr><td>@kvp.Key</td><td>@kvp.Value</td></tr>
            }
        </tbody>
```

```
    </table>
}

@functions {
    public class MessageModel : PageModel {

        public object Message { get; set; } = "This is the Message Razor Page";
    }
}
```

Enabling HTTPS Connections

Some of the examples in this chapter require the use of SSL. Add the configuration entries shown in Listing 30-5 to the `launchSettings.json` file in the `Properties` folder to enable SSL and set the port to 44350.

Listing 30-5. 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"
      }
    },
    "WebApp": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "http://localhost:5000;https://localhost:44350"
    }
  }
}
```

The .NET Core runtime includes a test certificate that is used for HTTPS requests. Run the commands shown in Listing 30-6 in the `WebApp` folder to regenerate and trust the test certificate.

Listing 30-6. Regenerating the Development Certificates

```
dotnet dev-certs https --clean

dotnet dev-certs https --trust
```

Click Yes to the prompts to delete the existing certificate that has already been trusted and click Yes to trust the new certificate, as shown in Figure 30-1.

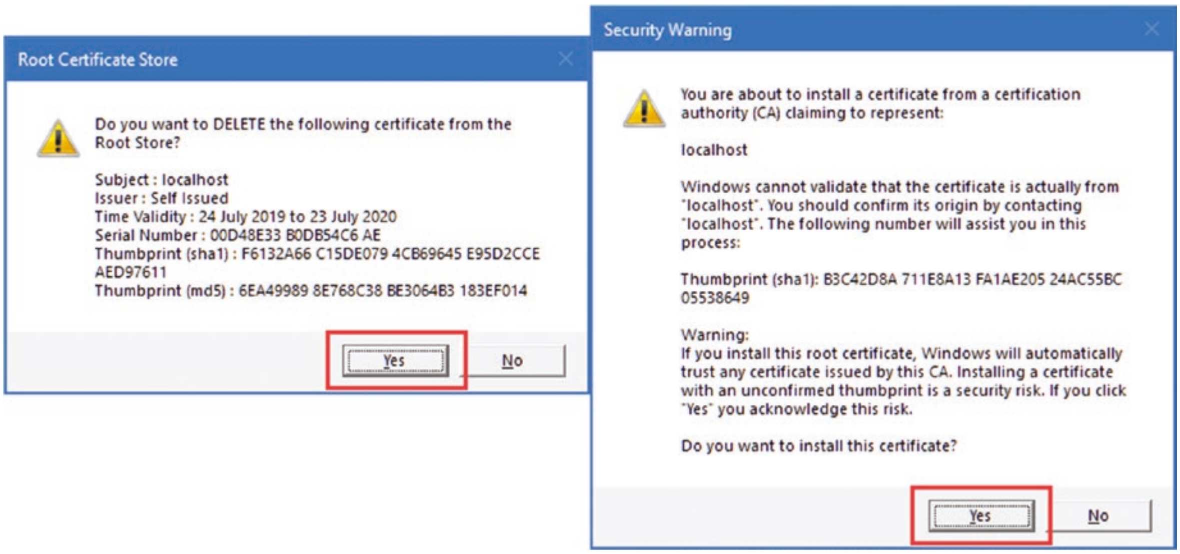

Figure 30-1. *Regenerating the HTTPS certificate*

Dropping the Database

Open a new PowerShell command prompt, navigate to the folder that contains the `WebApp.csproj` file, and run the command shown in Listing 30-7 to drop the database.

Listing 30-7. Dropping the Database

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

Running the Example Application

Select Start Without Debugging or Run Without Debugging from the Debug menu or use the PowerShell command prompt to run the command shown in Listing 30-8.

Listing 30-8. Running the Example Application

```
dotnet run
```

Use a browser to request `http://localhost:5000` and `https://localhost:44350`. Both URLs will be handled by the `Index` action defined by the `Home` controller, producing the responses shown in Figure 30-2.

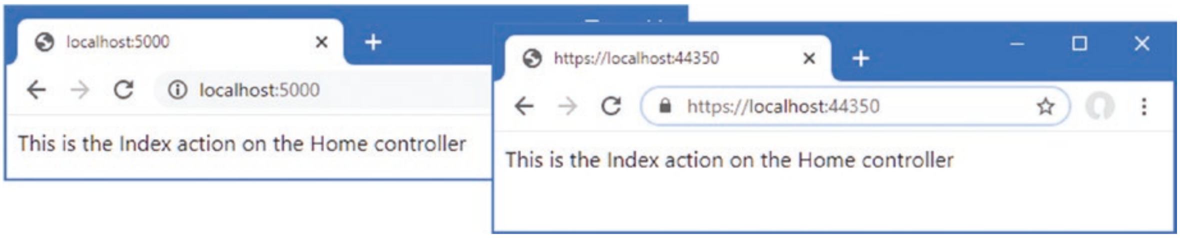

Figure 30-2. *Responses from the Home controller*

Request `http://localhost:5000/pages/message` and `https://localhost:44350/pages/message` to see the response from the `Message` Razor Page, delivered over HTTP and HTTPS, as shown in Figure 30-3.

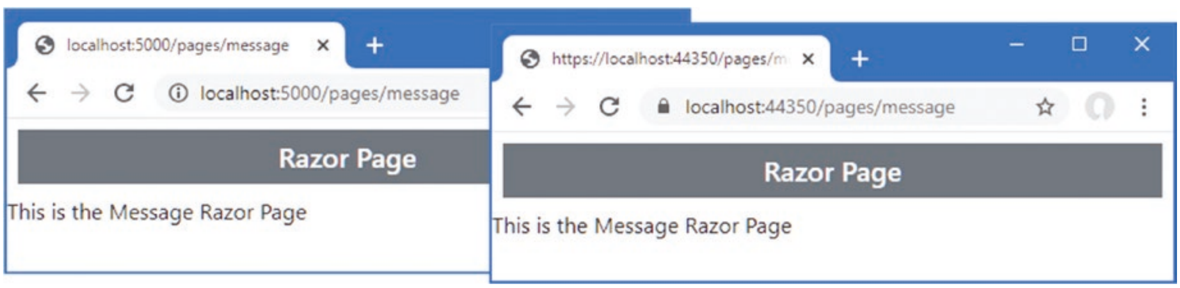

Figure 30-3. *Responses from the Message Razor Page*

Using Filters

Filters allow logic that would otherwise be applied in a middleware component or action method to be defined in a class where it can be easily reused.

Imagine that you want to enforce HTTPS requests for some action methods. In Chapter 16, I showed you how this can be done in middleware by reading the `IsHttps` property of the `HttpRequest` object. The problem with this approach is that the middleware would have to understand the configuration of the routing system to know how to intercept requests for specific action methods. A more focused approach would be to read the `HttpRequest.IsHttps` property within action methods, as shown in Listing 30-9.

Listing 30-9. Selectively Enforcing HTTPS in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;

namespace WebApp.Controllers {

    public class HomeController : Controller {

        public IActionResult Index() {
            if (Request.IsHttps) {
                return View("Message",
                    "This is the Index action on the Home controller");
            } else {
                return new StatusCodeResult(StatusCodes.Status403Forbidden);
            }
        }
    }
}
```

Restart ASP.NET Core and request `http://localhost:5000`. This method now requires HTTPS, and you will see an error response. Request `https://localhost:44350`, and you will see the message output. Figure 30-4 shows both responses.

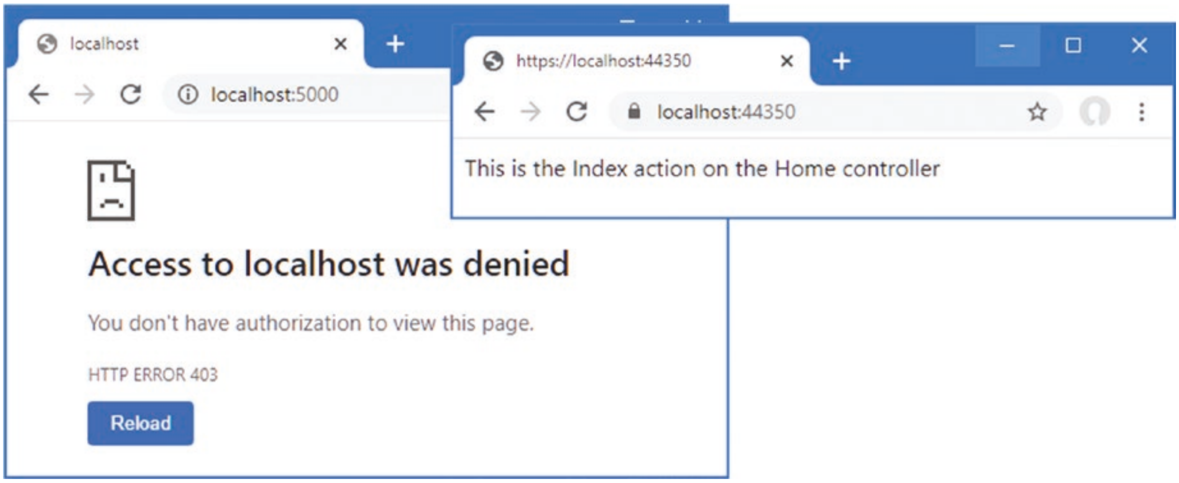

Figure 30-4. *Enforcing HTTPS in an action method*

■ **Tip** Clear your browser's history if you don't get the results you expect from the examples in this section. Browsers will often refuse to send requests to servers that have previously generated HTTPS errors, which is a good security practice but can be frustrating during development.

This approach works but has problems. The first problem is that the action method contains code that is more about implementing a security policy than about handling the request. A more serious problem is that including the HTTP-detecting code within the action method doesn't scale well and must be duplicated in every action method in the controller, as shown in Listing 30-10.

Listing 30-10. Adding Action Methods in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;

namespace WebApp.Controllers {

    public class HomeController : Controller {

        public IActionResult Index() {
            if (Request.IsHttps) {
                return View("Message",
                    "This is the Index action on the Home controller");
            } else {
                return new StatusCodeResult(StatusCodes.Status403Forbidden);
            }
        }

        public IActionResult Secure() {
            if (Request.IsHttps) {
                return View("Message",
                    "This is the Secure action on the Home controller");
```

```
            } else {
                return new StatusCodeResult(StatusCodes.Status403Forbidden);
            }
        }
    }
}
```

I must remember to implement the same check in every action method in every controller for which I want to require HTTPS. The code to implement the security policy is a substantial part of the—admittedly simple—controller, which makes the controller harder to understand, and it is only a matter of time before I forget to add it to a new action method, creating a hole in my security policy.

This is the type of problem that filters address. Listing 30-11 replaces my checks for HTTPS and implements a filter instead.

Listing 30-11. Applying a Filter in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;

namespace WebApp.Controllers {

    public class HomeController : Controller {

        [RequireHttps]
        public IActionResult Index() {
            return View("Message",
                "This is the Index action on the Home controller");
        }

        [RequireHttps]
        public IActionResult Secure() {
            return View("Message",
                "This is the Secure action on the Home controller");
        }
    }
}
```

The `RequireHttps` attribute applies one of the built-in filters provided by ASP.NET Core. This filter restricts access to action methods so that only HTTPS requests are supported and allows me to remove the security code from each method and focus on handling the successful requests.

■ **Note** The `RequireHttps` filter doesn't work the same way as my custom code. For `GET` requests, the `RequireHttps` attribute redirects the client to the originally requested URL, but it does so by using the `https` scheme so that a request to `http://localhost:5000` will be redirected to `https://localhost:5000`. This makes sense for most deployed applications but not during development because HTTP and HTTPS are on different local ports. The `RequireHttpsAttribute` class defines a protected method called `HandleNonHttpsRequest` that you can override to change the behavior. Alternatively, I re-create the original functionality from scratch in the "Understanding Authorization Filters" section.

I must still remember to apply the `RequireHttps` attribute to each action method, which means that I might forget. But filters have a useful trick: applying the attribute to a controller class has the same effect as applying it to each individual action method, as shown in Listing 30-12.

Listing 30-12. Applying a Filter to All Actions in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;

namespace WebApp.Controllers {

    [RequireHttps]
    public class HomeController : Controller {

        public IActionResult Index() {
            return View("Message",
                "This is the Index action on the Home controller");
        }

        public IActionResult Secure() {
            return View("Message",
                "This is the Secure action on the Home controller");
        }
    }
}
```

Filters can be applied with differing levels of granularity. If you want to restrict access to some actions but not others, then you can apply the `RequireHttps` attribute to just those methods. If you want to protect all the action methods, including any that you add to the controller in the future, then the `RequireHttps` attribute can be applied to the class. If you want to apply a filter to every action in an application, then you can use *global filters*, which I describe later in this chapter.

Using Filters in Razor Pages

Filters can also be used in Razor Pages. To implement the HTTPS-only policy in the `Message` Razor Pages, for example, I would have to add a handler method that inspects the connection, as shown in Listing 30-13.

Listing 30-13. Checking Connections in the Message.cshtml File in the Pages Folder

```
@page "/pages/message"
@model MessageModel
@using Microsoft.AspNetCore.Mvc.RazorPages
@using System.Collections.Generic
@using Microsoft.AspNetCore.Http

@if (Model.Message is string) {
    @Model.Message
} else if (Model.Message is IDictionary<string, string>) {
    var dict = Model.Message as IDictionary<string, string>;
    <table class="table table-sm table-striped table-bordered">
        <thead><tr><th>Name</th><th>Value</th></tr></thead>
        <tbody>
            @foreach (var kvp in dict) {
                <tr><td>@kvp.Key</td><td>@kvp.Value</td></tr>
            }
        </tbody>
    </table>
}
```

```
@functions {

    public class MessageModel : PageModel {

        public object Message { get; set; } = "This is the Message Razor Page";

        public IActionResult OnGet() {
            if (!Request.IsHttps) {
                return new StatusCodeResult(StatusCodes.Status403Forbidden);
            } else {
                return Page();
            }
        }
    }
}
```

The handler method works, but it is awkward and presents the same problems encountered with action methods. When using filters in Razor Pages, the attribute can be applied to the handler method or, as shown in Listing 30-14, to the entire class.

Listing 30-14. Applying a Filter in the Message.cshtml File in the Pages Folder

```
@page "/pages/message"
@model MessageModel
@using Microsoft.AspNetCore.Mvc.RazorPages
@using System.Collections.Generic
@using Microsoft.AspNetCore.Http

@if (Model.Message is string) {
    @Model.Message
} else if (Model.Message is IDictionary<string, string>) {
    var dict = Model.Message as IDictionary<string, string>;
    <table class="table table-sm table-striped table-bordered">
        <thead><tr><th>Name</th><th>Value</th></tr></thead>
        <tbody>
            @foreach (var kvp in dict) {
                <tr><td>@kvp.Key</td><td>@kvp.Value</td></tr>
            }
        </tbody>
    </table>
}

@functions {

    [RequireHttps]
    public class MessageModel : PageModel {

        public object Message { get; set; } = "This is the Message Razor Page";
    }
}
```

You will see a normal response if you request `https://localhost:44350/pages/message`. If you request the regular HTTP URL, `http://localhost:5000/pages/messages`, the filter will redirect the request, and you will see an error (as noted earlier, the `RequireHttps` filter redirects the browser to a port that is not enabled in the example application).

Understanding Filters

ASP.NET Core supports different types of filters, each of which is intended for a different purpose. Table 30-2 describes the filter categories.

Table 30-2. The Filter Types

Name	Description
Authorization filters	This type of filter is used to apply the application's authorization policy.
Resource filters	This type of filter is used to intercept requests, typically to implement features such as caching.
Action filters	This type of filter is used to modify the request before it is received by an action method or to modify the action result after it has been produced. This type of filter can be applied only to controllers and actions.
Page filters	This type of filter is used to modify the request before it is received by a Razor Page handler method or to modify the action result after it has been produced. This type of filter can be applied only to Razor Pages.
Result filters	This type of filter is used to alter the action result before it is executed or to modify the result after execution.
Exception filters	This type of filter is used to handle exceptions that occur during the execution of the action method or page handler.

Filters have their own pipeline and are executed in a specific order, as shown in Figure 30-5.

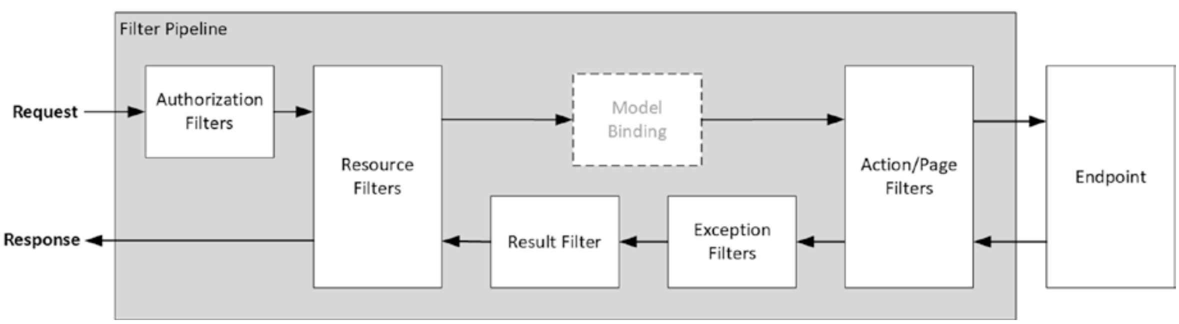

***Figure 30-5.** The filter pipeline*

Filters can short-circuit the filter pipeline to prevent a request from being forwarded to the next filter. For example, an authorization filter can short-circuit the pipeline and return an error response if the user is unauthenticated. The resource, action, and page filters are able to inspect the request before and after it has been handled by the endpoint, allowing these types of filter to short-circuit the pipeline; to alter the request before it is handled; or to alter the response. (I have simplified the flow of filters in Figure 30-5. Page filters run before and after the model binding process, as described in the "Understanding Page Filters" section.)

Each type of filter is implemented using interfaces defined by ASP.NET Core, which also provides base classes that make it easy to apply some types of filters as attributes. I describe each interface and the attribute classes in the sections that follow, but they are shown in Table 30-3 for quick reference.

Table 30-3. *The Filter Types, Interfaces, and Attribute Base Classes*

Filter Type	Interfaces	Attribute Class
Authorization filters	IAuthorizationFilter IAsyncAuthorizationFilter	No attribute class is provided.
Resource filters	IResourceFilter IAsyncResourceFilter	No attribute class is provided.
Action filters	IActionFilter IAsyncActionFilter	ActionFilterAttribute
Page filters	IPageFilter IAsyncPageFilter	No attribute class is provided.
Result filters	IResultFilter IAsyncResultFilter IAlwaysRunResultFilter IAsyncAlwaysRunResultFilter	ResultFilterAttribute
Exception Filters	IExceptionFilter IAsyncExceptionFilter	ExceptionFilterAttribute

Creating Custom Filters

Filters implement the IFilterMetadata interface, which is in the Microsoft.AspNetCore.Mvc.Filters namespace. Here is the interface:

```
namespace Microsoft.AspNetCore.Mvc.Filters {
    public interface IFilterMetadata { }
}
```

The interface is empty and doesn't require a filter to implement any specific behaviors. This is because each of the categories of filter described in the previous section works in a different way. Filters are provided with context data in the form of a FilterContext object. For convenience, Table 30-4 describes the properties that FilterContext provides.

Table 30-4. *The FilterContext Properties*

Name	Description
ActionDescriptor	This property returns an ActionDescriptor object, which describes the action method.
HttpContext	This property returns an HttpContext object, which provides details of the HTTP request and the HTTP response that will be sent in return.
ModelState	This property returns a ModelStateDictionary object, which is used to validate data sent by the client.
RouteData	This property returns a RouteData object that describes the way that the routing system has processed the request.
Filters	This property returns a list of filters that have been applied to the action method, expressed as an IList<IFilterMetadata>.

Understanding Authorization Filters

Authorization filters are used to implement an application's security policy. Authorization filters are executed before other types of filter and before the endpoint handles the request. Here is the definition of the IAuthorizationFilter interface:

```
namespace Microsoft.AspNetCore.Mvc.Filters {

    public interface IAuthorizationFilter : IFilterMetadata {

        void OnAuthorization(AuthorizationFilterContext context);
    }
}
```

The OnAuthorization method is called to provide the filter with the opportunity to authorize the request. For asynchronous authorization filters, here is the definition of the IAsyncAuthorizationFilter interface:

```
using System.Threading.Tasks;

namespace Microsoft.AspNetCore.Mvc.Filters {

    public interface IAsyncAuthorizationFilter : IFilterMetadata {

        Task OnAuthorizationAsync(AuthorizationFilterContext context);
    }
}
```

The OnAuthorizationAsync method is called so that the filter can authorize the request. Whichever interface is used, the filter receives context data describing the request through an AuthorizationFilterContext object, which is derived from the FilterContext class and adds one important property, as described in Table 30-5.

Table 30-5. *The AuthorizationFilterContext Property*

Name	Description
Result	This IActionResult property is set by authorization filters when the request doesn't comply with the application's authorization policy. If this property is set, then ASP.NET Core executes the IActionResult instead of invoking the endpoint.

Creating an Authorization Filter

To demonstrate how authorization filters work, I created a Filters folder in the example project, added a class file called HttpsOnlyAttribute.cs, and used it to define the filter shown in Listing 30-15.

Listing 30-15. The Contents of the HttpsOnlyAttribute.cs File in the Filters Folder

```
using System;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;

namespace WebApp.Filters {
    public class HttpsOnlyAttribute : Attribute, IAuthorizationFilter {

        public void OnAuthorization(AuthorizationFilterContext context) {
            if (!context.HttpContext.Request.IsHttps) {
                context.Result =
                    new StatusCodeResult(StatusCodes.Status403Forbidden);
            }
        }
    }
}
```

An authorization filter does nothing if a request complies with the authorization policy and inaction allows ASP.NET Core to move on to the next filter and, eventually, to execute the endpoint. If there is a problem, the filter sets the Result property of the AuthorizationFilterContext object that is passed to the OnAuthorization method. This prevents further execution from happening and provides a result to return to the client. In the listing, the HttpsOnlyAttribute class inspects the IsHttps property of the HttpRequest context object and sets the Result property to interrupt execution if the request has been made without HTTPS. Authorization filters can be applied to controllers, action methods, and Razor Pages. Listing 30-16 applies the new filter to the Home controller.

Listing 30-16. Applying a Custom Filter in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;
using WebApp.Filters;

namespace WebApp.Controllers {

    [HttpsOnly]
    public class HomeController : Controller {

        public IActionResult Index() {
            return View("Message",
                "This is the Index action on the Home controller");
        }

        public IActionResult Secure() {
            return View("Message",
                "This is the Secure action on the Home controller");
        }
    }
}
```

This filter re-creates the functionality that I included in the action methods in Listing 30-10. This is less useful in real projects than doing a redirection like the built-in `RequireHttps` filter because users won't understand the meaning of a 403 status code, but it does provide a useful example of how authorization filters work. Restart ASP.NET Core and request `http://localhost:5000`, and you will see the effect of the filter, as shown in Figure 30-6. Request `https://localhost:44350`, and you will receive the response from the action method, also shown in the figure.

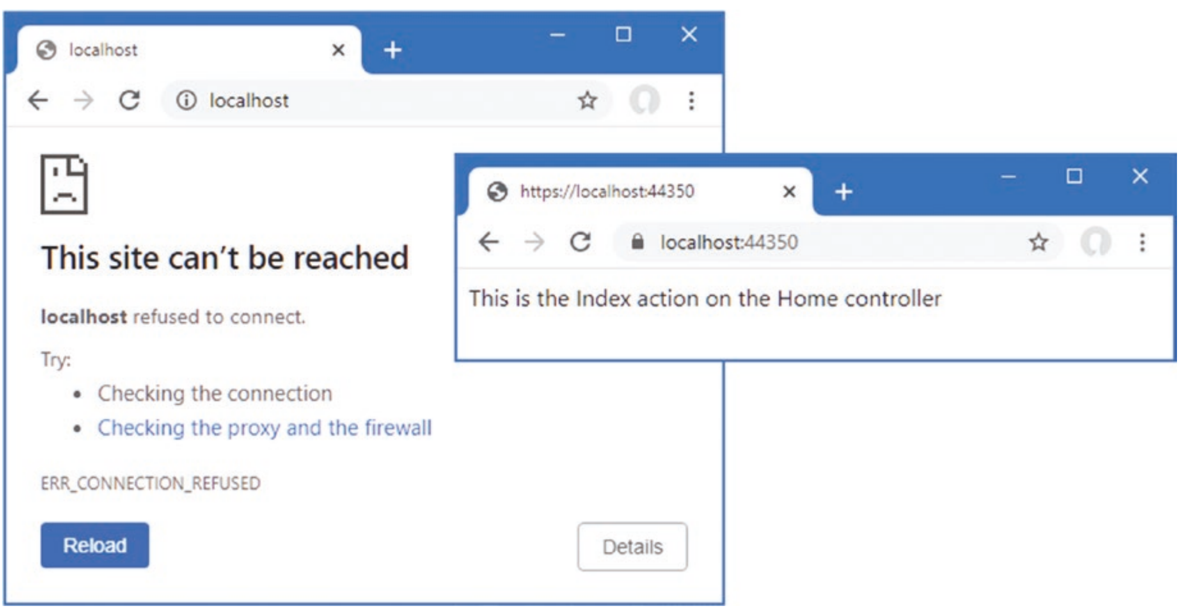

Figure 30-6. *Applying a custom authorization filter*

Understanding Resource Filters

Resource filters are executed twice for each request: before the ASP.NET Core model binding process and again before the action result is processed to generate the result. Here is the definition of the IResourceFilter interface:

```
namespace Microsoft.AspNetCore.Mvc.Filters {
    public interface IResourceFilter : IFilterMetadata {

        void OnResourceExecuting(ResourceExecutingContext context);

        void OnResourceExecuted(ResourceExecutedContext context);
    }
}
```

The OnResourceExecuting method is called when a request is being processed, and the OnResourceExecuted method is called after the endpoint has handled the request but before the action result is executed. For asynchronous resource filters, here is the definition of the IAsyncResourceFilter interface:

```
namespace Microsoft.AspNetCore.Mvc.Filters {
    public interface IAsyncResourceFilter : IFilterMetadata {

        Task OnResourceExecutionAsync(ResourceExecutingContext context,
            ResourceExecutionDelegate next);
    }
}
```

This interface defines a single method that receives a context object and a delegate to invoke. The resource filter is able to inspect the request before invoking the delegate and inspect the response before it is executed. The OnResourceExecuting method is provided with context using the ResourceExecutingContext class, which defines the property shown in Table 30-6 in addition to those defined by the FilterContext class.

Table 30-6. *The Property Defined by the ResourceExecutingContext Class*

Name	Description
Result	This IActionResult property is used to provide a result to short-circuit the pipeline.

The OnResourceExecuted method is provided with context using the ResourceExecutedContext class, which defines the properties shown in Table 30-7, in addition to those defined by the FilterContext class.

Table 30-7. *The Properties Defined by the ResourceExecutedContext Class*

Name	Description
Result	This IActionResult property provides the action result that will be used to produce a response.
ValueProviderFactories	This property returns an IList<IValueProviderFactory>, which provides access to the objects that provide values for the model binding process.

Creating a Resource Filter

Resource filters are usually used where it is possible to short-circuit the pipeline and provide a response early, such as when implementing data caching. To create a simple caching filter, add a class file called SimpleCacheAttribute.cs to the Filters folder with the code shown in Listing 30-17.

FILTERS AND DEPENDENCY INJECTION

Filters that are applied as attributes cannot declare dependencies in their constructors unless they implement the `IFilterFactory` interface and take responsibility for creating instances directly, as explained in the "Creating Filter Factories" section later in this chapter.

Listing 30-17. The Contents of the SimpleCacheAttribute.cs File in the Filters Folder

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

namespace WebApp.Filters {

    public class SimpleCacheAttribute : Attribute, IResourceFilter {
        private Dictionary<PathString, IActionResult> CachedResponses
            = new Dictionary<PathString, IActionResult>();

        public void OnResourceExecuting(ResourceExecutingContext context) {
            PathString path = context.HttpContext.Request.Path;
            if (CachedResponses.ContainsKey(path)) {
                context.Result = CachedResponses[path];
                CachedResponses.Remove(path);

            }
        }

        public void OnResourceExecuted(ResourceExecutedContext context) {
            CachedResponses.Add(context.HttpContext.Request.Path, context.Result);
        }
    }
}
```

This filter isn't an especially useful cache, but it does show how a resource filter works. The `OnResourceExecuting` method provides the filter with the opportunity to short-circuit the pipeline by setting the context object's `Result` property to a previously cached action result. If a value is assigned to the `Result` property, then the filter pipeline is short-circuited, and the action result is executed to produce the response for the client. Cached action results are used only once and then discarded from the cache. If no value is assigned to the `Result` property, then the request passes to the next step in the pipeline, which may be another filter or the endpoint.

The `OnResourceExecuted` method provides the filter with the action results that are produced when the pipeline is not short-circuited. In this case, the filter caches the action result so that it can be used for subsequent requests. Resource filters can be applied to controllers, action methods, and Razor Pages. Listing 30-18 applies the custom resource filter to the `Message` Razor Page and adds a timestamp that will help determine when an action result is cached.

Listing 30-18. Applying a Resource Filter in the Message.cshtml File in the Pages Folder

```
@page "/pages/message"
@model MessageModel
@using Microsoft.AspNetCore.Mvc.RazorPages
@using System.Collections.Generic
@using Microsoft.AspNetCore.Http
@using WebApp.Filters
```

```
@if (Model.Message is string) {
    @Model.Message
} else if (Model.Message is IDictionary<string, string>) {
    var dict = Model.Message as IDictionary<string, string>;
    <table class="table table-sm table-striped table-bordered">
        <thead><tr><th>Name</th><th>Value</th></tr></thead>
        <tbody>
            @foreach (var kvp in dict) {
                <tr><td>@kvp.Key</td><td>@kvp.Value</td></tr>
            }
        </tbody>
    </table>
}

@functions {

    [RequireHttps]
    [SimpleCache]
    public class MessageModel : PageModel {

        public object Message { get; set; } =
            $"{DateTime.Now.ToLongTimeString()}: This is the Message Razor Page";
    }
}
```

To see the effect of the resource filter, restart ASP.NET Core and request `https://localhost:44350/pages/message`. Since this is the first request for the path, there will be no cached result, and the request will be forwarded along the pipeline. As the response is processed, the resource filter will cache the action result for future use. Reload the browser to repeat the request, and you will see the same timestamp, indicating that the cached action result has been used. The cached item is removed when it is used, which means that reloading the browser will generate a response with a fresh timestamp, as shown in Figure 30-7.

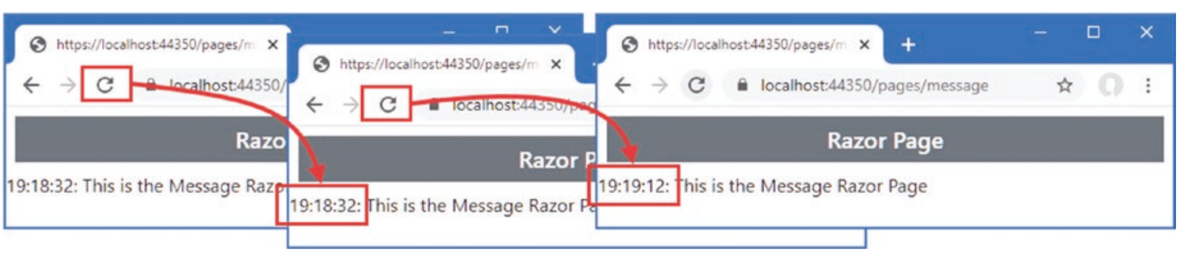

Figure 30-7. *Using a resource filter*

Creating an Asynchronous Resource Filter

The interface for asynchronous resource filters uses a single method that receives a delegate used to forward the request along the filter pipeline. Listing 30-19 reimplements the caching filter from the previous example so that it implements the `IAsyncResourceFilter` interface.

Listing 30-19. Creating an Asynchronous Filter in the SimpleCacheAttribute.cs File in the Filters Folder

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
```

```
namespace WebApp.Filters {

    public class SimpleCacheAttribute : Attribute, IAsyncResourceFilter {
        private Dictionary<PathString, IActionResult> CachedResponses
            = new Dictionary<PathString, IActionResult>();

        public async Task OnResourceExecutionAsync(ResourceExecutingContext context,
                ResourceExecutionDelegate next) {
            PathString path = context.HttpContext.Request.Path;
            if (CachedResponses.ContainsKey(path)) {
                context.Result = CachedResponses[path];
                CachedResponses.Remove(path);
            } else {
                ResourceExecutedContext execContext = await next();
                CachedResponses.Add(context.HttpContext.Request.Path,
                    execContext.Result);
            }
        }
    }
}
```

The OnResourceExecutionAsync method receives a ResourceExecutingContext object, which is used to determine whether the pipeline can be short-circuited. If it cannot, the delegate is invoked without arguments and asynchronously produces a ResourceExecutedContext object when the request has been handled and is making its way back along the pipeline. Restart ASP.NET Core and repeat the requests described in the previous section, and you will see the same caching behavior, as shown in Figure 30-7.

■ **Caution** It is important not to confuse the two context objects. The action result produced by the endpoint is available only in the context object that is returned by the delegate.

Understanding Action Filters

Like resource filters, action filters are executed twice. The difference is that action filters are executed after the model binding process, whereas resource filters are executed before model binding. This means that resource filters can short-circuit the pipeline and minimize the work that ASP.NET Core does on the request. Action filters are used when model binding is required, which means they are used for tasks such as altering the model or enforcing validation. Action filters can be applied only to controllers and action methods, unlike resource filters, which can also be used with Razor Pages. (The Razor Pages equivalent to action filters is the page filter, described in the "Understanding Page Filters" section.) Here is the IActionFilter interface:

```
namespace Microsoft.AspNetCore.Mvc.Filters {

    public interface IActionFilter : IFilterMetadata {

        void OnActionExecuting(ActionExecutingContext context);

        void OnActionExecuted(ActionExecutedContext context);
    }
}
```

When an action filter has been applied to an action method, the OnActionExecuting method is called just before the action method is invoked, and the OnActionExecuted method is called just after. Action filters are provided with context data through two different context classes: ActionExecutingContext for the OnActionExecuting method and ActionExecutedContext for the OnActionExecuted method.

The ActionExecutingContext class, which is used to describe an action that is about to be invoked, defines the properties described in Table 30-8, in addition to the FilterContext properties.

Table 30-8. *The ActionExecutingContext Property*

Name	Description
Controller	This property returns the controller whose action method is about to be invoked. (Details of the action method are available through the ActionDescriptor property inherited from the base classes.)
ActionArguments	This property returns a dictionary of the arguments that will be passed to the action method, indexed by name. The filter can insert, remove, or change the arguments.
Result	If the filter assigns an IActionResult to this property, then the pipeline will be short-circuited, and the action result will be used to generate the response to the client without invoking the action method.

The ActionExecutedContext class is used to represent an action that has been executed and defines the properties described in Table 30-9, in addition to the FilterContext properties.

Table 30-9. *The ActionExecutedContext Properties*

Name	Description
Controller	This property returns the Controller object whose action method will be invoked.
Canceled	This bool property is set to true if another action filter has short-circuited the pipeline by assigning an action result to the Result property of the ActionExecutingContext object.
Exception	This property contains any Exception that was thrown by the action method.
ExceptionDispatchInfo	This method returns an ExceptionDispatchInfo object that contains the stack trace details of any exception thrown by the action method.
ExceptionHandled	Setting this property to true indicates that the filter has handled the exception, which will not be propagated any further.
Result	This property returns the IActionResult produced by the action method. The filter can change or replace the action result if required.

Asynchronous action filters are implemented using the IAsyncActionFilter interface.

```
namespace Microsoft.AspNetCore.Mvc.Filters {

    public interface IAsyncActionFilter : IFilterMetadata {

        Task OnActionExecutionAsync(ActionExecutingContext context,
            ActionExecutionDelegate next);
    }
}
```

This interface follows the same pattern as the IAsyncResourceFilter interface described earlier in the chapter. The OnActionExecutionAsync method is provided with an ActionExecutingContext object and a delegate. The ActionExecutingContext object describes the request before it is received by the action method. The filter can short-circuit the pipeline by assigning a value to the ActionExecutingContext.Result property or pass it along by invoking the delegate. The delegate asynchronously produces an ActionExecutedContext object that describes the result from the action method.

Creating an Action Filter

Add a class file called `ChangeArgAttribute.cs` to the `Filters` folder and use it to define the action filter shown in Listing 30-20.

Listing 30-20. The Contents of the ChangeArgAttribute.cs File in the Filters Folder

```
using Microsoft.AspNetCore.Mvc.Filters;
using System;
using System.Threading.Tasks;

namespace WebApp.Filters {
    public class ChangeArgAttribute : Attribute, IAsyncActionFilter {

        public async Task OnActionExecutionAsync(ActionExecutingContext context,
                ActionExecutionDelegate next) {

            if (context.ActionArguments.ContainsKey("message1")) {
                context.ActionArguments["message1"] = "New message";
            }
            await next();
        }
    }
}
```

The filter looks for an action argument named `message1` and changes the value that will be used to invoke the action method. The values that will be used for the action method arguments are determined by the model binding process. Listing 30-21 adds an action method to the `Home` controller and applies the new filter.

Listing 30-21. Applying a Filter in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;
using WebApp.Filters;

namespace WebApp.Controllers {

    [HttpsOnly]
    public class HomeController : Controller {

        public IActionResult Index() {
            return View("Message",
                "This is the Index action on the Home controller");
        }

        public IActionResult Secure() {
            return View("Message",
                "This is the Secure action on the Home controller");
        }

        [ChangeArg]
        public IActionResult Messages(string message1, string message2 = "None") {
            return View("Message", $"{message1}, {message2}");
        }
    }
}
```

Restart ASP.NET Core and request `https://localhost:44350/home/messages?message1=hello&message2=world`. The model binding process will locate values for the parameters defined by the action method from the query string. One of those values is then modified by the action filter, producing the response shown in Figure 30-8.

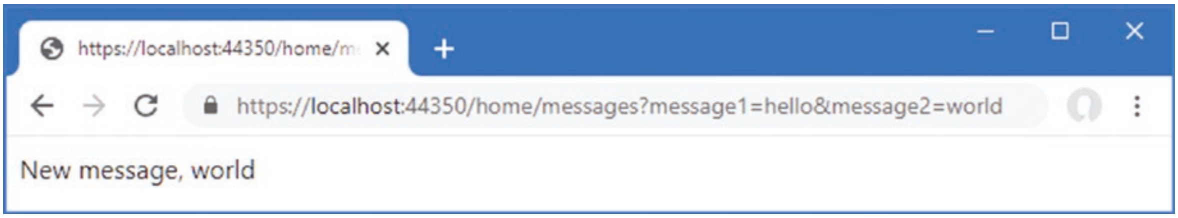

Figure 30-8. *Using an action filter*

Implementing an Action Filter Using the Attribute Base Class

Action attributes can also be implemented by deriving from the `ActionFilterAttribute` class, which extends `Attribute` and inherits both the `IActionFilter` and `IAsyncActionFilter` interfaces so that implementation classes override just the methods they require. In Listing 30-22, I have reimplemented the `ChangeArg` filter so that it is derived from `ActionFilterAttribute`.

Listing 30-22. Using a Filter Base Class in the ChangeArgsAttribute.cs File in the Filters Folder

```
using Microsoft.AspNetCore.Mvc.Filters;
using System;
using System.Threading.Tasks;

namespace WebApp.Filters {
    public class ChangeArgAttribute : ActionFilterAttribute {

        public override async Task OnActionExecutionAsync(
            ActionExecutingContext context,
                ActionExecutionDelegate next) {

            if (context.ActionArguments.ContainsKey("message1")) {
                context.ActionArguments["message1"] = "New message";
            }
            await next();
        }
    }
}
```

This attribute behaves in just the same way as the earlier implementation, and the use of the base class is a matter of preference. Restart ASP.NET Core and request `https://localhost:44350/home/messages?message1=hello&message2=world`, and you will see the response shown in Figure 30-8.

Using the Controller Filter Methods

The `Controller` class, which is the base for controllers that render Razor views, implements the `IActionFilter` and `IAsyncActionFilter` interfaces, which means you can define functionality and apply it to the actions defined by a controller and any derived controllers. Listing 30-23 implements the `ChangeArg` filter functionality directly in the `HomeController` class.

Listing 30-23. Using Action Filter Methods in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;
using WebApp.Filters;
using Microsoft.AspNetCore.Mvc.Filters;

namespace WebApp.Controllers {

    [HttpsOnly]
    public class HomeController : Controller {

        public IActionResult Index() {
            return View("Message",
                "This is the Index action on the Home controller");
        }

        public IActionResult Secure() {
            return View("Message",
                "This is the Secure action on the Home controller");
        }

        //[ChangeArg]
        public IActionResult Messages(string message1, string message2 = "None") {
            return View("Message", $"{message1}, {message2}");
        }

        public override void OnActionExecuting(ActionExecutingContext context) {
            if (context.ActionArguments.ContainsKey("message1")) {
                context.ActionArguments["message1"] = "New message";
            }
        }
    }
}
```

The Home controller overrides the Controller implementation of the OnActionExecuting method and uses it to modify the arguments that will be passed to the execution method.

Restart ASP.NET Core and request https://localhost:44350/home/messages?message1=hello&message2=world, and you will see the response shown in Figure 30-8.

Understanding Page Filters

Page filters are the Razor Page equivalent of action filters. Here is the IPageFilter interface, which is implemented by synchronous page filters:

```
namespace Microsoft.AspNetCore.Mvc.Filters {

    public interface IPageFilter : IFilterMetadata {

        void OnPageHandlerSelected(PageHandlerSelectedContext context);

        void OnPageHandlerExecuting(PageHandlerExecutingContext context);

        void OnPageHandlerExecuted(PageHandlerExecutedContext context);
    }
}
```

The OnPageHandlerSelected method is invoked after ASP.NET Core has selected the page handler method but before model binding has been performed, which means the arguments for the handler method have not been determined. This method receives context through the PageHandlerSelectedContext class, which defines the properties shown in Table 30-10, in addition to those defined by the FilterContext class. This method cannot be used to short-circuit the pipeline, but it can alter the handler method that will receive the request.

Table 30-10. *The PageHandlerSelectedContext Properties*

Name	Description
ActionDescriptor	This property returns the description of the Razor Page.
HandlerMethod	This property returns a HandlerMethodDescriptor object that describes the selected handler method.
HandlerInstance	This property returns the instance of the Razor Page that will handle the request.

The OnPageHandlerExecuting method is called after the model binding process has completed but before the page handler method is invoked. This method receives context through the PageHandlerExecutingContext class, which defines the properties shown in Table 30-11.

Table 30-11. *The PageHandlerExecutingContext Properties*

Name	Description
HandlerArguments	This property returns a dictionary containing the page handler arguments, indexed by name.
Result	The filter can short-circuit the pipeline by assigning an IActionResult object to this property.

The OnPageHandlerExecuted method is called after the page handler method has been invoked but before the action result is processed to create a response. This method receives context through the PageHandlerExecutedContext class, which defines the properties shown in Table 30-12 in addition to the PageHandlerSelectedContext properties.

Table 30-12. *The PageHandlerExecutedContext Properties*

Name	Description
Canceled	This property returns true if another filter short-circuited the filter pipeline.
Exception	This property returns an exception if one was thrown by the page handler method.
ExceptionHandled	This property is set to true to indicate that an exception thrown by the page handler has been handled by the filter.
Result	This property returns the action result that will be used to create a response for the client.

Asynchronous page filters are created by implementing the IAsyncPageFilter interface, which is defined like this:

```
namespace Microsoft.AspNetCore.Mvc.Filters {
    public interface IAsyncPageFilter : IFilterMetadata {

        Task OnPageHandlerSelectionAsync(PageHandlerSelectedContext context);

        Task OnPageHandlerExecutionAsync(PageHandlerExecutingContext context,
            PageHandlerExecutionDelegate next);
    }
}
```

The OnPageHandlerSelectionAsync is called after the handler method is selected and is equivalent to the synchronous OnPageHandlerSelected method. The OnPageHandlerExecutionAsync is provided with a PageHandlerExecutingContext object that allows it to short-circuit the pipeline and a delegate that is invoked to pass on the request. The delegate produces a PageHandlerExecutedContext object that can be used to inspect or alter the action result produced by the handler method.

Creating a Page Filter

To create a page filter, add a class file named ChangePageArgs.cs to the Filters folder and use it to define the class shown in Listing 30-24.

Listing 30-24. The Contents of the ChangePageArgs.cs File in the Filters Folder

```
using Microsoft.AspNetCore.Mvc.Filters;
using System;

namespace WebApp.Filters {
    public class ChangePageArgs : Attribute, IPageFilter {

        public void OnPageHandlerSelected(PageHandlerSelectedContext context) {
            // do nothing
        }

        public void OnPageHandlerExecuting(PageHandlerExecutingContext context) {
            if (context.HandlerArguments.ContainsKey("message1")) {
                context.HandlerArguments["message1"] = "New message";
            }
        }

        public void OnPageHandlerExecuted(PageHandlerExecutedContext context) {
            // do nothing
        }
    }
}
```

The page filter in Listing 30-24 performs the same task as the action filter I created in the previous section. In Listing 30-25, I have modified the Message Razor Page to define a handler method and have applied the page filter. Page filters can be applied to individual handler methods or, as in the listing, to the page model class, in which case the filter is used for all handler methods. (I also disabled the SimpleCache filter in Listing 30-25. Resource filters can work alongside page filters. I disabled this filter because caching responses makes some of the examples more difficult to follow.)

Listing 30-25. Using a Page Filter in the Message.cshtml File in the Pages Folder

```
@page "/pages/message"
@model MessageModel
@using Microsoft.AspNetCore.Mvc.RazorPages
@using System.Collections.Generic
@using Microsoft.AspNetCore.Http
@using WebApp.Filters

@if (Model.Message is string) {
    @Model.Message
} else if (Model.Message is IDictionary<string, string>) {
    var dict = Model.Message as IDictionary<string, string>;
    <table class="table table-sm table-striped table-bordered">
        <thead><tr><th>Name</th><th>Value</th></tr></thead>
        <tbody>
```

```
            @foreach (var kvp in dict) {
                <tr><td>@kvp.Key</td><td>@kvp.Value</td></tr>
            }
        </tbody>
    </table>
}

@functions {

    [RequireHttps]
    //[SimpleCache]
    [ChangePageArgs]
    public class MessageModel : PageModel {

        public object Message { get; set; } =
            $"{DateTime.Now.ToLongTimeString()}: This is the Message Razor Page";

        public void OnGet(string message1, string message2) {
            Message = $"{message1}, {message2}";
        }
    }
}
```

Restart ASP.NET Core and request `https://localhost:44350/pages/message?message1=hello&message2=world`. The page filter will replace the value of the `message1` argument for the `OnGet` handler method, which produces the response shown in Figure 30-9.

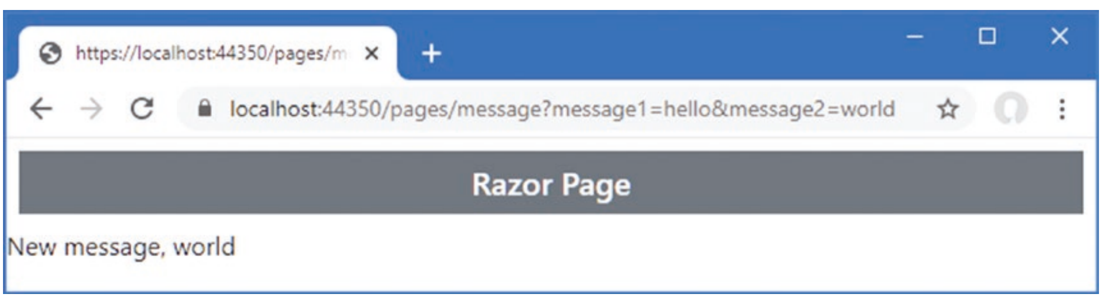

Figure 30-9. *Using a page filter*

Using the Page Model Filter Methods

The `PageModel` class, which is used as the base for page model classes, implements the `IPageFilter` and `IAsyncPageFilter` interfaces, which means you can add filter functionality directly to a page model, as shown in Listing 30-26.

Listing 30-26. Using the PageModel Filter Methods in the Message.cshtml File in the Pages Folder

```
@page "/pages/message"
@model MessageModel
@using Microsoft.AspNetCore.Mvc.RazorPages
@using System.Collections.Generic
@using Microsoft.AspNetCore.Http
@using WebApp.Filters
@using Microsoft.AspNetCore.Mvc.Filters
```

```
@if (Model.Message is string) {
    @Model.Message
} else if (Model.Message is IDictionary<string, string>) {
    var dict = Model.Message as IDictionary<string, string>;
    <table class="table table-sm table-striped table-bordered">
        <thead><tr><th>Name</th><th>Value</th></tr></thead>
        <tbody>
            @foreach (var kvp in dict) {
                <tr><td>@kvp.Key</td><td>@kvp.Value</td></tr>
            }
        </tbody>
    </table>
}

@functions {

    [RequireHttps]
    //[SimpleCache]
    //[ChangePageArgs]
    public class MessageModel : PageModel {

        public object Message { get; set; } =
            $"{DateTime.Now.ToLongTimeString()}: This is the Message Razor Page";

        public void OnGet(string message1, string message2) {
            Message = $"{message1}, {message2}";
        }

        public override void OnPageHandlerExecuting(
                PageHandlerExecutingContext context) {
            if (context.HandlerArguments.ContainsKey("message1")) {
                context.HandlerArguments["message1"] = "New message";
            }
        }
    }
}
```

Request `https://localhost:44350/pages/message?message1=hello&message2=world`. The method implemented by the page model class in Listing 30-26 will produce the same result as shown in Figure 30-9.

Understanding Result Filters

Result filters are executed before and after an action result is used to generate a response, allowing responses to be modified after they have been handled by the endpoint. Here is the definition of the `IResultFilter` interface:

```
namespace Microsoft.AspNetCore.Mvc.Filters {
    public interface IResultFilter : IFilterMetadata {

        void OnResultExecuting(ResultExecutingContext context);

        void OnResultExecuted(ResultExecutedContext context);
    }
}
```

The `OnResultExecuting` method is called after the endpoint has produced an action result. This method receives context through the `ResultExecutingContext` class, which defines the properties described in Table 30-13, in addition to those defined by the `FilterContext` class.

Table 30-13. The ResultExecutingContext Class Properties

Name	Description
`Result`	This property returns the action result produced by the endpoint.
`ValueProviderFactories`	This property returns an `IList<IValueProviderFactory>`, which provides access to the objects that provide values for the model binding process.

The `OnResultExecuted` method is called after the action result has been executed to generate the response for the client. This method receives context through the `ResultExecutedContext` class, which defines the properties shown in Table 30-14, in addition to those it inherits from the `FilterContext` class.

Table 30-14. The ResultExecutedContext Class

Name	Description
`Canceled`	This property returns `true` if another filter short-circuited the filter pipeline.
`Controller`	This property returns the object that contains the endpoint.
`Exception`	This property returns an exception if one was thrown by the page handler method.
`ExceptionHandled`	This property is set to `true` to indicate that an exception thrown by the page handler has been handled by the filter.
`Result`	This property returns the action result that will be used to create a response for the client. This property is read-only.

Asynchronous result filters implement the `IAsyncResultFilter` interface, which is defined like this:

```
namespace Microsoft.AspNetCore.Mvc.Filters {

    public interface IAsyncResultFilter : IFilterMetadata {

        Task OnResultExecutionAsync(ResultExecutingContext context,
            ResultExecutionDelegate next);
    }
}
```

This interface follows the pattern established by the other filter types. The `OnResultExecutionAsync` method is invoked with a context object whose `Result` property can be used to alter the response and a delegate that will forward the response along the pipeline.

Understanding Always-Run Result Filters

Filters that implement the `IResultFilter` and `IAsyncResultFilter` interfaces are used only when a request is handled normally by the endpoint. They are not used if another filter short-circuits the pipeline or if there is an exception. Filters that need to inspect or alter the response, even when the pipeline is short-circuited, can implement the `IAlwaysRunResultFilter` or `IAsyncAlwaysRunResultFilter` interface. These interfaces derived from `IResultFilter` and `IAsyncResultFilter` but define no new features. Instead, ASP.NET Core detects the always-run interfaces and always applies the filters.

Creating a Result Filter

Add a class file named `ResultDiagnosticsAttribute.cs` to the `Filters` folder and use it to define the filter shown in Listing 30-27.

Listing 30-27. The Contents of the ResultDiagnosticsAttribute.cs File in the Filters Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace WebApp.Filters {

    public class ResultDiagnosticsAttribute : Attribute, IAsyncResultFilter {

        public async Task OnResultExecutionAsync(
                ResultExecutingContext context, ResultExecutionDelegate next) {

            if (context.HttpContext.Request.Query.ContainsKey("diag")) {
                Dictionary<string, string> diagData =
                    new Dictionary<string, string> {
                        {"Result type", context.Result.GetType().Name }
                    };
                if (context.Result is ViewResult vr) {
                    diagData["View Name"] = vr.ViewName;
                    diagData["Model Type"] = vr.ViewData.Model.GetType().Name;
                    diagData["Model Data"] = vr.ViewData.Model.ToString();
                } else if (context.Result is PageResult pr) {
                    diagData["Model Type"] = pr.Model.GetType().Name;
                    diagData["Model Data"] = pr.ViewData.Model.ToString();
                }
                context.Result = new ViewResult() {
                    ViewName = "/Views/Shared/Message.cshtml",
                    ViewData = new ViewDataDictionary(
                                   new EmptyModelMetadataProvider(),
                                   new ModelStateDictionary()) {
                        Model = diagData
                    }
                };
            }
            await next();
        }
    }
}
```

This filter examines the request to see whether it contains a query string parameter named `diag`. If it does, then the filter creates a result that displays diagnostic information instead of the output produced by the endpoint. The filter in Listing 30-27 will work with the actions defined by the `Home` controller or the `Message` Razor Page. Listing 30-28 applies the result filter to the `Home` controller.

■ **Tip** Notice that I use a fully qualified name for the view when I create the action result in Listing 30-27. This avoids a problem with filters applied to Razor Pages, where ASP.NET Core tries to execute the new result as a Razor Page and throws an exception about the model type.

Listing 30-28. Applying a Result Filter in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;
using WebApp.Filters;
using Microsoft.AspNetCore.Mvc.Filters;

namespace WebApp.Controllers {

    [HttpsOnly]
    [ResultDiagnostics]
    public class HomeController : Controller {

        public IActionResult Index() {
            return View("Message",
                "This is the Index action on the Home controller");
        }

        public IActionResult Secure() {
            return View("Message",
                "This is the Secure action on the Home controller");
        }

        //[ChangeArg]
        public IActionResult Messages(string message1, string message2 = "None") {
            return View("Message", $"{message1}, {message2}");
        }

        public override void OnActionExecuting(ActionExecutingContext context) {
            if (context.ActionArguments.ContainsKey("message1")) {
                context.ActionArguments["message1"] = "New message";
            }
        }
    }
}
```

Restart ASP.NET Core and request `https://localhost:44350/?diag`. The query string parameter will be detected by the filter, which will generate the diagnostic information shown in Figure 30-10.

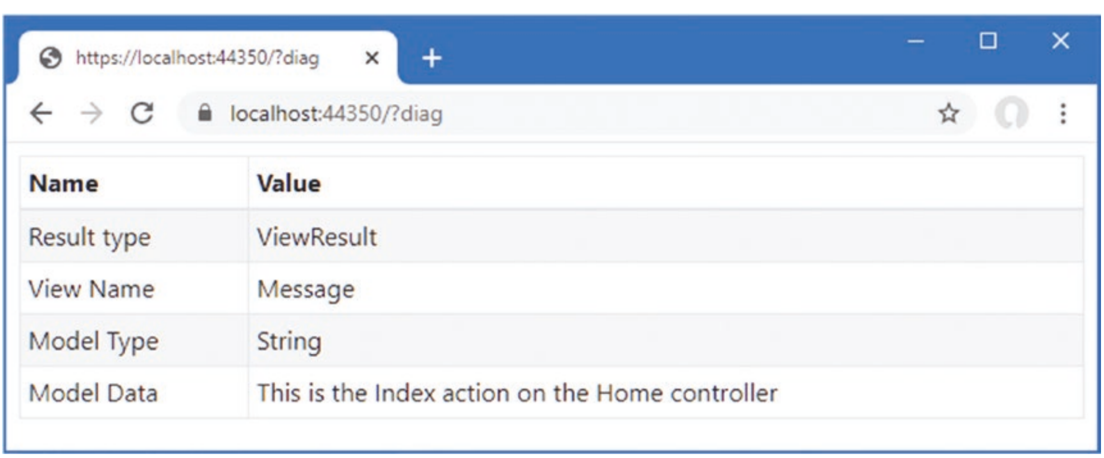

Figure 30-10. *Using a result filter*

Implementing a Result Filter Using the Attribute Base Class

The ResultFilterAttribute class is derived from Attribute and implements the IResultFilter and IAsyncResultFilter interfaces and can be used as the base class for result filters, as shown in Listing 30-29. There is no attribute base class for the always-run interfaces.

Listing 30-29. Using the Attribute Base Class in the ResultDiagnosticsAttribute.cs File in the Filters Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace WebApp.Filters {

    public class ResultDiagnosticsAttribute : ResultFilterAttribute {

        public override async Task OnResultExecutionAsync(
                ResultExecutingContext context, ResultExecutionDelegate next) {

            if (context.HttpContext.Request.Query.ContainsKey("diag")) {
                Dictionary<string, string> diagData =
                    new Dictionary<string, string> {
                        {"Result type", context.Result.GetType().Name }
                    };
                if (context.Result is ViewResult vr) {
                    diagData["View Name"] = vr.ViewName;
                    diagData["Model Type"] = vr.ViewData.Model.GetType().Name;
                    diagData["Model Data"] = vr.ViewData.Model.ToString();
                } else if (context.Result is PageResult pr) {
                    diagData["Model Type"] = pr.Model.GetType().Name;
                    diagData["Model Data"] = pr.ViewData.Model.ToString();
                }
                context.Result = new ViewResult() {
                    ViewName = "/Views/Shared/Message.cshtml",
                    ViewData = new ViewDataDictionary(
                                   new EmptyModelMetadataProvider(),
                                   new ModelStateDictionary()) {
                        Model = diagData
                    }
                };
            }
            await next();
        }
    }
}
```

Restart ASP.NET Core and request https://localhost:44350/?diag. The filter will produce the output shown in Figure 30-10.

Understanding Exception Filters

Exception filters allow you to respond to exceptions without having to write `try...catch` blocks in every action method. Exception filters can be applied to controller classes, action methods, page model classes, or handler methods. They are invoked when an exception is not handled by the endpoint or by the action, page, and result filters that have been applied to the endpoint. (Action, page, and result filters can deal with an unhandled exception by setting the `ExceptionHandled` property of their context objects to `true`.) Exception filters implement the `IExceptionFilter` interface, which is defined as follows:

```
namespace Microsoft.AspNetCore.Mvc.Filters {

    public interface IExceptionFilter : IFilterMetadata {

        void OnException(ExceptionContext context);
    }
}
```

The `OnException` method is called if an unhandled exception is encountered. The `IAsyncExceptionFilter` interface can be used to create asynchronous exception filters. Here is the definition of the asynchronous interface:

```
using System.Threading.Tasks;

namespace Microsoft.AspNetCore.Mvc.Filters {

    public interface IAsyncExceptionFilter : IFilterMetadata {

        Task OnExceptionAsync(ExceptionContext context);
    }
}
```

The `OnExceptionAsync` method is the asynchronous counterpart to the `OnException` method from the `IExceptionFilter` interface and is called when there is an unhandled exception. For both interfaces, context data is provided through the `ExceptionContext` class, which is derived from `FilterContext` and defines the additional properties shown in Table 30-15.

Table 30-15. *The ExceptionContext Properties*

Name	Description
`Exception`	This property contains any `Exception` that was thrown.
`ExceptionHandled`	This `bool` property is used to indicate if the exception has been handled.
`Result`	This property sets the `IActionResult` that will be used to generate the response.

Creating an Exception Filter

Exception filters can be created by implementing one of the filter interfaces or by deriving from the `ExceptionFilterAttribute` class, which is derived from `Attribute` and implements both the `IExceptionFilter` and `IAsyncException` filters. The most common use for an exception filter is to present a custom error page for a specific exception type in order to provide the user with more useful information than the standard error-handling capabilities can provide.

To create an exception filter, add a class file named `RangeExceptionAttribute.cs` to the `Filters` folder with the code shown in Listing 30-30.

Listing 30-30. The Contents of the RangeExceptionAttribute.cs File in the Filters Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using System;
```

```
namespace WebApp.Filters {
    public class RangeExceptionAttribute : ExceptionFilterAttribute {

        public override void OnException(ExceptionContext context) {
            if (context.Exception is ArgumentOutOfRangeException) {
                context.Result = new ViewResult() {
                    ViewName = "/Views/Shared/Message.cshtml",
                    ViewData = new ViewDataDictionary(
                        new EmptyModelMetadataProvider(),
                        new ModelStateDictionary()) {
                        Model = @"The data received by the
                                application cannot be processed"
                    }
                };
            }
        }
    }
}
```

This filter uses the ExceptionContext object to get the type of the unhandled exception and, if the type is ArgumentOutOfRangeException, creates an action result that displays a message to the user. Listing 30-31 adds an action method to the Home controller to which I have applied the exception filter.

Listing 30-31. Applying an Exception Filter in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;
using WebApp.Filters;
using Microsoft.AspNetCore.Mvc.Filters;
using System;

namespace WebApp.Controllers {

    [HttpsOnly]
    [ResultDiagnostics]
    public class HomeController : Controller {

        public IActionResult Index() {
            return View("Message",
                "This is the Index action on the Home controller");
        }

        public IActionResult Secure() {
            return View("Message",
                "This is the Secure action on the Home controller");
        }

        //[ChangeArg]
        public IActionResult Messages(string message1, string message2 = "None") {
            return View("Message", $"{message1}, {message2}");
        }

        public override void OnActionExecuting(ActionExecutingContext context) {
            if (context.ActionArguments.ContainsKey("message1")) {
                context.ActionArguments["message1"] = "New message";
            }
        }
```

```
        [RangeException]
        public ViewResult GenerateException(int? id) {
            if (id == null) {
                throw new ArgumentNullException(nameof(id));
            } else if (id > 10) {
                throw new ArgumentOutOfRangeException(nameof(id));
            } else {
                return View("Message", $"The value is {id}");
            }
        }
    }
}
```

The `GenerateException` action method relies on the default routing pattern to receive a nullable `int` value from the request URL. The action method throws an `ArgumentNullException` if there is no matching URL segment and throws an `ArgumentOutOfRangeException` if its value is greater than 50. If there is a value and it is in range, then the action method returns a `ViewResult`.

Restart ASP.NET Core and request `https://localhost:44350/Home/GenerateException/100`. The final segment will exceed the range expected by the action method, which will throw the exception type that is handled by the filter, producing the result shown in Figure 30-11. If you request `/Home/GenerateException`, then the exception thrown by the action method won't be handled by the filter, and the default error handling will be used.

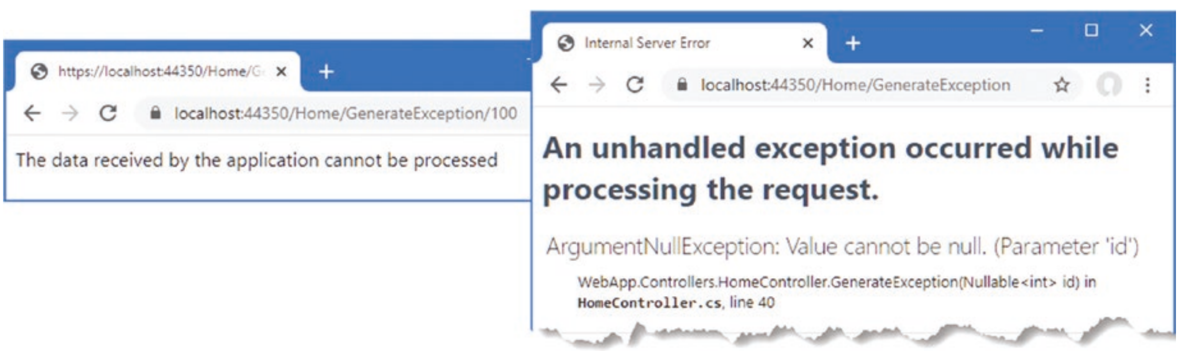

Figure 30-11. *Using an exception filter*

Managing the Filter Lifecycle

By default, ASP.NET Core manages the filter objects it creates and will reuse them for subsequent requests. This isn't always the desired behavior, and in the sections that follow, I describe different ways to take control of how filters are created. To create a filter that will show the lifecycle, add a class file called `GuidResponseAttribute.cs` to the `Filters` folder, and use it to define the filter shown in Listing 30-32.

Listing 30-32. The Contents of the GuidResponseAttribute.cs File in the Filters Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
```

```
namespace WebApp.Filters {

    [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,
         AllowMultiple = true)]
    public class GuidResponseAttribute : Attribute, IAsyncAlwaysRunResultFilter {
        private int counter = 0;
        private string guid = Guid.NewGuid().ToString();

        public async Task OnResultExecutionAsync(ResultExecutingContext context,
            ResultExecutionDelegate next) {

            Dictionary<string, string> resultData;
            if (context.Result is ViewResult vr
                && vr.ViewData.Model is Dictionary<string, string> data) {
                    resultData = data;
            } else {
                resultData = new Dictionary<string, string>();
                context.Result = new ViewResult() {
                    ViewName = "/Views/Shared/Message.cshtml",
                    ViewData = new ViewDataDictionary(
                                        new EmptyModelMetadataProvider(),
                                        new ModelStateDictionary()) {
                        Model = resultData
                    }
                };
            }
            while (resultData.ContainsKey($"Counter_{counter}")) {
                counter++;
            }
            resultData[$"Counter_{counter}"] = guid;
            await next();
        }
    }
}
```

This result filter replaces the action result produced by the endpoint with one that will render the Message view and display a unique GUID value. The filter is configured so that it can be applied more than once to the same target and will add a new message if a filter earlier in the pipeline has created a suitable result. Listing 30-33 applies the filter twice to the Home controller. (I have also removed all but one of the action methods for brevity.)

Listing 30-33. Applying a Filter in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;
using WebApp.Filters;
using Microsoft.AspNetCore.Mvc.Filters;
using System;

namespace WebApp.Controllers {

    [HttpsOnly]
    [ResultDiagnostics]
    [GuidResponse]
    [GuidResponse]
    public class HomeController : Controller {
```

```
        public IActionResult Index() {
            return View("Message",
                "This is the Index action on the Home controller");
        }
    }
}
```

To confirm that the filter is being reused, restart ASP.NET Core and request `https://localhost:44350/?diag`. The response will contain GUID values from the two `GuidResponse` filter attributes. Two instances of the filter have been created to handle the request. Reload the browser, and you will see the same GUID values displayed, indicating that the filter objects created to handle the first request have been reused (Figure 30-12).

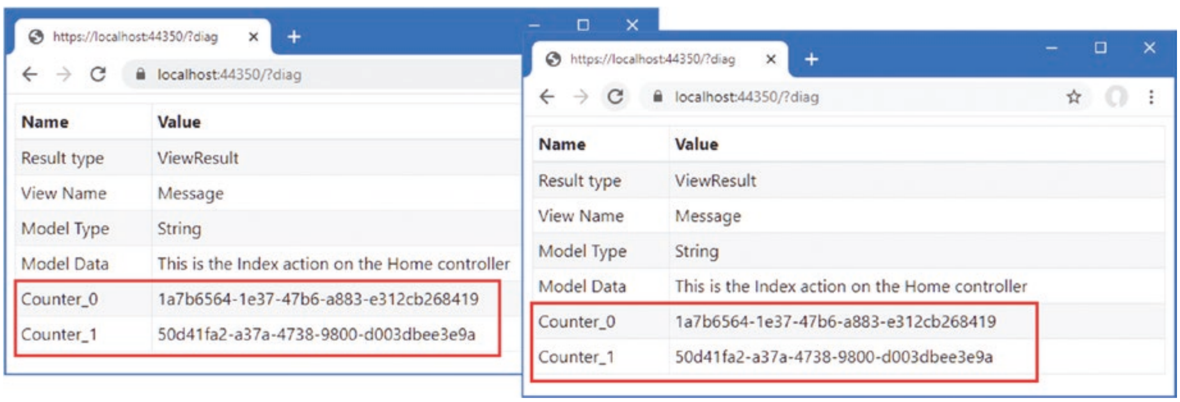

Figure 30-12. *Demonstrating filter reuse*

Creating Filter Factories

Filters can implement the `IFilterFactory` interface to take responsibility for creating instances of filters and specify whether those instances can be reused. The `IFilterFactory` interface defines the members described in Table 30-16.

Table 30-16. *The IFilterFactory Members*

Name	Description
`IsReusable`	This `bool` property indicates whether instances of the filter can be reused.
`CreateInstance(serviceProvider)`	This method is invoked to create new instances of the filter and is provided with an `IServiceProvider` object.

Listing 30-34 implements the `IFilterFactory` interface and returns `false` for the `IsReusable` property, which prevents the filter from being reused.

Listing 30-34. Implementing an Interface in the GuidResponseAttribute.cs File in the Filters Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.Extensions.DependencyInjection;
```

```
namespace WebApp.Filters {

    [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,
         AllowMultiple = true)]
    public class GuidResponseAttribute : Attribute,
            IAsyncAlwaysRunResultFilter, IFilterFactory {
        private int counter = 0;
        private string guid = Guid.NewGuid().ToString();

        public bool IsReusable => false;

        public IFilterMetadata CreateInstance(IServiceProvider serviceProvider) {
            return ActivatorUtilities
                .GetServiceOrCreateInstance<GuidResponseAttribute>(serviceProvider);
        }

        public async Task OnResultExecutionAsync(ResultExecutingContext context,
            ResultExecutionDelegate next) {

            Dictionary<string, string> resultData;
            if (context.Result is ViewResult vr
                && vr.ViewData.Model is Dictionary<string, string> data) {
                    resultData = data;
            } else {
                resultData = new Dictionary<string, string>();
                context.Result = new ViewResult() {
                    ViewName = "/Views/Shared/Message.cshtml",
                    ViewData = new ViewDataDictionary(
                                        new EmptyModelMetadataProvider(),
                                        new ModelStateDictionary()) {
                        Model = resultData
                    }
                };
            }
            while (resultData.ContainsKey($"Counter_{counter}")) {
                counter++;
            }
            resultData[$"Counter_{counter}"] = guid;
            await next();
        }
    }
}
```

I create new filter objects using the `GetServiceOrCreateInstance` method, defined by the `ActivatorUtilities` class in the `Microsoft.Extensions.DependencyInjection` namespace. Although you can use the new keyword to create a filter, this approach will resolve any dependencies on services that are declared through the filter's constructor.

To see the effect of implementing the `IFilterFactory` interface, restart ASP.NET Core and request `https://localhost:44350/?diag`. Reload the browser, and each time the request is handled, new filters will be created, and new GUIDs will be displayed, as shown in Figure 30-13.

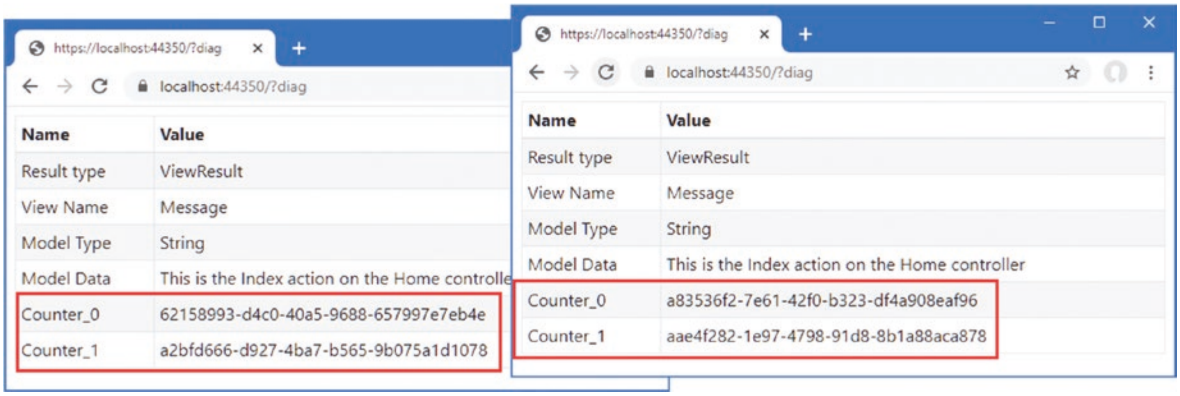

Figure 30-13. *Preventing filter reuse*

Using Dependency Injection Scopes to Manage Filter Lifecycles

Filters can be registered as services, which allows their lifecycle to be controlled through dependency injection, which I described in Chapter 14. Listing 30-35 registers the GuidResponse filter as a scoped service.

Listing 30-35. Creating a Filter Service in the Startup.cs File in the WebApp Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using WebApp.Models;
using Microsoft.AspNetCore.Antiforgery;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using WebApp.Filters;

namespace WebApp {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        public IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<DataContext>(opts => {
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:ProductConnection"]);
                opts.EnableSensitiveDataLogging(true);
            });
            services.AddControllersWithViews().AddRazorRuntimeCompilation();
            services.AddRazorPages().AddRazorRuntimeCompilation();
            services.AddSingleton<CitiesData>();

            services.Configure<AntiforgeryOptions>(opts => {
                opts.HeaderName = "X-XSRF-TOKEN";
            });
```

```
            services.Configure<MvcOptions>(opts => opts.ModelBindingMessageProvider
                .SetValueMustNotBeNullAccessor(value => "Please enter a value"));

            services.AddScoped<GuidResponseAttribute>();
        }

        public void Configure(IApplicationBuilder app, DataContext context,
                IAntiforgery antiforgery) {

            // ...statements omitted for brevity...
        }
    }
}
```

By default, ASP.NET Core creates a scope for each request, which means that a single instance of the filter will be created for each request. To see the effect, restart ASP.NET Core and request `https://localhost:44350/?diag`. Both attributes applied to the Home controller are processed using the same instance of the filter, which means that both GUIDs in the response are the same. Reload the browser; a new scope will be created, and a new filter object will be used, as shown in Figure 30-14.

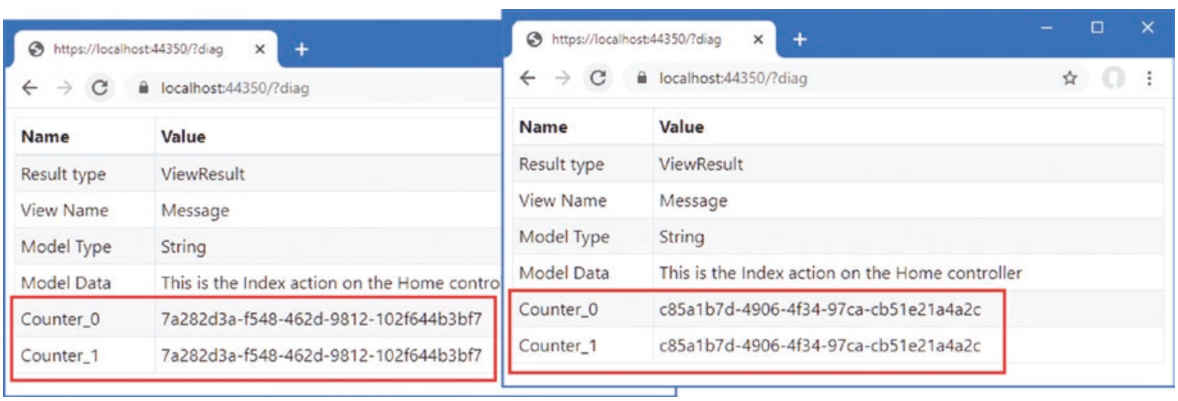

Figure 30-14. *Using dependency injection to manage filters*

USING FILTERS AS SERVICES WITHOUT THE IFILTERFACTORY INTERFACE

The change in lifecycle took effect immediately in this example because I used the `ActivatorUtilities.GetServiceOrCreateInstance` method to create the filter object when I implemented the `IFilterFactory` interface. This method will check to see whether there is a service available for the requested type before invoking its constructor. If you want to use filters as services without implementing `IFilterFactory` and using `ActivatorUtilities`, you can apply the filter using the `ServiceFilter` attribute, like this:

```
...
[ServiceFilter(typeof(GuidResponseAttribute))]
...
```

ASP.NET Core will create the filter object from the service and apply it to the request. Filters that are applied in this way do not have to be derived from the `Attribute` class.

Creating Global Filters

Global filters are applied to every request that ASP.NET Core handles, which means they don't have to be applied to individual controllers or Razor Pages. Any filter can be used as a global filter; however, action filters will be applied to requests only where the endpoint is an action method, and page filters will be applied to requests only where the endpoint is a Razor Page.

Global filters are set up using the options pattern in the `Startup` class, as shown in Listing 30-36.

Listing 30-36. Creating a Global Filter in the Startup.cs File in the WebApp Folder

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddDbContext<DataContext>(opts => {
        opts.UseSqlServer(Configuration[
            "ConnectionStrings:ProductConnection"]);
        opts.EnableSensitiveDataLogging(true);
    });
    services.AddControllersWithViews().AddRazorRuntimeCompilation();
    services.AddRazorPages().AddRazorRuntimeCompilation();
    services.AddSingleton<CitiesData>();

    services.Configure<AntiforgeryOptions>(opts => {
        opts.HeaderName = "X-XSRF-TOKEN";
    });

    services.Configure<MvcOptions>(opts => opts.ModelBindingMessageProvider
        .SetValueMustNotBeNullAccessor(value => "Please enter a value"));

    services.AddScoped<GuidResponseAttribute>();
    services.Configure<MvcOptions>(opts => opts.Filters.Add<HttpsOnlyAttribute>());
}
...
```

The `MvcOptions.Filters` property returns a collection to which filters are added to apply them globally, either using the `Add<T>` method or using the `AddService<T>` method for filters that are also services. There is also an `Add` method without a generic type argument that can be used to register a specific object as a global filter.

The statement in Listing 30-36 registers the `HttpsOnly` filter I created earlier in the chapter, which means that it no longer needs to be applied directly to individual controllers or Razor Pages, so Listing 30-37 removes the filter from the `Home` controller.

■ **Note** Notice that I have disabled the `GuidResponse` filter in Listing 30-37. This is an always-run result filter and will replace the result generated by the global filter.

Listing 30-37. Removing a Filter in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;
using WebApp.Filters;
using Microsoft.AspNetCore.Mvc.Filters;
using System;

namespace WebApp.Controllers {

    //[HttpsOnly]
    [ResultDiagnostics]
    //[GuidResponse]
    //[GuidResponse]
```

```
    public class HomeController : Controller {

        public IActionResult Index() {
            return View("Message",
                "This is the Index action on the Home controller");
        }
    }
}
```

Restart ASP.NET Core and request `http://localhost:5000` to confirm that the HTTPS-only policy is being applied even though the attribute is no longer used to decorate the controller. The global authorization filter will short-circuit the filter pipeline and produce the response shown in Figure 30-15.

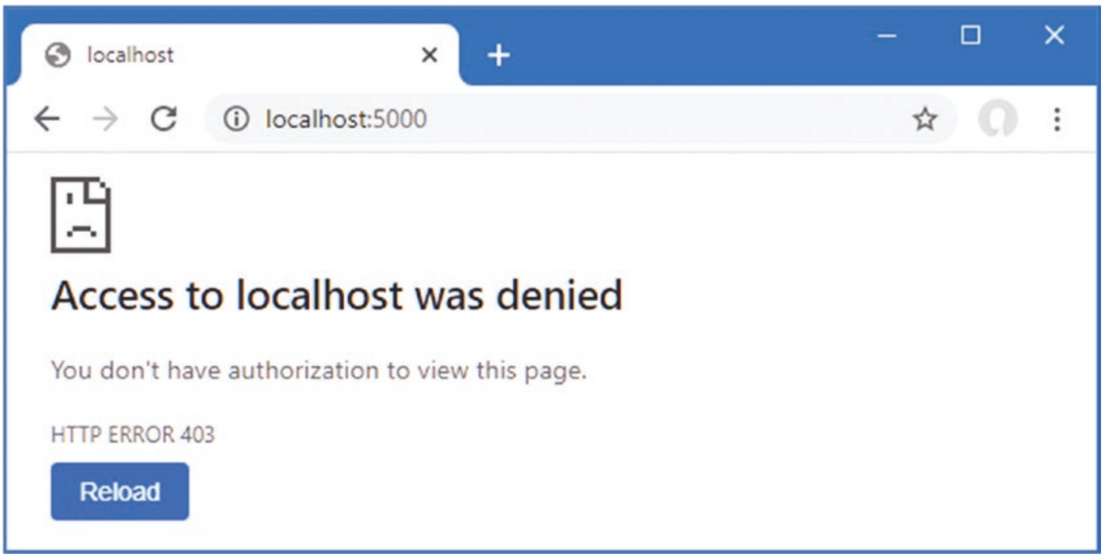

Figure 30-15. *Using a global filter*

Understanding and Changing Filter Order

Filters run in a specific sequence: authorization, resource, action, or page, and then result. But if there are multiple filters of a given type, then the order in which they are applied is driven by the scope through which the filters have been applied.

To demonstrate how this works, add a class file named `MessageAttribute.cs` to the `Filters` folder and use it to define the filter shown in Listing 30-38.

Listing 30-38. The Contents of the MessageAttribute.cs File in the Filters Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
```

```
namespace WebApp.Filters {

    [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,
        AllowMultiple = true)]
    public class MessageAttribute : Attribute, IAsyncAlwaysRunResultFilter {
        private int counter = 0;
        private string msg;

        public MessageAttribute(string message) => msg = message;

        public async Task OnResultExecutionAsync(ResultExecutingContext context,
               ResultExecutionDelegate next) {
            Dictionary<string, string> resultData;
            if (context.Result is ViewResult vr
                && vr.ViewData.Model is Dictionary<string, string> data) {
                resultData = data;
            } else {
                resultData = new Dictionary<string, string>();
                context.Result = new ViewResult() {
                    ViewName = "/Views/Shared/Message.cshtml",
                    ViewData = new ViewDataDictionary(
                                    new EmptyModelMetadataProvider(),
                                    new ModelStateDictionary()) {
                        Model = resultData
                    }
                };
            }
            while (resultData.ContainsKey($"Message_{counter}")) {
                counter++;
            }
            resultData[$"Message_{counter}"] = msg;
            await next();
        }
    }
}
```

This result filter uses techniques shown in earlier examples to replace the result from the endpoint and allows multiple filters to build up a series of messages that will be displayed to the user. Listing 30-39 applies several instances of the `Message` filter to the `Home` controller.

Listing 30-39. Applying a Filter in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;
using WebApp.Filters;
using Microsoft.AspNetCore.Mvc.Filters;
using System;

namespace WebApp.Controllers {

    [Message("This is the controller-scoped filter")]
    public class HomeController : Controller {

        [Message("This is the first action-scoped filter")]
        [Message("This is the second action-scoped filter")]
        public IActionResult Index() {
            return View("Message",
```

```
                "This is the Index action on the Home controller");
        }
    }
}
```

Listing 30-40 registers the `Message` filter globally.

Listing 30-40. Creating a Global Filter in the Startup.cs File in the WebApp Folder

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddDbContext<DataContext>(opts => {
        opts.UseSqlServer(Configuration[
            "ConnectionStrings:ProductConnection"]);
        opts.EnableSensitiveDataLogging(true);
    });
    services.AddControllersWithViews().AddRazorRuntimeCompilation();
    services.AddRazorPages().AddRazorRuntimeCompilation();
    services.AddSingleton<CitiesData>();

    services.Configure<AntiforgeryOptions>(opts => {
        opts.HeaderName = "X-XSRF-TOKEN";
    });

    services.Configure<MvcOptions>(opts => opts.ModelBindingMessageProvider
        .SetValueMustNotBeNullAccessor(value => "Please enter a value"));

    services.AddScoped<GuidResponseAttribute>();
    services.Configure<MvcOptions>(opts => {
        opts.Filters.Add<HttpsOnlyAttribute>();
        opts.Filters.Add(new MessageAttribute("This is the globally-scoped filter"));
    });
}
...
```

There are four instances of the same filter. To see the order in which they are applied, restart ASP.NET Core and request `https://localhost:44350`, which will produce the response shown in Figure 30-16.

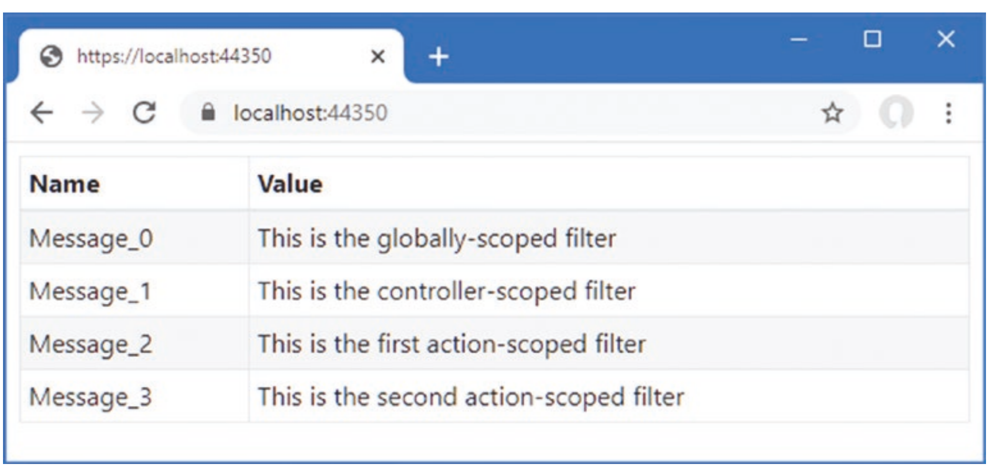

Figure 30-16. *Applying the same filter in different scopes*

By default, ASP.NET Core runs global filters, then filters applied to controllers or page model classes, and finally filters applied to action or handler methods.

Changing Filter Order

The default order can be changed by implementing the `IOrderedFilter` interface, which ASP.NET Core looks for when it is working out how to sequence filters. Here is the definition of the interface:

```
namespace Microsoft.AspNetCore.Mvc.Filters {

    public interface IOrderedFilter : IFilterMetadata {
        int Order { get; }
    }
}
```

The `Order` property returns an `int` value, and filters with low values are applied before those with higher `Order` values. In Listing 30-41, I have implemented the interface in the `Message` filter and defined a constructor argument that will allow the value for the `Order` property to be specified when the filter is applied.

Listing 30-41. Adding Ordering Support in the MessageAttribute.cs File in the Filters Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace WebApp.Filters {

    [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,
        AllowMultiple = true)]
    public class MessageAttribute : Attribute, IAsyncAlwaysRunResultFilter,
            IOrderedFilter {
        private int counter = 0;
        private string msg;

        public MessageAttribute(string message) => msg = message;

        public int Order { get; set; }

        public async Task OnResultExecutionAsync(ResultExecutingContext context,
               ResultExecutionDelegate next) {

            // ...statements omitted for brevity...
        }
    }
}
```

In Listing 30-42, I have used the constructor argument to change the order in which the filters are applied.

Listing 30-42. Setting Filter Order in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;
using WebApp.Filters;
```

```
using Microsoft.AspNetCore.Mvc.Filters;
using System;

namespace WebApp.Controllers {

    [Message("This is the controller-scoped filter", Order = 10)]
    public class HomeController : Controller {

        [Message("This is the first action-scoped filter", Order = 1)]
        [Message("This is the second action-scoped filter", Order = -1)]
        public IActionResult Index() {
            return View("Message",
                "This is the Index action on the Home controller");
        }
    }
}
```

`Order` values can be negative, which is a helpful way of ensuring that a filter is applied before any global filters with the default order (although you can also set the order when creating global filters, too). Restart ASP.NET Core and request `https://localhost:44350` to see the new filter order, which is shown in Figure 30-17.

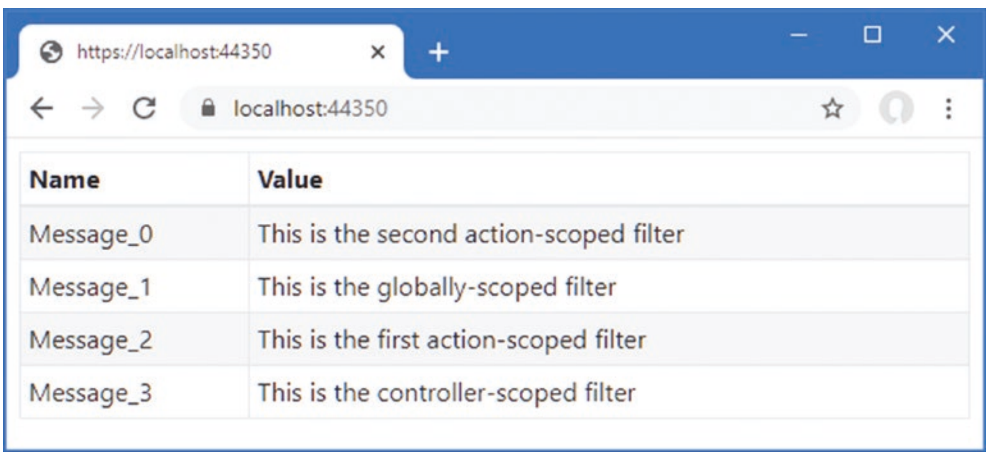

Figure 30-17. *Changing filter order*

Summary

In this chapter, I described the ASP.NET Core filter feature and explained how it can be used to alter requests and results for specific endpoints. I described the different types of filters and demonstrated how to create and apply each of them. I also showed you how to manage the lifecycle of filters and control the order in which they are executed. In the next chapter, I show you how to combine the features described in this part of the book to create form applications.

CHAPTER 31

Creating Form Applications

The previous chapters have focused on individual features that deal with one aspect of HTML forms, and it can sometimes be difficult to see how they fit together to perform common tasks. In this chapter, I go through the process of creating controllers, views, and Razor Pages that support an application with create, read, update, and delete (CRUD) functionality. There are no new features described in this chapter, and the objective is to demonstrate how features such as tag helpers, model binding, and model validation can be used in conjunction with Entity Framework Core.

Preparing for This Chapter

This chapter uses the WebApp project from Chapter 30. To prepare for this chapter, replace the contents of the `HomeController.cs` file in the `Controllers` folder with those shown in Listing 31-1.

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-3`. See Chapter 1 for how to get help if you have problems running the examples.

Listing 31-1. The Contents of the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Threading.Tasks;
using WebApp.Models;

namespace WebApp.Controllers {

    [AutoValidateAntiforgeryToken]
    public class HomeController : Controller {
        private DataContext context;

        private IEnumerable<Category> Categories => context.Categories;
        private IEnumerable<Supplier> Suppliers => context.Suppliers;

        public HomeController(DataContext data) {
            context = data;
        }

        public IActionResult Index() {
            return View(context.Products.
                Include(p => p.Category).Include(p => p.Supplier));
        }
    }
}
```


A. Freeman, *Pro ASP.NET Core 3*, https://doi.org/10.1007/978-1-4842-5440-0_31

Create the `Views/HomeIndex.cshtml`,

Listing 31-2. The Contents of the Index.cshtml File in the Views/Home Folder

```
@model IEnumerable<Product>
@{ Layout = "_SimpleLayout"; }

<h4 class="bg-primary text-white text-center p-2">Products</h4>
<table class="table table-sm table-bordered table-striped">
    <thead>
        <tr>
            <th>ID</th><th>Name</th><th>Price</th><th>Category</th><th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (Product p in Model) {
            <tr>
                <td>@p.ProductId</td>
                <td>@p.Name</td>
                <td>@p.Price</td>
                <td>@p.Category.Name</td>
                <td class="text-center">
                    <a asp-action="Details" asp-route-id="@p.ProductId"
                       class="btn btn-sm btn-info">Details</a>
                    <a asp-action="Edit" asp-route-id="@p.ProductId"
                       class="btn btn-sm btn-warning">Edit</a>
                    <a asp-action="Delete" asp-route-id="@p.ProductId"
                       class="btn btn-sm btn-danger">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
<a asp-action="Create" class="btn btn-primary">Create</a>
```

Next, update the `Product` class as shown in Listing 31-3 to change the validation constraints to remove the model-level checking and disable remote validation.

Listing 31-3. Changing Validation in the Product.cs File in the Models Folder

```
using System.ComponentModel.DataAnnotations.Schema;
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using WebApp.Validation;
using Microsoft.AspNetCore.Mvc;

namespace WebApp.Models {

    //[PhraseAndPrice(Phrase = "Small", Price = "100")]
    public class Product {

        public long ProductId { get; set; }

        [Required]
        [Display(Name = "Name")]
        public string Name { get; set; }
```

```
        [Column(TypeName = "decimal(8, 2)")]
        [Required(ErrorMessage = "Please enter a price")]
        [Range(1, 999999, ErrorMessage = "Please enter a positive price")]
        public decimal Price { get; set; }

        [PrimaryKey(ContextType = typeof(DataContext),
            DataType = typeof(Category))]
        //[Remote("CategoryKey", "Validation",
        //     ErrorMessage = "Enter an existing key")]
        public long CategoryId { get; set; }
        public Category Category { get; set; }

        [PrimaryKey(ContextType = typeof(DataContext),
            DataType = typeof(Category))]
        //[Remote("SupplierKey", "Validation",
        //    ErrorMessage = "Enter an existing key")]
        public long SupplierId { get; set; }
        public Supplier Supplier { get; set; }
    }
}
```

Finally, disable the global filters in the Startup class, as shown in Listing 31-4.

Listing 31-4. Disabling Filters in the Startup.cs File in the WebApp Folder

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddDbContext<DataContext>(opts => {
        opts.UseSqlServer(Configuration[
            "ConnectionStrings:ProductConnection"]);
        opts.EnableSensitiveDataLogging(true);
    });
    services.AddControllersWithViews().AddRazorRuntimeCompilation();
    services.AddRazorPages().AddRazorRuntimeCompilation();
    services.AddSingleton<CitiesData>();

    services.Configure<AntiforgeryOptions>(opts => {
        opts.HeaderName = "X-XSRF-TOKEN";
    });

    services.Configure<MvcOptions>(opts => opts.ModelBindingMessageProvider
        .SetValueMustNotBeNullAccessor(value => "Please enter a value"));

    services.AddScoped<GuidResponseAttribute>();
    //services.Configure<MvcOptions>(opts => {
    //    opts.Filters.Add<HttpsOnlyAttribute>();
    //    opts.Filters.Add(new MessageAttribute(
    //         "This is the globally-scoped filter"));
    //});
}
...
```

Dropping the Database

Open a new PowerShell command prompt, navigate to the folder that contains the `WebApp.csproj` file, and run the command shown in Listing 31-5 to drop the database.

Listing 31-5. Dropping the Database

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

Running the Example Application

Select Start Without Debugging or Run Without Debugging from the Debug menu or use the PowerShell command prompt to run the command shown in Listing 31-6.

Listing 31-6. Running the Example Application

```
dotnet run
```

Use a browser to request `http://localhost:5000/controllers`, which will display a list of products, as shown in Figure 31-1. There are anchor elements styled to appear as buttons, but these will not work until later when I add the features to create, edit, and delete objects.

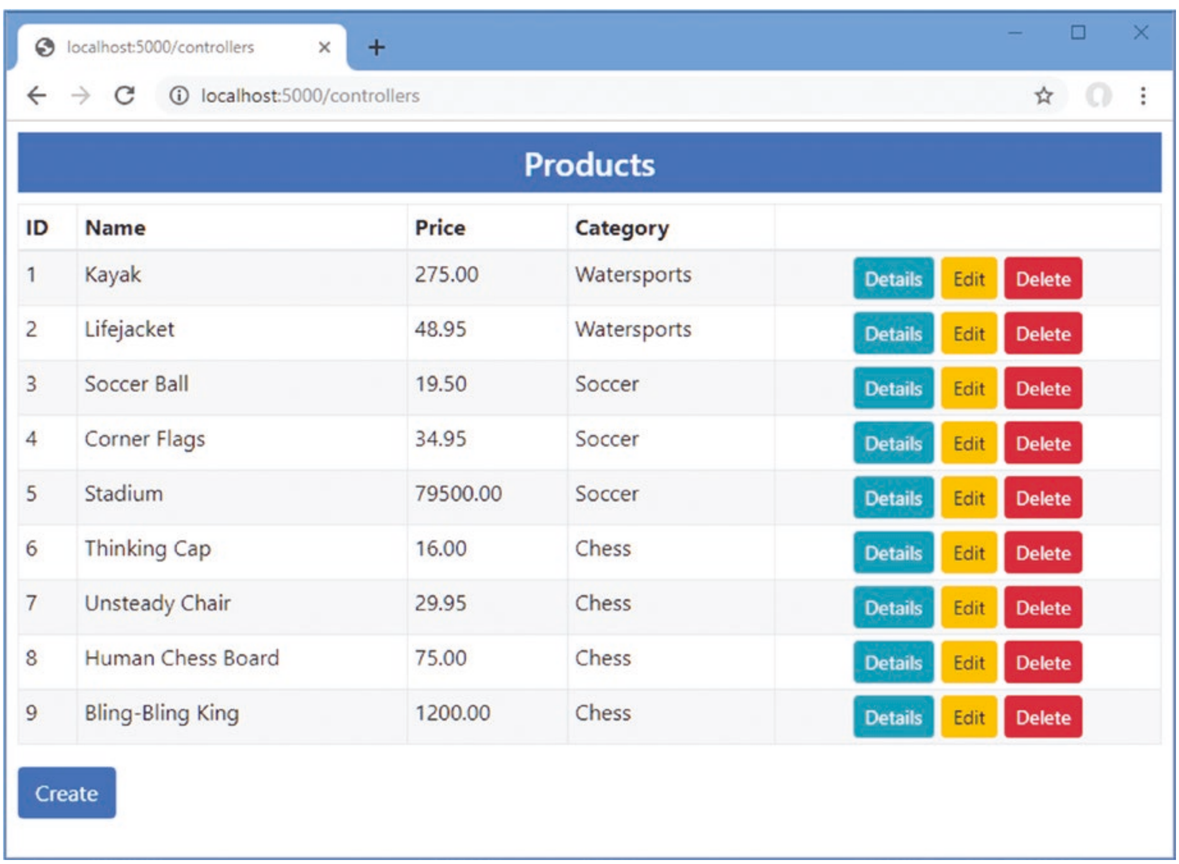

Figure 31-1. *Running the example application*

Creating an MVC Forms Application

In the sections that follow, I show you how to perform the core data operations using MVC controllers and views. Later in the chapter, I create the same functionality using Razor Pages.

Preparing the View Model and the View

I am going to define a single form that will be used for multiple operations, configured through its view model class. To create the view model class, add a Class File named `ProductViewModel.cs` to the `Models` folder and add the code shown in Listing 31-7.

Listing 31-7. The Contents of the ProductViewModel.cs File in the Models Folder

```
using System.Collections.Generic;
using System.Linq;

namespace WebApp.Models {

    public class ProductViewModel {
        public Product Product { get; set; }
        public string Action { get; set; } = "Create";
        public bool ReadOnly { get; set; } = false;
        public string Theme { get; set; } = "primary";
        public bool ShowAction { get; set; } = true;
        public IEnumerable<Category> Categories { get; set; }
            = Enumerable.Empty<Category>();
        public IEnumerable<Supplier> Suppliers { get; set; }
            = Enumerable.Empty<Supplier>();
    }
}
```

This class will allow the controller to pass data and display settings to its view. The `Product` property provides the data to display, and the `Categories` and `Suppliers` properties provide access to the `Category` and `Suppliers` objects when they are required. The other properties configure aspects of how the content is presented to the user: the `Action` property specifies the name of the action method for the current task, the `ReadOnly` property specifies whether the user can edit the data, the `Theme` property specifies the Bootstrap theme for the content, and the `ShowAction` property is used to control the visibility of the button that submits the form.

To create the view that will allow the user to interact with the application's data, add a Razor View named `ProductEditor.cshtml` to the `Views/Home` folder with the content shown in Listing 31-8.

Listing 31-8. The Contents of the ProductEditor.cshtml File in the Views/Home Folder

```
@model ProductViewModel
@{ Layout = "_SimpleLayout"; }

<partial name="_Validation" />

<h5 class="bg-@Model.Theme text-white text-center p-2">@Model.Action</h5>

<form asp-action="@Model.Action" method="post">
    <div class="form-group">
        <label asp-for="Product.ProductId"></label>
        <input class="form-control" asp-for="Product.ProductId" readonly />
    </div>
    <div class="form-group">
        <label asp-for="Product.Name"></label>
```

```
        <div>
            <span asp-validation-for="Product.Name" class="text-danger"></span>
        </div>
        <input class="form-control" asp-for="Product.Name"
               readonly="@Model.ReadOnly" />
    </div>
    <div class="form-group">
        <label asp-for="Product.Price"></label>
        <div>
            <span asp-validation-for="Product.Price" class="text-danger"></span>
        </div>
        <input class="form-control" asp-for="Product.Price"
               readonly="@Model.ReadOnly" />
    </div>
    <div class="form-group">
        <label asp-for="Product.CategoryId">Category</label>
        <div>
            <span asp-validation-for="Product.CategoryId" class="text-danger"></span>
        </div>
        <select asp-for="Product.CategoryId" class="form-control"
                disabled="@Model.ReadOnly"
                asp-items="@(new SelectList(Model.Categories,
                    "CategoryId", "Name"))">
            <option value="" disabled selected>Choose a Category</option>
        </select>
    </div>
    <div class="form-group">
        <label asp-for="Product.SupplierId">Supplier</label>
        <div>
            <span asp-validation-for="Product.SupplierId" class="text-danger"></span>
        </div>
        <select asp-for="Product.SupplierId" class="form-control"
                disabled="@Model.ReadOnly"
                asp-items="@(new SelectList(Model.Suppliers,
                    "SupplierId", "Name"))">
            <option value="" disabled selected>Choose a Supplier</option>
        </select>
    </div>
    @if (Model.ShowAction) {
        <button class="btn btn-@Model.Theme" type="submit">@Model.Action</button>
    }
    <a class="btn btn-secondary" asp-action="Index">Back</a>
</form>
```

This view can look complicated, but it combines only the features you have seen in earlier chapters and will become clearer once you see it in action. The model for this view is a `ProductViewModel` object, which provides both the data that is displayed to the user and some direction about how that data should be presented.

For each of the properties defined by the `Product` class, the view contains a set of elements: a `label` element that describes the property, an `input` or `select` element that allows the value to be edited, and a `span` element that will display validation messages. Each of the elements is configured with the `asp-for` attribute, which ensures tag helpers will transform the elements for each property. There are `div` elements to define the view structure, and all the elements are members of Bootstrap CSS classes to style the form.

Reading Data

The simplest operation is reading data from the database and presenting it to the user. In most applications, this will allow the user to see additional details that are not present in the list view. Each task performed by the application will require a different set of `ProductViewModel` properties. To manage these combinations, add a class file named `ViewModelFactory.cs` to the `Models` folder with the code shown in Listing 31-9.

Listing 31-9. The Contents of the ViewModelFactory.cs File in the Models Folder

```
using System.Collections.Generic;
using System.Linq;

namespace WebApp.Models {

    public static class ViewModelFactory {

        public static ProductViewModel Details(Product p) {
            return new ProductViewModel {
                Product = p, Action = "Details",
                ReadOnly = true, Theme = "info", ShowAction = false,
                Categories = p == null ? Enumerable.Empty<Category>()
                    : new List<Category> { p.Category },
                Suppliers = p == null ? Enumerable.Empty<Supplier>()
                    : new List<Supplier> { p.Supplier},
            };
        }
    }
}
```

The `Details` method produces a `ProductViewModel` object configured for viewing an object. When the user views the details, the category and supplier details will be read-only, which means that I need to provide only the current category and supplier information.

Next, add an action method to the `Home` controller that uses the `ViewModelFactory.Details` method to create a `ProductViewModel` object and display it to the user with the `ProductEditor` view, as shown in Listing 31-10.

Listing 31-10. Adding an Action Method in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Threading.Tasks;
using WebApp.Models;

namespace WebApp.Controllers {

    [AutoValidateAntiforgeryToken]
    public class HomeController : Controller {
        private DataContext context;

        private IEnumerable<Category> Categories => context.Categories;
        private IEnumerable<Supplier> Suppliers => context.Suppliers;

        public HomeController(DataContext data) {
            context = data;
        }

        public IActionResult Index() {
            return View(context.Products.
                Include(p => p.Category).Include(p => p.Supplier));
        }

        public async Task<IActionResult> Details(long id) {
            Product p = await context.Products.
                Include(p => p.Category).Include(p => p.Supplier)
                .FirstOrDefaultAsync(p => p.ProductId == id);
```

```
            ProductViewModel model = ViewModelFactory.Details(p);
            return View("ProductEditor", model);
        }
    }
}
```

The action method uses the id parameter, which will be model bound from the routing data, to query the database and passes the Product object to the ViewModelFactory.Details method. Most of the operations are going to require the Category and Supplier data, so I have added properties that provide direct access to the data.

To test the details feature, restart ASP.NET Core and request http://localhost:5000/controllers. Click one of the Details buttons, and you will see the selected object presented in read-only form using the ProductEditor view, as shown in Figure 31-2.

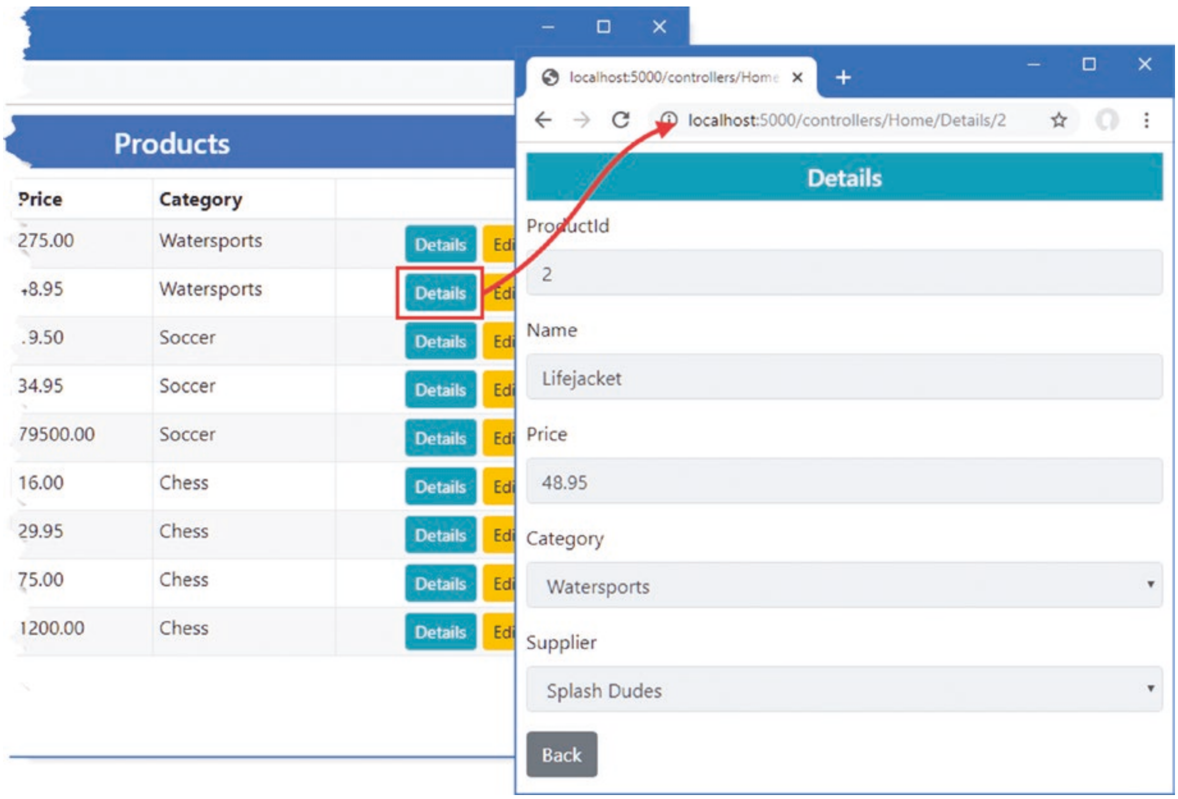

Figure 31-2. *Viewing data*

If the user navigates to a URL that doesn't correspond to an object in the database, such as http://localhost:5000/controllers/Home/Details/100, for example, then an empty form will be displayed.

Creating Data

Creating data relies on model binding to get the form data from the request and relies on validation to ensure the data can be stored in the database. The first step is to add a factory method that will create the view model object for creating data, as shown in Listing 31-11.

Listing 31-11. Adding a Method in the ViewModelFactory.cs File in the Models Folder

```
using System.Collections.Generic;
using System.Linq;

namespace WebApp.Models {
```

```
    public static class ViewModelFactory {

        public static ProductViewModel Details(Product p) {
            return new ProductViewModel {
                Product = p, Action = "Details",
                ReadOnly = true, Theme = "info", ShowAction = false,
                Categories = p == null ? Enumerable.Empty<Category>()
                    : new List<Category> { p.Category },
                Suppliers = p == null ? Enumerable.Empty<Supplier>()
                    : new List<Supplier> { p.Supplier},
            };
        }

        public static ProductViewModel Create(Product product,
            IEnumerable<Category> categories, IEnumerable<Supplier> suppliers) {
            return new ProductViewModel {
                Product = product, Categories = categories, Suppliers = suppliers
            };
        }
    }
}
```

The defaults I used for the ProductViewModel properties were set for creating data, so the Create method in Listing 31-11 sets only the Product, Categories, and Suppliers properties. Listing 31-12 adds the action methods that will create data to the Home controller.

Listing 31-12. Adding Actions in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Threading.Tasks;
using WebApp.Models;

namespace WebApp.Controllers {

    [AutoValidateAntiforgeryToken]
    public class HomeController : Controller {
        private DataContext context;

        private IEnumerable<Category> Categories => context.Categories;
        private IEnumerable<Supplier> Suppliers => context.Suppliers;

        public HomeController(DataContext data) {
            context = data;
        }

        public IActionResult Index() {
            return View(context.Products.
                Include(p => p.Category).Include(p => p.Supplier));
        }

        public async Task<IActionResult> Details(long id) {
            Product p = await context.Products.
                Include(p => p.Category).Include(p => p.Supplier)
                .FirstOrDefaultAsync(p => p.ProductId == id);
            ProductViewModel model = ViewModelFactory.Details(p);
            return View("ProductEditor", model);
        }
```

```
        public IActionResult Create() {
            return View("ProductEditor",
                ViewModelFactory.Create(new Product(), Categories, Suppliers));
        }

        [HttpPost]
        public async Task<IActionResult> Create([FromForm] Product product) {
            if (ModelState.IsValid) {
                product.ProductId = default;
                product.Category = default;
                product.Supplier = default;
                context.Products.Add(product);
                await context.SaveChangesAsync();
                return RedirectToAction(nameof(Index));
            }
            return View("ProductEditor",
                ViewModelFactory.Create(product, Categories, Suppliers));
        }
    }
}
```

There are two Create methods, which are differentiated by the HttpPost attribute and method parameters. HTTP GET requests will be handled by the first method, which selects the ProductEditor view and provides it with a ProductViewModel object. When the user submits the form, it will be received by the second method, which relies on model binding to receive the data and model validation to ensure the data is valid.

If the data passes validation, then I prepare the object for storage in the database by resetting three properties, like this:

```
...
product.ProductId = default;
product.Category = default;
product.Supplier = default;
...
```

Entity Framework Core configures the database so that primary keys are allocated by the database server when new data is stored. If you attempt to store an object and provide a ProductId value other than zero, then an exception will be thrown.

I reset the Category and Supplier properties to prevent Entity Framework Core from trying to deal with related data when storing an object. Entity Framework Core is capable of processing related data, but it can produce unexpected outcomes. (I show you how to create related data in the "Creating New Related Data Objects" section, later in this chapter.)

Notice I call the View method with arguments when validation fails, like this:

```
...
return View("ProductEditor",
    ViewModelFactory.Create(product, Categories, Suppliers));
...
```

I do this because the view model object expected by the view isn't the same data type that I have extracted from the request using model binding. Instead, I create a new view model object that incorporates the model bound data and passes this to the View method.

Restart ASP.NET Core, request http://localhost:5000/controllers, and click Create. Fill out the form and click the Create button to submit the data. The new object will be stored in the database and displayed when the browser is redirected to the Index action, as shown in Figure 31-3.

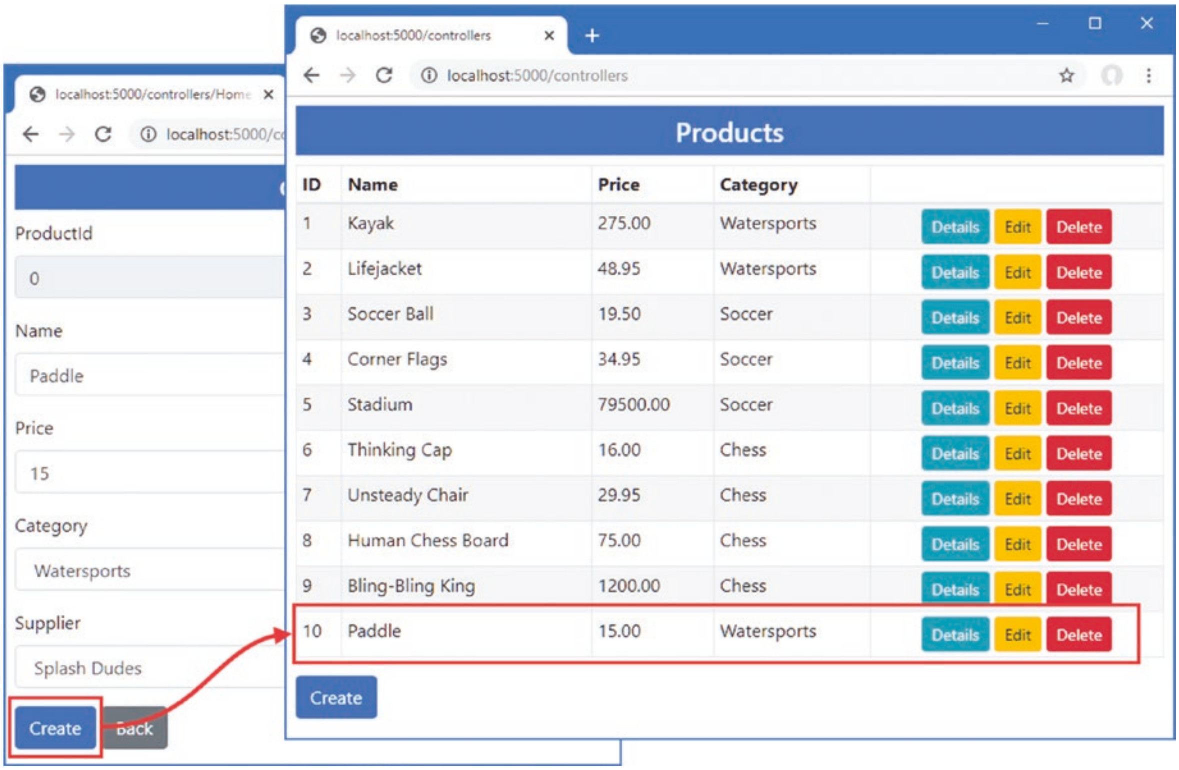

Figure 31-3. *Creating a new object*

Notice that `select` elements allow the user to select the values for the `CategoryId` and `SupplierId` properties, using the category and supplier names, like this:

```
...
<select asp-for="Product.SupplierId" class="form-control" disabled="@Model.ReadOnly"
        asp-items="@(new SelectList(Model.Suppliers, "SupplierId", "Name"))">
    <option value="" disabled selected>Choose a Supplier</option>
</select>
...
```

In Chapter 30, I used `input` elements to allow the value of these properties to be set directly, but that was because I wanted to demonstrate different types of validation. In real applications, it is a good idea to provide the user with restricted choices when the application already has the data it expects the user to choose from. Making the user enter a valid primary key, for example, makes no sense in a real project because the application can easily provide the user with a list of those keys to choose from, as shown in Figure 31-4.

■ **Tip** I show you different techniques for creating related data in the "Creating New Related Data Objects" section.

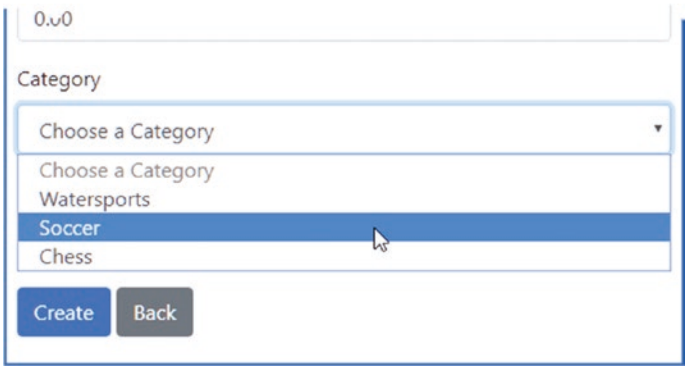

Figure 31-4. *Presenting the user with a choice*

Editing Data

The process for editing data is similar to creating data. The first step is to add a new method to the view model factory that will configure the way the data is presented to the user, as shown in Listing 31-13.

Listing 31-13. Adding a Method in the ViewModelFactory.cs File in the Models Folder

```
using System.Collections.Generic;
using System.Linq;

namespace WebApp.Models {

    public static class ViewModelFactory {

        public static ProductViewModel Details(Product p) {
            return new ProductViewModel {
                Product = p, Action = "Details",
                ReadOnly = true, Theme = "info", ShowAction = false,
                Categories = p == null ? Enumerable.Empty<Category>()
                    : new List<Category> { p.Category },
                Suppliers = p == null ? Enumerable.Empty<Supplier>()
                    : new List<Supplier> { p.Supplier},
            };
        }

        public static ProductViewModel Create(Product product,
                IEnumerable<Category> categories, IEnumerable<Supplier> suppliers) {
            return new ProductViewModel {
                Product = product, Categories = categories, Suppliers = suppliers
            };
        }

        public static ProductViewModel Edit(Product product,
                IEnumerable<Category> categories, IEnumerable<Supplier> suppliers) {
            return new ProductViewModel {
                Product = product, Categories = categories, Suppliers = suppliers,
                Theme = "warning", Action = "Edit"
            };
        }
    }
}
```

The next step is to add the action methods to the Home controller that will display the current properties of a Product object to the user and receive the changes the user makes, as shown in Listing 31-14.

Listing 31-14. Adding Action Methods in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Threading.Tasks;
using WebApp.Models;

namespace WebApp.Controllers {

    [AutoValidateAntiforgeryToken]
    public class HomeController : Controller {
        private DataContext context;

        private IEnumerable<Category> Categories => context.Categories;
        private IEnumerable<Supplier> Suppliers => context.Suppliers;

        public HomeController(DataContext data) {
            context = data;
        }

        // ...other action methods omitted for brevity...

        public async Task<IActionResult> Edit(long id) {
            Product p = await context.Products.FindAsync(id);
            ProductViewModel model = ViewModelFactory.Edit(p, Categories, Suppliers);
            return View("ProductEditor", model);
        }

        [HttpPost]
        public async Task<IActionResult> Edit([FromForm]Product product) {
            if (ModelState.IsValid) {
                product.Category = default;
                product.Supplier = default;
                context.Products.Update(product);
                await context.SaveChangesAsync();
                return RedirectToAction(nameof(Index));
            }
            return View("ProductEditor",
                ViewModelFactory.Edit(product, Categories, Suppliers));
        }
    }
}
```

To see the editing feature at work, restart ASP.NET Core, navigate to http://localhost:5000/controllers, and click one of the Edit buttons. Change one or more property values and submit the form. The changes will be stored in the database and reflected in the list displayed when the browser is redirected to the Index action, as shown in Figure 31-5.

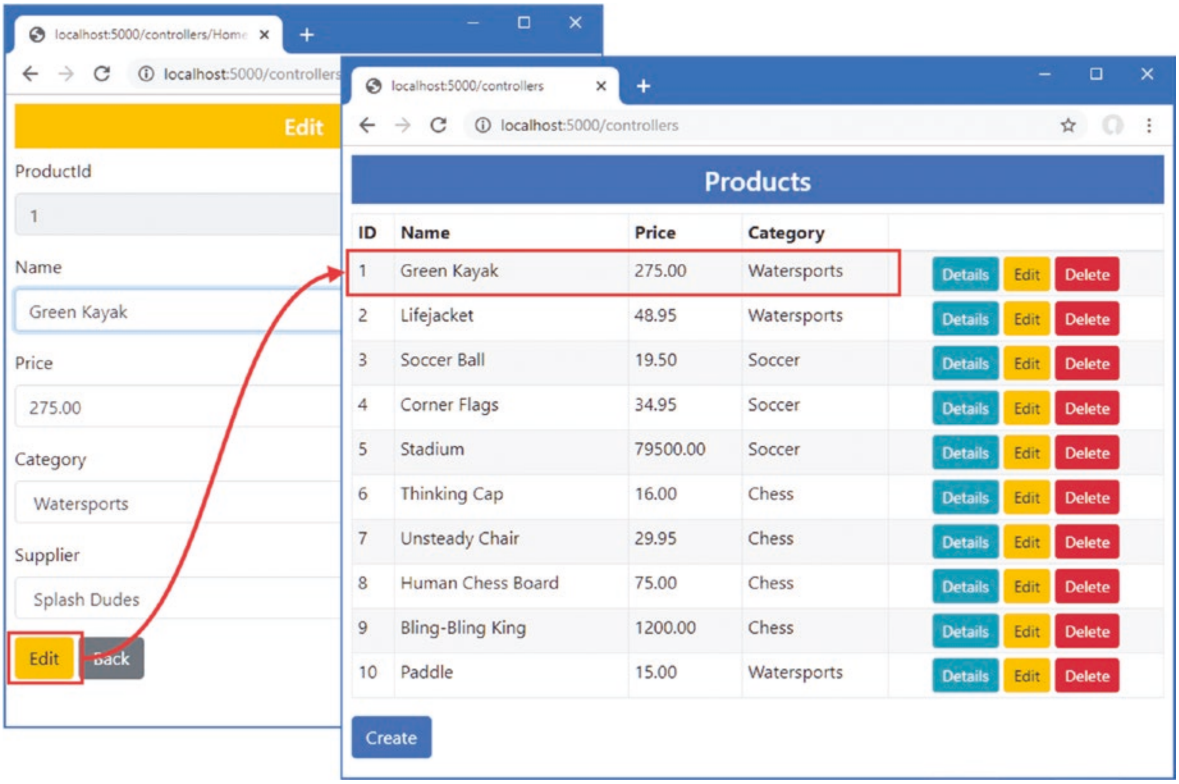

Figure 31-5. *Editing a product*

Notice that the `ProductId` property cannot be changed. Attempting to change the primary key of an object should be avoided because it interferes with the Entity Framework Core understanding of the identity of its objects. If you can't avoid changing the primary key, then the safest approach is to delete the existing object and store a new one.

Deleting Data

The final basic operation is removing objects from the database. By now the pattern will be clear, and the first step is to add a method to create a view model object to determine how the data is presented to the user, as shown in Listing 31-15.

Listing 31-15. Adding a Method in the ViewModelFactory.cs File in the Models Folder

```
using System.Collections.Generic;
using System.Linq;

namespace WebApp.Models {

    public static class ViewModelFactory {

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

        public static ProductViewModel Delete(Product p,
                IEnumerable<Category> categories, IEnumerable<Supplier> suppliers) {
            return new ProductViewModel {
                Product = p, Action = "Delete",
                ReadOnly = true, Theme = "danger",
```

```
                Categories = categories, Suppliers = suppliers
            };
        }
    }
}
```

Listing 31-16 adds the action methods to the Home controller that will respond to the GET request by displaying the selected object and the POST request to remove that object from the database.

Listing 31-16. Adding Action Methods in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Threading.Tasks;
using WebApp.Models;

namespace WebApp.Controllers {

    [AutoValidateAntiforgeryToken]
    public class HomeController : Controller {
        private DataContext context;

        private IEnumerable<Category> Categories => context.Categories;
        private IEnumerable<Supplier> Suppliers => context.Suppliers;

        public HomeController(DataContext data) {
            context = data;
        }

        // ...other action methods removed for brevity...

        public async Task<IActionResult> Delete(long id) {
            ProductViewModel model = ViewModelFactory.Delete(
                await context.Products.FindAsync(id), Categories, Suppliers);
            return View("ProductEditor", model);
        }

        [HttpPost]
        public async Task<IActionResult> Delete(Product product) {
            context.Products.Remove(product);
            await context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
    }
}
```

The model binding process creates a Product object from the form data, which is passed to Entity Framework Core to remove from the database. Once the data has been removed from the database, the browser is redirected to the Index action, as shown in Figure 31-6.

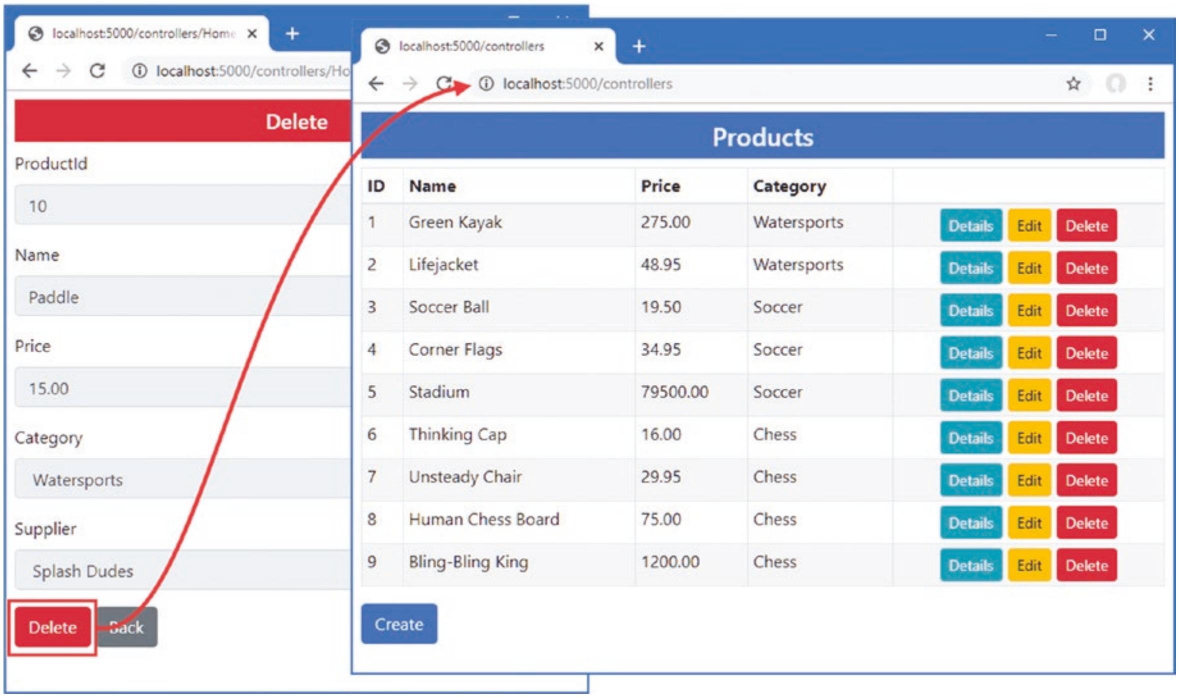

Figure 31-6. *Deleting data*

Creating a Razor Pages Forms Application

Working with Razor Forms relies on similar techniques as the controller examples, albeit broken up into smaller chunks of functionality. As you will see, the main difficulty is preserving the modular nature of Razor Pages without duplicating code and markup. The first step is to create the Razor Page that will display the list of `Product` objects and provide the links to the other operations. Add a Razor Page named `Index.cshtml` to the `Pages` folder with the content shown in Listing 31-17.

Listing 31-17. The Contents of the Index.cshtml File in the Pages Folder

```
@page "/pages/{id:long?}"
@model IndexModel
@using Microsoft.AspNetCore.Mvc.RazorPages
@using Microsoft.EntityFrameworkCore

<div class="m-2">
    <h4 class="bg-primary text-white text-center p-2">Products</h4>
    <table class="table table-sm table-bordered table-striped">
        <thead>
            <tr>
                <th>ID</th><th>Name</th><th>Price</th><th>Category</th><th></th>
            </tr>
        </thead>
        <tbody>
            @foreach (Product p in Model.Products) {
```

```
                <tr>
                    <td>@p.ProductId</td>
                    <td>@p.Name</td>
                    <td>@p.Price</td>
                    <td>@p.Category.Name</td>
                    <td class="text-center">
                        <a asp-page="Details" asp-route-id="@p.ProductId"
                           class="btn btn-sm btn-info">Details</a>
                        <a asp-page="Edit" asp-route-id="@p.ProductId"
                           class="btn btn-sm btn-warning">Edit</a>
                        <a asp-page="Delete" asp-route-id="@p.ProductId"
                           class="btn btn-sm btn-danger">Delete</a>
                    </td>
                </tr>
            }
        </tbody>
    </table>
    <a asp-page="Create" class="btn btn-primary">Create</a>
</div>

@functions {

    public class IndexModel: PageModel {
        private DataContext context;

        public IndexModel(DataContext dbContext) {
            context = dbContext;
        }

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

        public void OnGetAsync(long id = 1) {
            Products = context.Products
                .Include(p => p.Category).Include(p => p.Supplier);
        }
    }
}
```

This view part of the page displays a table populated with the details of the Product objects obtained from the database by the page model. Use a browser to request http://localhost:5000/pages, and you will see the response shown in Figure 31-7. Alongside the details of the Product objects, the page displays anchor elements that navigate to other Razor Pages, which I define in the sections that follow.

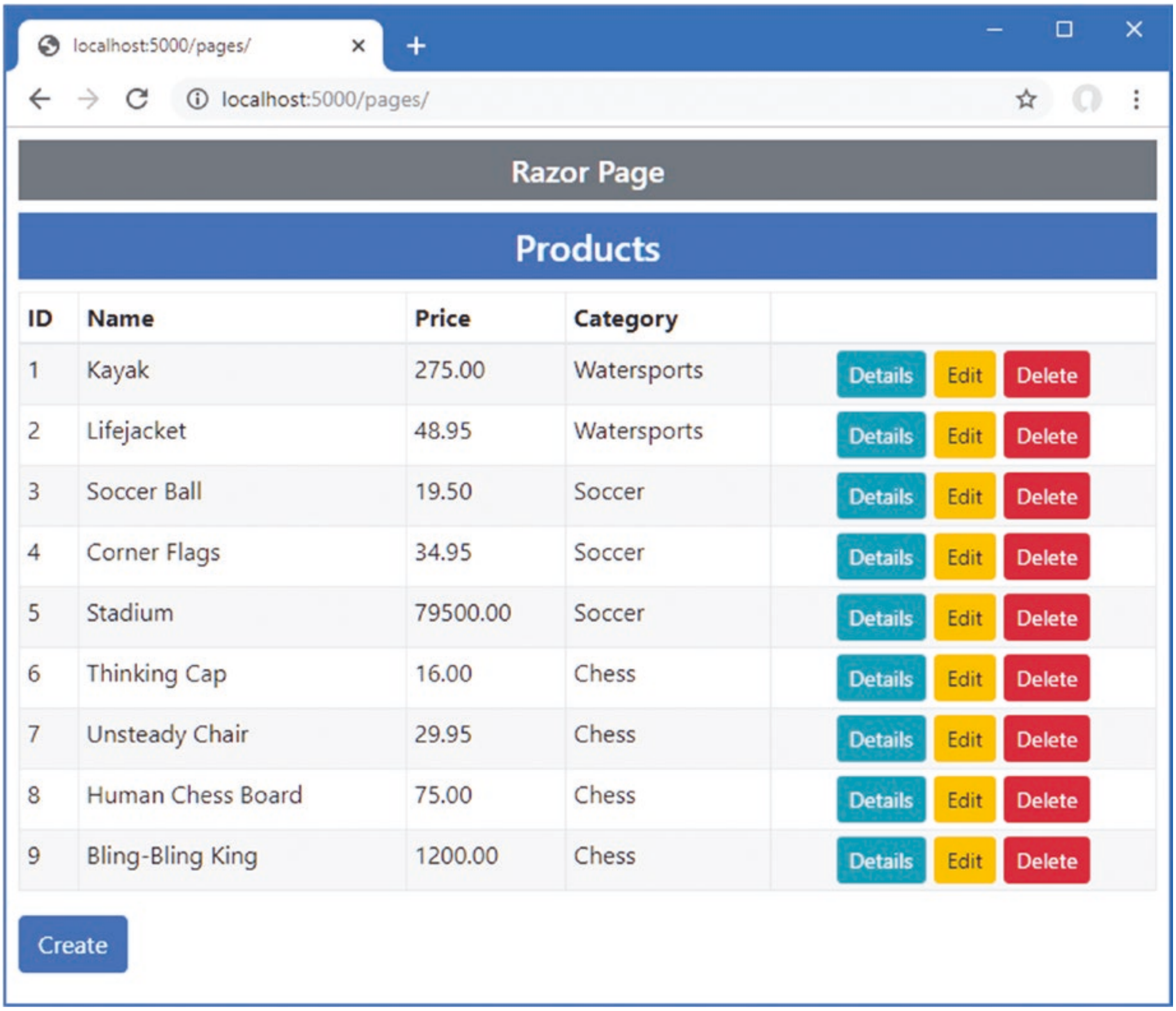

Figure 31-7. *Listing data using a Razor Page*

Creating Common Functionality

I don't want to duplicate the same HTML form and supporting code in each of the pages required by the example application. Instead, I am going to define a partial view that defines the HTML form and a base class that defines the common code required by the page model classes. For the partial view, a Razor View named `_ProductEditor.cshtml` to the `Pages` folder with the content shown in Listing 31-18.

USING MULTIPLE PAGE

The `asp-page-handler` attribute can be used to specify the name of a handler method, which allows a Razor Page to be used for more than one operation. I don't like this feature because the result is too close to a standard MVC controller and undermines the self-contained and modular aspects of Razor Page development that I like.

The approach I prefer is, of course, the one that I have taken in this chapter, which is to consolidate common content in partial views and a shared base class. Either approach works, and I recommend you try both to see which suits you and your project.

Listing 31-18. The Contents of the _ProductEditor.cshtml File in the Pages Folder

```
@model ProductViewModel

<partial name="_Validation" />

<h5 class="bg-@Model.Theme text-white text-center p-2">@Model.Action</h5>

<form asp-page="@Model.Action" method="post">
    <div class="form-group">
        <label asp-for="Product.ProductId"></label>
        <input class="form-control" asp-for="Product.ProductId" readonly />
    </div>
    <div class="form-group">
        <label asp-for="Product.Name"></label>
        <div>
            <span asp-validation-for="Product.Name" class="text-danger"></span>
        </div>
        <input class="form-control" asp-for="Product.Name"
               readonly="@Model.ReadOnly" />
    </div>
    <div class="form-group">
        <label asp-for="Product.Price"></label>
        <div>
            <span asp-validation-for="Product.Price" class="text-danger"></span>
        </div>
        <input class="form-control" asp-for="Product.Price"
               readonly="@Model.ReadOnly" />
    </div>
    <div class="form-group">
        <label asp-for="Product.CategoryId">Category</label>
        <div>
            <span asp-validation-for="Product.CategoryId" class="text-danger"></span>
        </div>
        <select asp-for="Product.CategoryId" class="form-control"
                disabled="@Model.ReadOnly"

                asp-items="@(new SelectList(Model.Categories,
                    "CategoryId", "Name"))">
            <option value="" disabled selected>Choose a Category</option>
        </select>
    </div>
    <div class="form-group">
        <label asp-for="Product.SupplierId">Supplier</label>
        <div>
            <span asp-validation-for="Product.SupplierId" class="text-danger"></span>
        </div>
        <select asp-for="Product.SupplierId" class="form-control"
                disabled="@Model.ReadOnly"
                asp-items="@(new SelectList(Model.Suppliers,
                    "SupplierId", "Name"))">
            <option value="" disabled selected>Choose a Supplier</option>
        </select>
    </div>
    @if (Model.ShowAction) {
        <button class="btn btn-@Model.Theme" type="submit">@Model.Action</button>
    }
    <a class="btn btn-secondary" asp-page="Index">Back</a>
</form>
```

The partial view uses the ProductViewModel class as its model type and relies on the built-in tag helpers to present input and select elements for the properties defined by the Product class. This is the same content used earlier in the chapter, except with the asp-action attribute replaced with asp-page to specify the target for the form and anchor elements.

To define the page model base class, add a class file named EditorPageModel.cs to the Pages folder and use it to define the class shown in Listing 31-19.

Listing 31-19. The Contents of the EditorPageModel.cs File in the Pages Folder

```
using Microsoft.AspNetCore.Mvc.RazorPages;
using System.Collections.Generic;
using WebApp.Models;

namespace WebApp.Pages {

    public class EditorPageModel : PageModel {

        public EditorPageModel(DataContext dbContext) {
            DataContext = dbContext;
        }

        public DataContext DataContext { get; set; }

        public IEnumerable<Category> Categories => DataContext.Categories;
        public IEnumerable<Supplier> Suppliers => DataContext.Suppliers;

        public ProductViewModel ViewModel { get; set; }
    }
}
```

The properties defined by this class are simple, but they will help simplify the page model classes of the Razor Pages that handle each operation.

All the Razor Pages required for this example depend on the same namespaces. Add the expressions shown in Listing 31-20 to the _ViewImports.cshtml file in the Pages folder to avoid duplicate expressions in the individual pages.

■ **Tip** Make sure you alter the _ViewImports.cshtml file in the Pages folder and not the file with the same name in the Views folder.

Listing 31-20. Adding Namespaces in the _ViewImports.cshtml File in the Pages Folder

```
@namespace WebApp.Pages
@using WebApp.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, WebApp
@using Microsoft.AspNetCore.Mvc.RazorPages
@using Microsoft.EntityFrameworkCore
@using WebApp.Pages
@using System.Text.Json
@using Microsoft.AspNetCore.Http
```

Defining Pages for the CRUD Operations

With the partial view and shared base class in place, the pages that handle individual operations are simple. Add a Razor Page named Details.cshtml to the Pages folder with the code and content shown in Listing 31-21.

Listing 31-21. The Contents of the Details.cshtml File in the Pages Folder

```
@page "/pages/details/{id}"
@model DetailsModel

<div class="m-2">
    <partial name="_ProductEditor" model="@Model.ViewModel" />
</div>

@functions {

    public class DetailsModel: EditorPageModel {

        public DetailsModel(DataContext dbContext): base(dbContext) {}

        public async Task OnGetAsync(long id) {
            Product p = await DataContext.Products.
                Include(p => p.Category).Include(p => p.Supplier)
                .FirstOrDefaultAsync(p => p.ProductId == id);
            ViewModel = ViewModelFactory.Details(p);
        }
    }
}
```

The constructor receives an Entity Framework Core context object, which it passes to the base class. The handler method responds to requests by querying the database and using the response to create a `ProductViewModel` object using the `ViewModelFactory` class.

Add a Razor Page named `Create.cshtml` to the `Pages` folder with the code and content shown in Listing 31-22.

■ **Tip** Using a partial view means that the `asp-for` attributes set element names without an additional prefix. This allows me to use the `FromForm` attribute for model binding without using the `Name` argument.

Listing 31-22. The Contents of the Create.cshtml File in the Pages Folder

```
@page "/pages/create"
@model CreateModel

<div class="m-2">
    <partial name="_ProductEditor" model="@Model.ViewModel" />
</div>

@functions {

    public class CreateModel: EditorPageModel {

        public CreateModel(DataContext dbContext): base(dbContext) {}

        public void OnGet() {
            ViewModel = ViewModelFactory.Create(new Product(),
                Categories, Suppliers);
        }
```

```
        public async Task<IActionResult> OnPostAsync([FromForm]Product product) {
            if (ModelState.IsValid) {
                product.ProductId = default;
                product.Category = default;
                product.Supplier = default;
                DataContext.Products.Add(product);
                await DataContext.SaveChangesAsync();
                return RedirectToPage(nameof(Index));
            }
            ViewModel = ViewModelFactory.Create(product, Categories, Suppliers);
            return Page();
        }
    }
}
```

Add a Razor Page named `Edit.cshtml` to the Pages folder with the code and content shown in Listing 31-23.

Listing 31-23. The Contents of the Edit.cshtml File in the Pages Folder

```
@page "/pages/edit/{id}"
@model EditModel

<div class="m-2">
    <partial name="_ProductEditor" model="@Model.ViewModel" />
</div>

@functions {

    public class EditModel: EditorPageModel {

        public EditModel(DataContext dbContext): base(dbContext) {}

        public async Task OnGetAsync(long id) {
            Product p = await this.DataContext.Products.FindAsync(id);
            ViewModel = ViewModelFactory.Edit(p, Categories, Suppliers);
        }

        public async Task<IActionResult> OnPostAsync([FromForm]Product product) {
            if (ModelState.IsValid) {
                product.Category = default;
                product.Supplier = default;
                DataContext.Products.Update(product);
                await DataContext.SaveChangesAsync();
                return RedirectToPage(nameof(Index));
            }
            ViewModel = ViewModelFactory.Edit(product, Categories, Suppliers);
            return Page();
        }
    }
}
```

Add a Razor Page named `Delete.cshtml` to the Pages folder with the code and content shown in Listing 31-24.

Listing 31-24. The Contents of the Delete.cshtml File in the Pages Folder

```
@page "/pages/delete/{id}"
@model DeleteModel

<div class="m-2">
    <partial name="_ProductEditor" model="@Model.ViewModel" />
</div>

@functions {

    public class DeleteModel: EditorPageModel {

        public DeleteModel(DataContext dbContext): base(dbContext) {}

        public async Task OnGetAsync(long id) {
            ViewModel = ViewModelFactory.Delete(
                await DataContext.Products.FindAsync(id), Categories, Suppliers);
        }

        public async Task<IActionResult> OnPostAsync([FromForm]Product product) {
            DataContext.Products.Remove(product);
            await DataContext.SaveChangesAsync();
            return RedirectToPage(nameof(Index));
        }
    }
}
```

Restart ASP.NET Core and navigate to `http://localhost:5000/pages`, and you will be able to click the links to view, create, edit, and remove data, as shown in Figure 31-8.

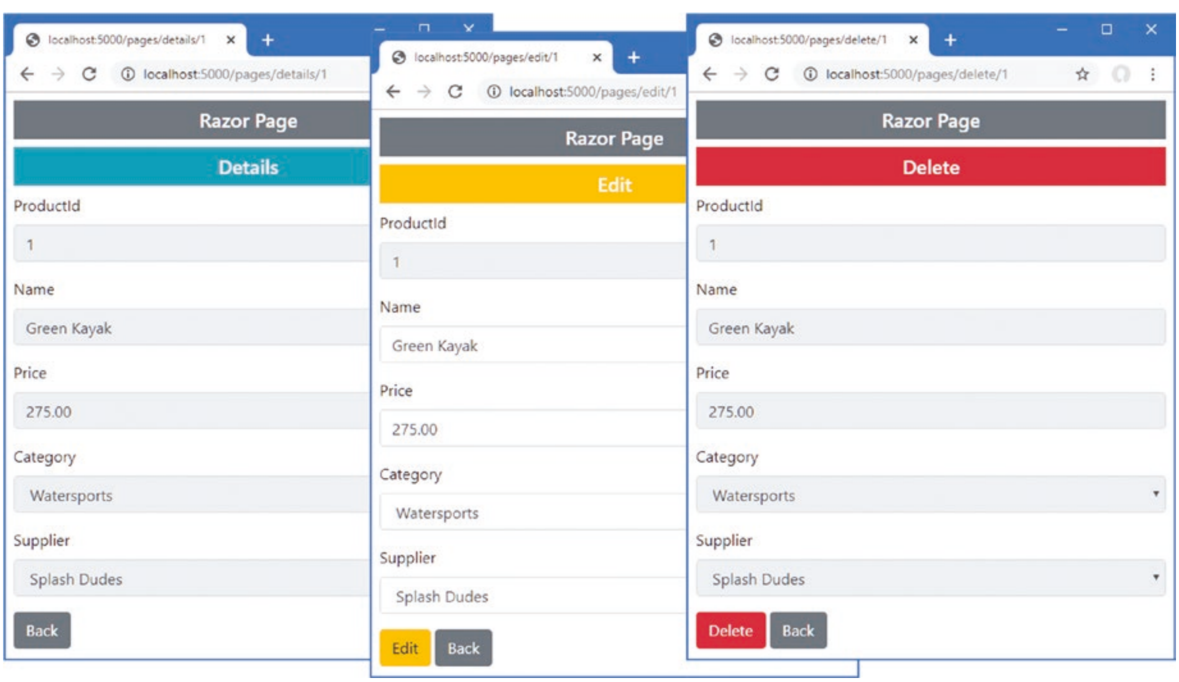

Figure 31-8. *Using Razor Pages*

Creating New Related Data Objects

Some applications will need to allow the user to create new related data so that, for example, a new Category can be created along with a Product in that Category. There are two ways to approach this problem, as described in the sections that follow.

Providing the Related Data in the Same Request

The first approach is to ask the user to provide the data required to create the related data in the same form. For the example application, this means collecting details for a Category object in the same form that the user enters the values for the Product object.

This can be a useful approach for simple data types, where only a small amount of data is required to create the related object but is not well suited for types with many properties.

I prefer to define the HTML elements for the related data type in their own partial view. Add a Razor View named _CategoryEditor.cshtml to the Pages folder with the content shown in Listing 31-25.

Listing 31-25. The Contents of the _CategoryEditor.cshtml File in the Pages Folder

```
@model Product
<script type="text/javascript">
    $(document).ready(() => {
        const catGroup = $("#categoryGroup").hide();
        $("select[name='Product.CategoryId']").on("change", (event) =>
            event.target.value === "-1" ? catGroup.show() : catGroup.hide());
    });
</script>

<div class="form-group bg-info p-1" id="categoryGroup">
    <label class="text-white" asp-for="Category.Name">
        New Category Name
    </label>
    <input class="form-control" asp-for="Category.Name" value="" />
</div>
```

The Category type requires only one property, which the user will provide using a standard input element. The script element in the partial view contains jQuery code that hides the new elements until the user selects an option element that sets a value of -1 for the Product.CategoryId property. (Using JavaScript is entirely optional, but it helps to emphasize the purpose of the new elements.)

Listing 31-26 adds the partial view to the editor, along with the option element that will display the elements for creating a new Category object.

Listing 31-26. Adding Elements in the _ProductEditor.cshtml File in the Pages Folder

```
...
<div class="form-group">
    <label asp-for="Product.CategoryId">Category</label>
    <div>
        <span asp-validation-for="Product.CategoryId" class="text-danger"></span>
    </div>
    <select asp-for="Product.CategoryId" class="form-control"
            disabled="@Model.ReadOnly" asp-items="@(new SelectList(Model.Categories,
                "CategoryId", "Name"))">
        <option value="-1">Create New Category...</option>
        <option value="" disabled selected>Choose a Category</option>
    </select>
</div>
```

```
<partial name="_CategoryEditor" for="Product" />
<div class="form-group">
    <label asp-for="Product.SupplierId">Supplier</label>
    <div><span asp-validation-for="Product.SupplierId" class="text-danger"></span></div>
    <select asp-for="Product.SupplierId" class="form-control" disabled="@Model.ReadOnly"
            asp-items="@(new SelectList(Model.Suppliers,
                "SupplierId", "Name"))">
        <option value="" disabled selected>Choose a Supplier</option>
    </select>
</div>
...
```

I need the new functionality in multiple pages, so to avoid code duplication, I have added a method that handles the related data to the page model base class, as shown in Listing 31-27.

Listing 31-27. Adding a Method in the EditorPageModel.cs File in the Pages Folder

```
using Microsoft.AspNetCore.Mvc.RazorPages;
using System.Collections.Generic;
using WebApp.Models;
using System.Threading.Tasks;

namespace WebApp.Pages {

    public class EditorPageModel : PageModel {

        public EditorPageModel(DataContext dbContext) {
            DataContext = dbContext;
        }

        public DataContext DataContext { get; set; }

        public IEnumerable<Category> Categories => DataContext.Categories;
        public IEnumerable<Supplier> Suppliers => DataContext.Suppliers;

        public ProductViewModel ViewModel { get; set; }

        protected async Task CheckNewCategory(Product product) {
            if (product.CategoryId == -1
                    && !string.IsNullOrEmpty(product.Category?.Name)) {
                DataContext.Categories.Add(product.Category);
                await DataContext.SaveChangesAsync();
                product.CategoryId = product.Category.CategoryId;
                ModelState.Clear();
                TryValidateModel(product);
            }
        }
    }
}
```

The new code creates a Category object using the data received from the user and stores it in the database. The database server assigns a primary key to the new object, which Entity Framework Core uses to update the Category object. This allows me to update the CategoryId property of the Product object and then re-validate the model data, knowing that the value assigned to the CategoryId property will pass validation because it corresponds to the newly allocated key. To integrate the new functionality into the Create page, add the statement shown in Listing 31-28.

Listing 31-28. Adding a Statement in the Create.cshtml File in the Pages Folder

```
...
public async Task<IActionResult> OnPostAsync([FromForm]Product product) {
    await CheckNewCategory(product);
    if (ModelState.IsValid) {
        product.ProductId = default;
        product.Category = default;
        product.Supplier = default;
        DataContext.Products.Add(product);
        await DataContext.SaveChangesAsync();
        return RedirectToPage(nameof(Index));
    }
    ViewModel = ViewModelFactory.Create(product, Categories, Suppliers);
    return Page();
}
...
```

Add the same statement to the handler method in the `Edit` page, as shown in Listing 31-29.

Listing 31-29. Adding a Statement in the Edit.cshtml File in the Pages Folder

```
...
public async Task<IActionResult> OnPostAsync([FromForm]Product product) {
    await CheckNewCategory(product);
    if (ModelState.IsValid) {
        product.Category = default;
        product.Supplier = default;
        DataContext.Products.Update(product);
        await DataContext.SaveChangesAsync();
        return RedirectToPage(nameof(Index));
    }
    ViewModel = ViewModelFactory.Edit(product, Categories, Suppliers);
    return Page();
}
...
```

Restart ASP.NET Core so the page model base class is recompiled and use a browser to request `http://localhost:5000/pages/edit/1`. Click the Category `select` element and choose Create New Category from the list of options. Enter a new category name into the input element and click the Edit button. When the request is processed, a new `Category` object will be stored in the database and associated with the `Product` object, as shown in Figure 31-9.

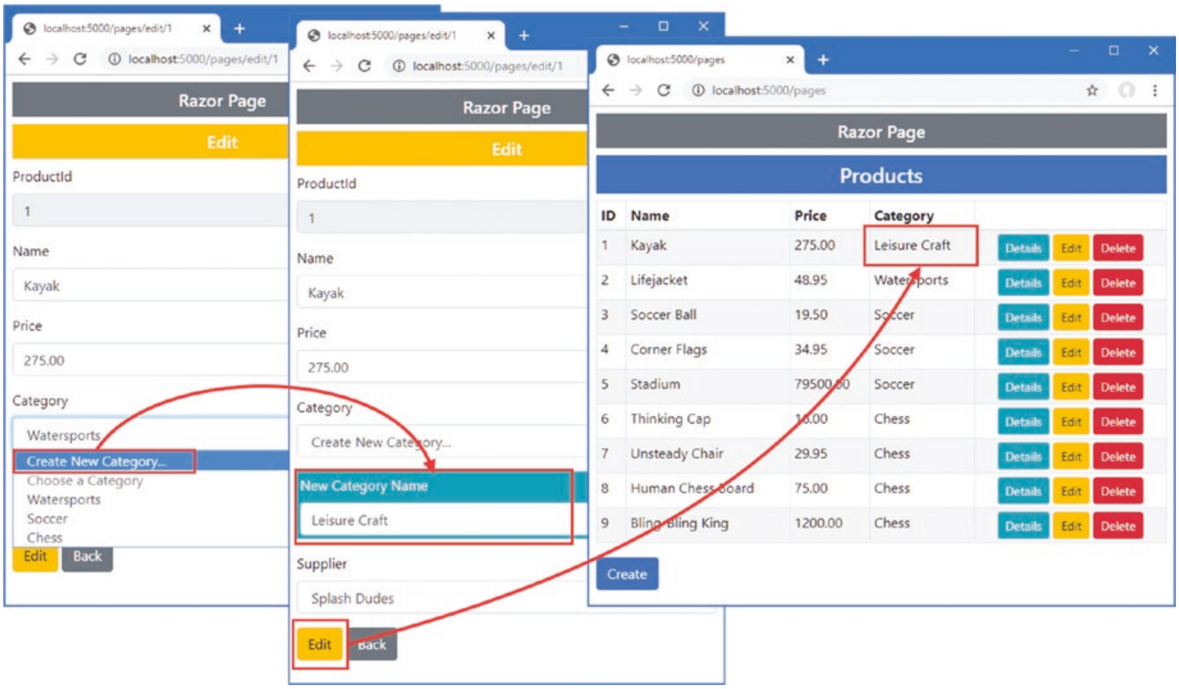

Figure 31-9. *Creating related data*

Breaking Out to Create New Data

For related data types that have their own complex creation process, adding elements to the main form can be overwhelming to the user; a better approach is to navigate away from the main form to another controller or page, let the user create the new object, and then return to complete the original task. I will demonstrate this technique for the creation of `Supplier` objects, even though the `Supplier` type is simple and requires only two values from the user.

To create a form that will let the user create `Supplier` objects, add a Razor Page named `SupplierBreakOut.cshtml` to the `Pages` folder with the content shown in Listing 31-30.

Listing 31-30. The Contents of the SupplierBreakOut.cshtml File in the Pages Folder

```
@page "/pages/supplier"
@model SupplierPageModel

<div class="m-2">
    <h5 class="bg-secondary text-white text-center p-2">New Supplier</h5>
    <form asp-page="SupplierBreakOut" method="post">
        <div class="form-group">
            <label asp-for="Supplier.Name"></label>
            <input class="form-control" asp-for="Supplier.Name"  />
        </div>
        <div class="form-group">
            <label asp-for="Supplier.City"></label>
            <input class="form-control" asp-for="Supplier.City"  />
        </div>
        <button class="btn btn-secondary" type="submit">Create</button>
        <a class="btn btn-outline-secondary"
                asp-page="@Model.ReturnPage" asp-route-id="@Model.ProductId">
            Cancel
        </a>
    </form>
</div>
```

```
@functions {

    public class SupplierPageModel: PageModel {
        private DataContext context;

        public SupplierPageModel(DataContext dbContext) {
            context = dbContext;
        }

        [BindProperty]
        public Supplier Supplier { get; set; }

        public string ReturnPage { get; set; }
        public string ProductId { get; set; }

        public void OnGet([FromQuery(Name="Product")] Product product,
                string returnPage) {
            TempData["product"] = Serialize(product);
            TempData["returnAction"] =  ReturnPage = returnPage;
            TempData["productId"] = ProductId = product.ProductId.ToString();
        }

        public async Task<IActionResult> OnPostAsync() {
            context.Suppliers.Add(Supplier);
            await context.SaveChangesAsync();
            Product product = Deserialize(TempData["product"] as string);
            product.SupplierId = Supplier.SupplierId;
            TempData["product"] = Serialize(product);
            string id = TempData["productId"] as string;
            return RedirectToPage(TempData["returnAction"] as string,
                new { id = id });
        }

        private string Serialize(Product p) => JsonSerializer.Serialize(p);
        private Product Deserialize(string json) =>
            JsonSerializer.Deserialize<Product>(json);
    }
}
```

The user will navigate to this page using a GET request that will contain the details of the Product the user has provided and the name of the page that the user should be returned to. This data is stored using the temp data feature.

This page presents the user with a form containing fields for the Name and City properties required to create a new Supplier object. When the form is submitted, the POST handler method stores a new Supplier object and uses the key assigned by the database server to update the Product object, which is then stored as temp data again. The user is redirected back to the page from which they arrived.

Listing 31-31 adds elements to the _ProductEditor partial view that will allow the user to navigate to the new page.

Listing 31-31. Adding Elements in the _ProductEditor.cshtml File in the Pages Folder

```
...
<partial name="_CategoryEditor" for="Product" />

<div class="form-group">
    <label asp-for="Product.SupplierId">
        Supplier
        @if (!Model.ReadOnly) {
            <input type="hidden" name="returnPage" value="@Model.Action" />
            <button class="btn btn-sm btn-outline-primary ml-3"
```

```
                asp-page="SupplierBreakOut" formmethod="get" formnovalidate>
            Create New Supplier
        </button>
    }
    </label>
    <div>
        <span asp-validation-for="Product.SupplierId" class="text-danger"></span>
    </div>
    <select asp-for="Product.SupplierId" class="form-control"
        disabled="@Model.ReadOnly" asp-items="@(new SelectList(Model.Suppliers,
                "SupplierId", "Name"))">
        <option value="" disabled selected>Choose a Supplier</option>
    </select>
</div>
...
```

The new elements add a hidden `input` element that captures the page to return to and a `button` element that submits the form data to the `SupplierBreakOut` page using a GET request, which means the form values will be encoded in the query string (and is the reason I used the `FromQuery` attribute in Listing 31-30). Listing 31-32 shows the change required to the `Create` page to add support for retrieving the temp data and using it to populate the `Product` form.

Listing 31-32. Retrieving Data in the Create.cshtml File in the Pages Folder

```
@page "/pages/create"
@model CreateModel

<div class="m-2">
    <partial name="_ProductEditor" model="@Model.ViewModel" />
</div>

@functions {

    public class CreateModel: EditorPageModel {

        public CreateModel(DataContext dbContext): base(dbContext) {}

        public void OnGet() {
            Product p = TempData.ContainsKey("product")
                ? JsonSerializer.Deserialize<Product>(TempData["product"] as string)
                : new Product();
            ViewModel = ViewModelFactory.Create(p, Categories, Suppliers);
        }

        public async Task<IActionResult> OnPostAsync([FromForm]Product product) {
            await CheckNewCategory(product);
            if (ModelState.IsValid) {
                product.ProductId = default;
                product.Category = default;
                product.Supplier = default;
                DataContext.Products.Add(product);
                await DataContext.SaveChangesAsync();
                return RedirectToPage(nameof(Index));
            }
            ViewModel = ViewModelFactory.Create(product, Categories, Suppliers);
            return Page();
        }
    }
}
```

A similar change is required in the Edit page, as shown in Listing 31-33. (The other pages do not require a change since the breakout is required only when the user is able to create or edit Product data.)

Listing 31-33. Retrieving Data in the Edit.cshtml File in the Pages Folder

```
@page "/pages/edit/{id}"
@model EditModel

<div class="m-2">
    <partial name="_ProductEditor" model="@Model.ViewModel" />
</div>

@functions {

    public class EditModel: EditorPageModel {

        public EditModel(DataContext dbContext): base(dbContext) {}

        public async Task OnGetAsync(long id) {
            Product p = TempData.ContainsKey("product")
                ? JsonSerializer.Deserialize<Product>(TempData["product"] as string)
                : await this.DataContext.Products.FindAsync(id);
            ViewModel = ViewModelFactory.Edit(p, Categories, Suppliers);
        }

        public async Task<IActionResult> OnPostAsync([FromForm]Product product) {
            await CheckNewCategory(product);
            if (ModelState.IsValid) {
                product.Category = default;
                product.Supplier = default;
                DataContext.Products.Update(product);
                await DataContext.SaveChangesAsync();
                return RedirectToPage(nameof(Index));
            }
            ViewModel = ViewModelFactory.Edit(product, Categories, Suppliers);
            return Page();
        }
    }
}
```

The effect is that the user is presented with a Create New Supplier button, which sends the browser to a form that can be used to create a Supplier object. Once the Supplier has been stored in the database, the browser is sent back to the originating page, and the form is populated with the data the user had entered, and the Supplier select element is set to the newly created object, as shown in Figure 31-10.

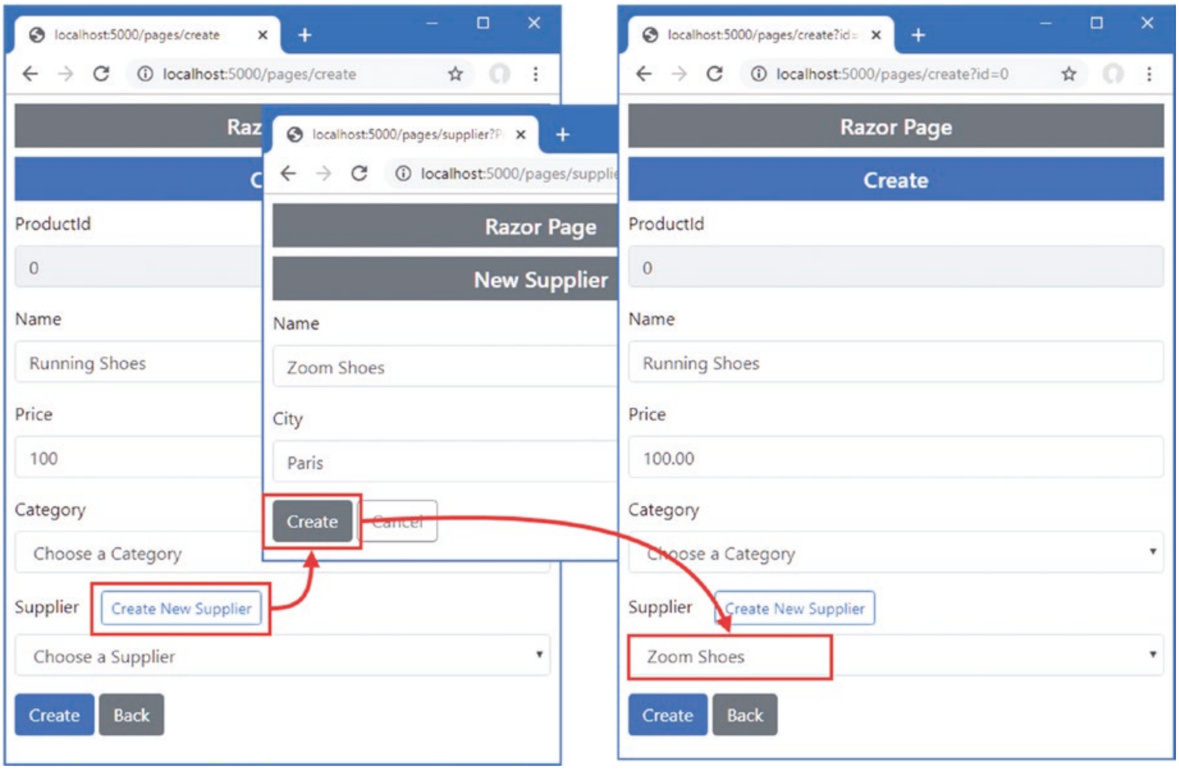

Figure 31-10. Breaking out to create related data

Summary

In this chapter, I demonstrated how the features described in earlier chapters can be combined with Entity Framework Core to create, read, update, and delete data. In Part 4, I describe some of the advanced features that ASP.NET Core provides.

PART IV

Advanced ASP.NET Core Features

CHAPTER 32

Creating the Example Project

In this chapter, you will create the example project used throughout this part of the book. The project contains a data model that is displayed using simple controllers and Razor Pages.

Creating the Project

Open a new PowerShell command prompt from the Windows Start menu and run the commands shown in Listing 32-1.

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-3`. See Chapter 1 for how to get help if you have problems running the examples.

Listing 32-1. Creating the Project

```
dotnet new globaljson --sdk-version 3.1.101 --output Advanced
dotnet new web --no-https --output Advanced --framework netcoreapp3.1
dotnet new sln -o Advanced

dotnet sln Advanced add Advanced
```

If you are using Visual Studio, open the `Advanced.sln` file in the `Advanced` folder. Select Project ➤ Platform Properties, navigate to the Debug page, and change the App URL field to **http://localhost:5000**, as shown in Figure 32-1. This changes the port that will be used to receive HTTP requests. Select File ➤ Save All to save the configuration changes.

A. Freeman, *Pro ASP.NET Core 3*, https://doi.org/10.1007/978-1-4842-5440-0_32

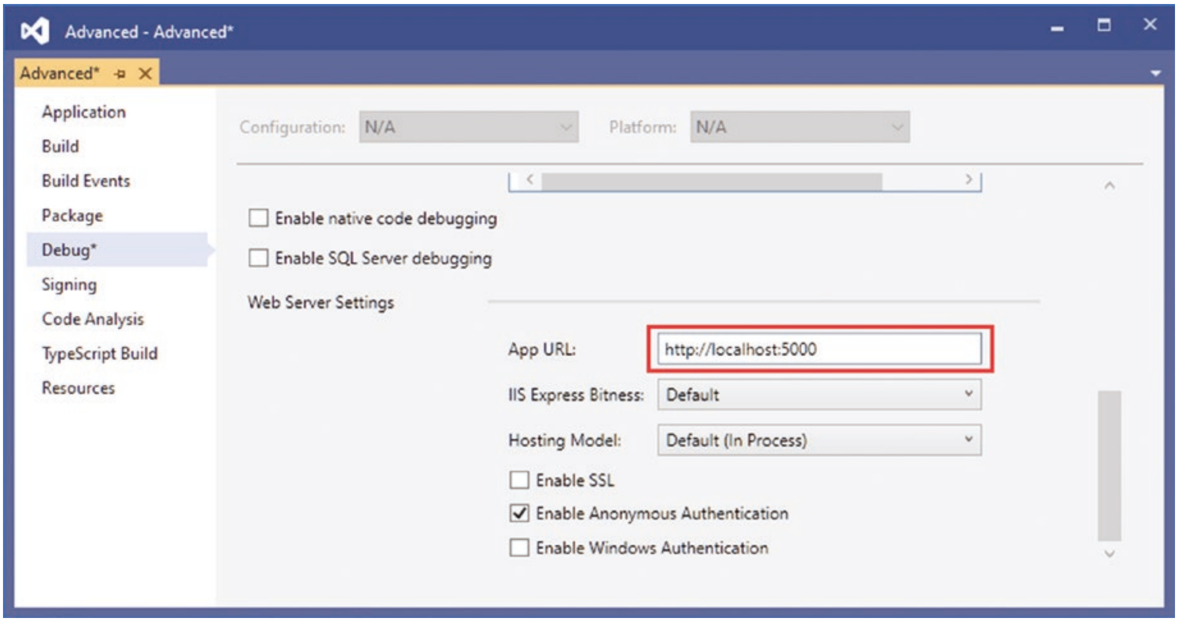

Figure 32-1. *Changing the HTTP port*

If you are using Visual Studio Code, open the `Advanced` folder. Click the Yes button when prompted to add the assets required for building and debugging the project, as shown in Figure 32-2.

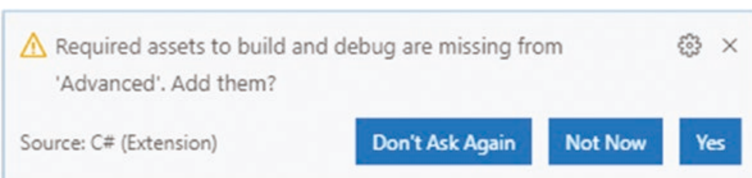

Figure 32-2. *Adding project assets*

Adding NuGet Packages to the Project

The data model will use Entity Framework Core to store and query data in a SQL Server LocalDB database. To add the NuGet packages for Entity Framework Core, use a PowerShell command prompt to run the commands shown in Listing 32-2 in the `Advanced` project folder.

Listing 32-2. Adding Packages to the Project

```
dotnet add package Microsoft.EntityFrameworkCore.Design --version 3.1.1
dotnet add package Microsoft.EntityFrameworkCore.SqlServer --version 3.1.1
```

If you are using Visual Studio, you can add the packages by selecting Project ➤ Manage NuGet Packages. Take care to choose the correct version of the packages to add to the project.

If you have not followed the examples in earlier chapters, you will need to install the global tool package that is used to create and manage Entity Framework Core migrations. Run the commands shown in Listing 32-3 to remove any existing version of the package and install the version required for this book.

Listing 32-3. Installing a Global Tool Package

```
dotne.t tool uninstall --global dotnet-ef
dotnet tool install --global dotnet-ef --version 3.1.1
```

Adding a Data Model

The data model for this application will consist of three classes, representing people, the department in which they work, and their location. Create a `Models` folder and add to it a class file named `Person.cs` with the code in Listing 32-4.

Listing 32-4. The Contents of the Person.cs File in the Models Folder

```
using System.Collections.Generic;

namespace Advanced.Models {

    public class Person {

        public long PersonId { get; set; }
        public string Firstname { get; set; }
        public string Surname { get; set; }
        public long DepartmentId { get; set; }
        public long LocationId { get; set; }

        public Department Department {get; set; }
        public Location Location { get; set; }
    }
}
```

Add a class file named `Department.cs` to the `Models` folder and use it to define the class shown in Listing 32-5.

Listing 32-5. The Contents of the Department.cs File in the Models Folder

```
using System.Collections.Generic;

namespace Advanced.Models {
    public class Department {

        public long Departmentid { get; set; }
        public string Name { get; set; }

        public IEnumerable<Person> People { get; set; }
    }
}
```

Add a class file named `Location.cs` to the `Models` folder and use it to define the class shown in Listing 32-6.

Listing 32-6. The Contents of the Location.cs File in the Models Folder

```
using System.Collections.Generic;

namespace Advanced.Models {
    public class Location {

        public long LocationId { get; set; }
        public string City { get; set; }
        public string State { get; set; }
```

```
        public IEnumerable<Person> People { get; set; }
    }
}
```

Each of the three data model classes defines a key property whose value will be allocated by the database when new objects are stored and defines foreign key properties that define the relationships between the classes. These are supplemented by navigation properties that will be used with the Entity Framework Core `Include` method to incorporate related data into queries.

To create the Entity Framework Core context class that will provide access to the database, add a file called `DataContext.cs` to the `Models` folder and add the code shown in Listing 32-7.

Listing 32-7. The Contents of the DataContext.cs File in the Models Folder

```
using Microsoft.EntityFrameworkCore;

namespace Advanced.Models {
    public class DataContext: DbContext {

        public DataContext(DbContextOptions<DataContext> opts)
            : base(opts) { }

        public DbSet<Person> People { get; set; }
        public DbSet<Department> Departments { get; set; }
        public DbSet<Location> Locations { get; set; }
    }
}
```

The context class defines properties that will be used to query the database for `Person`, `Department`, and `Location` data.

Preparing the Seed Data

Add a class called `SeedData.cs` to the `Models` folder and add the code shown in Listing 32-8 to define the seed data that will be used to populate the database.

Listing 32-8. The Contents of the SeedData.cs File in the Models Folder

```
using Microsoft.EntityFrameworkCore;
using System.Linq;

namespace Advanced.Models {
    public static class SeedData {

        public static void SeedDatabase(DataContext context) {
            context.Database.Migrate();
            if (context.People.Count() == 0 && context.Departments.Count() == 0 &&
                context.Locations.Count() == 0) {

                Department d1 = new Department { Name = "Sales" };
                Department d2 = new Department { Name = "Development" };
                Department d3 = new Department { Name = "Support" };
                Department d4 = new Department { Name = "Facilities" };

                context.Departments.AddRange(d1, d2, d3, d4);
                context.SaveChanges();

                Location l1 = new Location { City = "Oakland", State = "CA" };
                Location l2 = new Location { City = "San Jose", State = "CA" };
```

```
                Location l3 = new Location { City = "New York", State = "NY" };
                context.Locations.AddRange(l1, l2, l3);

                context.People.AddRange(
                    new Person {
                        Firstname = "Francesca", Surname = "Jacobs",
                        Department = d2, Location = l1
                    },
                    new Person {
                        Firstname = "Charles", Surname = "Fuentes",
                        Department = d2, Location = l3
                    },
                    new Person {
                        Firstname = "Bright", Surname = "Becker",
                        Department = d4, Location = l1
                    },
                    new Person {
                        Firstname = "Murphy", Surname = "Lara",
                        Department = d1, Location = l3
                    },
                    new Person {
                        Firstname = "Beasley", Surname = "Hoffman",
                        Department = d4, Location = l3
                    },
                    new Person {
                        Firstname = "Marks", Surname = "Hays",
                        Department = d4, Location = l1
                    },
                    new Person {
                        Firstname = "Underwood", Surname = "Trujillo",
                        Department = d2, Location = l1
                    },
                    new Person {
                        Firstname = "Randall", Surname = "Lloyd",
                        Department = d3, Location = l2
                    },
                    new Person {
                        Firstname = "Guzman", Surname = "Case",
                        Department = d2, Location = l2
                    });
                context.SaveChanges();
            }
        }
    }
}
```

The static SeedDatabase method ensures that all pending migrations have been applied to the database. If the database is empty, it is seeded with data. Entity Framework Core will take care of mapping the objects into the tables in the database, and the key properties will be assigned automatically when the data is stored.

Configuring Entity Framework Core Services and Middleware

Make the changes to the Startup class shown in Listing 32-9, which configure Entity Framework Core and set up the DataContext services that will be used throughout this part of the book to access the database.

Listing 32-9. Preparing Services and Middleware in the Startup.cs File in the Advanced Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
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 Advanced.Models;

namespace Advanced {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        public IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<DataContext>(opts => {
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:PeopleConnection"]);
                opts.EnableSensitiveDataLogging(true);
            });
        }

        public void Configure(IApplicationBuilder app, DataContext context) {

            app.UseDeveloperExceptionPage();
            app.UseRouting();

            app.UseEndpoints(endpoints => {
                endpoints.MapGet("/", async context => {
                    await context.Response.WriteAsync("Hello World!");
                });
            });

            SeedData.SeedDatabase(context);
        }
    }
}
```

To define the connection string that will be used for the application's data, add the configuration settings shown in Listing 32-10 in the appsettings.json file. The connection string should be entered on a single line.

Listing 32-10. Defining a Connection String in the appsettings.json File in the Advanced Folder

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

In addition to the connection string, Listing 32-10 increases the logging detail for Entity Framework Core so that the SQL queries sent to the database are logged.

Creating and Applying the Migration

To create the migration that will set up the database schema, use a PowerShell command prompt to run the command shown in Listing 32-11 in the Advanced project folder.

Listing 32-11. Creating an Entity Framework Core Migration

```
dotnet ef migrations add Initial
```

Once the migration has been created, apply it to the database using the command shown in Listing 32-12.

Listing 32-12. Applying the Migration to the Database

```
dotnet ef database update
```

The logging messages displayed by the application will show the SQL commands that are sent to the database.

■ **Note** If you need to reset the database, then run the `dotnet ef database drop --force` command and then the command in Listing 32-12.

Adding the Bootstrap CSS Framework

Following the pattern established in earlier chapters, I will use the Bootstrap CSS framework to style the HTML elements produced by the example application. To install the Bootstrap package, run the commands shown in Listing 32-13 in the Advanced project folder. These commands rely on the Library Manager package.

Listing 32-13. Installing the Bootstrap CSS Framework

```
libman init -p cdnjs
libman install twitter-bootstrap@4.3.1 -d wwwroot/lib/twitter-bootstrap
```

If you are using Visual Studio, you can install client-side packages by right-clicking the Advanced project item in the Solution Explorer and selecting Add ➤ Client-Side Library from the popup menu.

Configuring the Services and Middleware

I am going to enable runtime Razor view compilation in this project. Run the command shown in Listing 32-14 in the Advanced project folder to install the package that will provide the runtime compilation service.

Listing 32-14. Adding a Package to the Example Project

```
dotnet add package Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation --version 3.1.1
```

The example application in this part of the book will respond to requests using both MVC controllers and Razor Pages. Add the statements shown in Listing 32-15 to the `Startup` class to configure the services and middleware the application will use.

Listing 32-15. Adding Services and Middleware in the Startup.cs File in the Advanced Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
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 Advanced.Models;

namespace Advanced {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        public IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<DataContext>(opts => {
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:PeopleConnection"]);
                opts.EnableSensitiveDataLogging(true);
            });
            services.AddControllersWithViews().AddRazorRuntimeCompilation();
            services.AddRazorPages().AddRazorRuntimeCompilation();
        }

        public void Configure(IApplicationBuilder app, DataContext context) {

            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseRouting();
```

```
            app.UseEndpoints(endpoints => {
                endpoints.MapControllerRoute("controllers",
                    "controllers/{controller=Home}/{action=Index}/{id?}");
                endpoints.MapDefaultControllerRoute();
                endpoints.MapRazorPages();
            });

            SeedData.SeedDatabase(context);
        }
    }
}
```

In addition to the default controller route, I have added a route that matches URL paths that begin with `controllers`, which will make it easier to follow the examples in later chapters as they switch between controllers and Razor Pages. This is the same convention I adopted in earlier chapters, and I will route URL paths beginning with `/pages` to Razor Pages.

Creating a Controller and View

To display the application's data using a controller, create a folder named `Controllers` in the `Advanced` project folder and add to it a class file named `HomeController.cs`, with the content shown in Listing 32-16.

Listing 32-16. The Contents of the HomeController.cs File in the Controllers Folder

```
using Advanced.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;

namespace Advanced.Controllers {
    public class HomeController : Controller {
        private DataContext context;

        public HomeController(DataContext dbContext) {
            context = dbContext;
        }

        public IActionResult Index([FromQuery] string selectedCity) {
            return View(new PeopleListViewModel {
                People = context.People
                    .Include(p => p.Department).Include(p => p.Location),
                Cities = context.Locations.Select(l => l.City).Distinct(),
                SelectedCity = selectedCity
            });
        }
    }

    public class PeopleListViewModel {
        public IEnumerable<Person> People { get; set; }
        public IEnumerable<string> Cities { get; set; }
        public string SelectedCity { get; set; }

        public string GetClass(string city) =>
            SelectedCity == city ? "bg-info text-white" : "";
    }
}
```

To provide the controller with a view, create the Views/Home folder and add to it a Razor View named Index.cshtml with the content shown in Listing 32-17.

Listing 32-17. The Contents of the Index.cshtml File in the Views/Home Folder

```
@model PeopleListViewModel

<h4 class="bg-primary text-white text-center p-2">People</h4>

<table class="table table-sm table-bordered table-striped">
    <thead>
        <tr>
            <th>ID</th><th>Name</th><th>Dept</th><th>Location</th>
        </tr>
    </thead>
    <tbody>
        @foreach (Person p in Model.People) {
            <tr class="@Model.GetClass(p.Location.City)">
                <td>@p.PersonId</td>
                <td>@p.Surname, @p.Firstname</td>
                <td>@p.Department.Name</td>
                <td>@p.Location.City, @p.Location.State</td>
            </tr>
        }
    </tbody>
</table>

<form asp-action="Index" method="get">
    <div class="form-group">
        <label for="selectedCity">City</label>
        <select name="selectedCity" class="form-control">
            <option disabled selected>Select City</option>
            @foreach (string city in Model.Cities) {
                <option selected="@(city == Model.SelectedCity)">
                    @city
                </option>
            }
        </select>
    </div>
    <button class="btn btn-primary" type="submit">Select</button>
</form>
```

To enable tag helpers and add the namespaces that will be available by default in views, add a Razor View Imports file named _ViewImports.cshtml to the Views folder with the content shown in Listing 32-18.

Listing 32-18. The Contents of the _ViewImports.cshtml File in the Views Folder

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@using Advanced.Models
@using Advanced.Controllers
```

To specify the default layout for controller views, add a Razor View Start start file named _ViewStart.cshtml to the Views folder with the content shown in Listing 32-19.

Listing 32-19. The Contents of the _ViewStart.cshtml File in the Views Folder

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

To create the layout, create the Views/Shared folder and add to it a Razor Layout named _Layout.cshtml with the content shown in Listing 32-20.

Listing 32-20. The Contents of the _Layout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="m-2">
        @RenderBody()
    </div>
</body>
</html>
```

Creating a Razor Page

To display the application's data using a Razor Page, create the Pages folder and add to it a Razor Page named Index.cshtml with the content shown in Listing 32-21.

Listing 32-21. The Contents of the Index.cshtml File in the Pages Folder

```
@page "/pages"
@model IndexModel

<h4 class="bg-primary text-white text-center p-2">People</h4>

<table class="table table-sm table-bordered table-striped">
    <thead>
        <tr>
            <th>ID</th><th>Name</th><th>Dept</th><th>Location</th>
        </tr>
    </thead>
    <tbody>
        @foreach (Person p in Model.People) {
            <tr class="@Model.GetClass(p.Location.City)">
                <td>@p.PersonId</td>
                <td>@p.Surname, @p.Firstname</td>
                <td>@p.Department.Name</td>
                <td>@p.Location.City, @p.Location.State</td>
            </tr>
        }
    </tbody>
</table>

<form asp-page="Index" method="get">
    <div class="form-group">
        <label for="selectedCity">City</label>
        <select name="selectedCity" class="form-control">
            <option disabled selected>Select City</option>
            @foreach (string city in Model.Cities) {
                <option selected="@(city == Model.SelectedCity)">
                    @city
                </option>
            }
```

```
        </select>
    </div>
    <button class="btn btn-primary" type="submit">Select</button>
</form>

@functions {

    public class IndexModel: PageModel {
        private DataContext context;

        public IndexModel(DataContext dbContext) {
            context = dbContext;
        }

        public IEnumerable<Person> People { get; set; }

        public IEnumerable<string> Cities { get; set; }

        [FromQuery]
        public string SelectedCity { get; set; }

        public void OnGet() {
            People = context.People.Include(p => p.Department)
                .Include(p => p.Location);
            Cities = context.Locations.Select(l => l.City).Distinct();
        }

        public string GetClass(string city) =>
            SelectedCity == city ? "bg-info text-white" : "";
    }
}
```

To enable tag helpers and add the namespaces that will be available by default in the view section of the Razor Pages, add a Razor view imports file named `_ViewImports.cshtml` to the `Pages` folder with the content shown in Listing 32-22.

Listing 32-22. The Contents of the _ViewImports.cshtml File in the Pages Folder

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@using Advanced.Models
@using Microsoft.AspNetCore.Mvc.RazorPages
@using Microsoft.EntityFrameworkCore
```

To specify the default layout for Razor Pages, add a Razor View Start file named `_ViewStart.cshtml` to the `Pages` folder with the content shown in Listing 32-23.

Listing 32-23. The Contents of the _ViewStart.cshtml File in the Pages Folder

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

To create the layout, add a Razor Layout named `_Layout.cshtml` to the `Pages` folder with the content shown in Listing 32-24.

Listing 32-24. The Contents of the _Layout.cshtml File in the Pages Folder

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
```

```
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="m-2">
        <h5 class="bg-secondary text-white text-center p-2">Razor Page</h5>
        @RenderBody()
    </div>
</body>
</html>
```

Running the Example Application

Start the application, either by selecting Start Without Debugging or Run Without Debugging from the Debug menu or by running the command shown in Listing 32-25 in the Advanced project folder.

Listing 32-25. Running the Example Application

```
dotnet run
```

Use a browser to request `http://localhost:5000/controllers` and `http://localhost:5000/pages`. Select a city using the select element and click the Select button to highlight rows in the table, as shown in Figure 32-3.

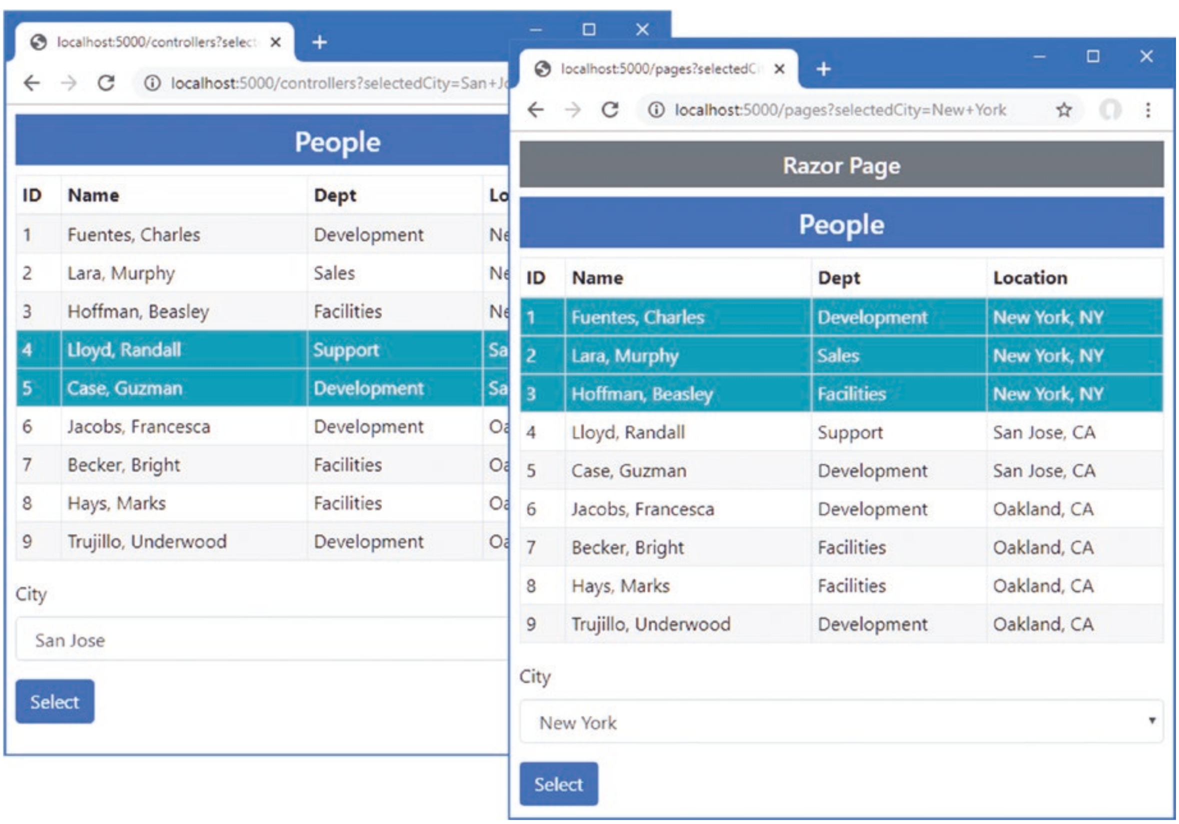

Figure 32-3. *Running the example application*

Summary

In this chapter, I showed how to create the example application that is used throughout this part of the book. The project was created with the Empty template, and it contains a data model that relies on Entity Framework Core and handles requests using a controller and a Razor Page. In the next chapter, I introduce Blazor, which is a new addition to ASP.NET Core.

CHAPTER 33

Using Blazor Server, Part 1

Blazor is a new addition to ASP.NET Core that adds client-side interactivity to web applications. There are two varieties of Blazor, and in this chapter, I focus on Blazor Server. I explain the problem it solves and how it works. I show you how to configure an ASP.NET Core application to use Blazor Server and describe the basic features available when using Razor Components, which are the building blocks for Blazor Server projects. I describe more advanced Blazor Server features in Chapters 34–36, and in Chapter 37, I describe Blazor WebAssembly, which is the other variety of Blazor. Table 33-1 puts Blazor Server in context.

Table 33-1. *Putting Blazor Server in Context*

Question	Answer
What is it?	Blazor Server uses JavaScript to receive browser events, which are forwarded to ASP.NET Core and evaluated using C# code. The effect of the event on the state of the application is sent back to the browser and displayed to the user.
Why is it useful?	Blazor Server can produce a richer and more responsive user experience compared to standard web applications.
How is it used?	The building block for Blazor Server is the Razor Component, which uses a syntax similar to Razor Pages. The view section of the Razor Component contains special attributes that specify how the application will respond to user interaction.
Are there any pitfalls or limitations?	Blazor Server relies on a persistent HTTP connection to the server and cannot function when that connection is interrupted. Blazor Server is not supported by older browsers.
Are there any alternatives?	The features described in Part 3 of this book can be used to create web applications that work broadly but that offer a less responsive experience. You could also consider a client-side JavaScript framework, such as Angular, React, or Vue.js.

Table 33-2 summarizes the chapter.

Table 33-2. *Chapter Summary*

Problem	Solution	Listing
Configuring Blazor	Use the `AddServerSideBlazor` and `MapBlazorHub` methods to set up the required services and middleware and configure the JavaScript file	3–6
Creating a Blazor Component	Create a `.blazor` file and use it to define code and markup	7
Applying a component	Use a `component` element	8, 9
Handling events	Use an attribute to specify the method or expression that will handle an event	10–15
Creating a two-way relationship with an element	Create a data binding	16–20
Defining the code separately from the markup	Use a code-behind class	21–23
Defining a component without declarative markup	Use a Razor Component class	24, 25

A. Freeman, *Pro ASP.NET Core 3*, https://doi.org/10.1007/978-1-4842-5440-0_33

Preparing for This Chapter

This chapter uses the Advanced project from Chapter 32. No changes are required to prepare for this chapter.

■ **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-3`. See Chapter 1 for how to get help if you have problems running the examples.

Open a new PowerShell command prompt, navigate to the folder that contains the `Advanced.csproj` file, and run the command shown in Listing 33-1 to drop the database.

Listing 33-1. Dropping the Database

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

Select Start Without Debugging or Run Without Debugging from the Debug menu or use the PowerShell command prompt to run the command shown in Listing 33-2.

Listing 33-2. Running the Example Application

```
dotnet run
```

Use a browser to request `http://localhost:5000/controllers`, which will display a list of data items. Pick a city from the drop-down list and click the Select button to highlight elements, as shown in Figure 33-1.

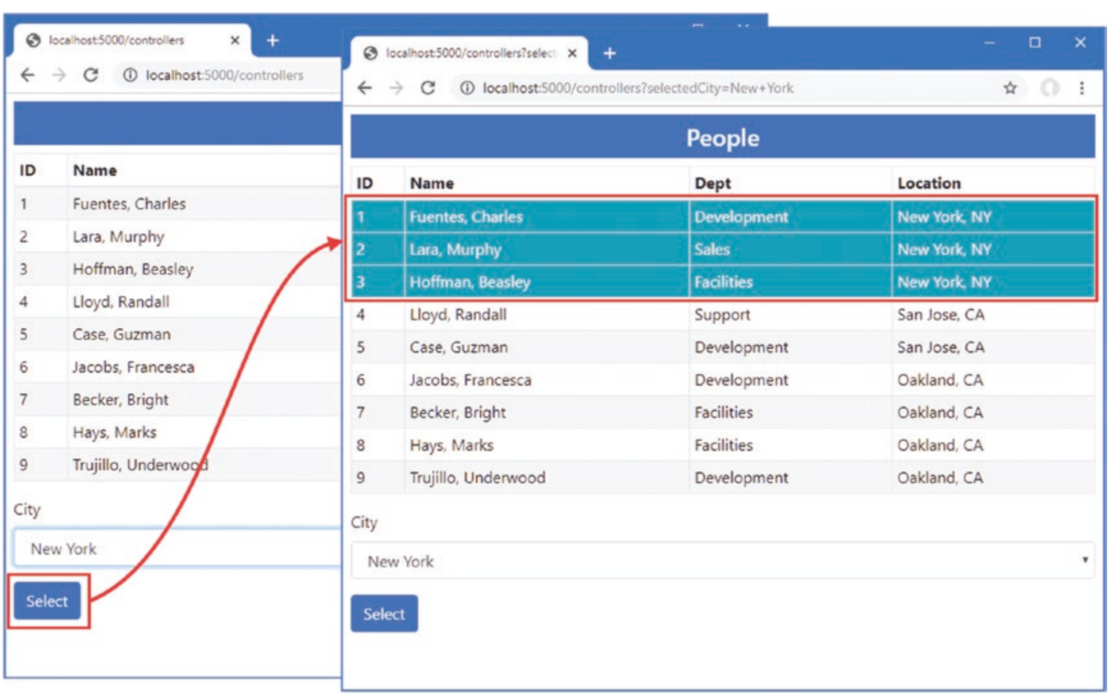

Figure 33-1. *Running the example application*

Understanding Blazor Server

Consider what happens when you choose a city and click the Select button presented by the example application. The browser sends an HTTP GET request that submits a form, which is received by either an action method or a handler method, depending on whether you use the controller or Razor Page. The action or handler renders its view, which sends a new HTML document that reflects the selection to the browser, as illustrated by Figure 33-2.

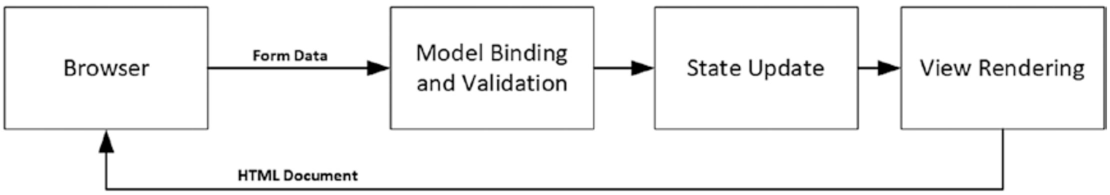

Figure 33-2. *Interacting with the example application*

This cycle is effective but can be inefficient. Each time the Submit button is clicked, the browser sends a new HTTP request to ASP.NET Core. Each request contains a complete set of HTTP headers that describe the request and the types of responses the browser is willing to receive. In its response, the server includes HTTP headers that describe the response and includes a complete HTML document for the browser to display.

The amount of data sent by the example application is about 3KB on my system, and almost all of it is duplicated between requests. The browser only wants to tell the server which city has been selected, and the server only wants to indicate which table rows should be highlighted; however, each HTTP request is self-contained, so the browser must parse a complete HTML document each time. The root issue that every interaction is the same: send a request and get a complete HTML document in return.

Blazor takes a different approach. A JavaScript library is included in the HTML document that is sent to the browser. When the JavaScript code is executed, it opens an HTTP connection back to the server and leaves it open, ready for user interaction. When the user picks a value using the select element, for example, details of the selection are sent to the server, which responds with just the changes to apply to the existing HTML, as shown in Figure 33-3.

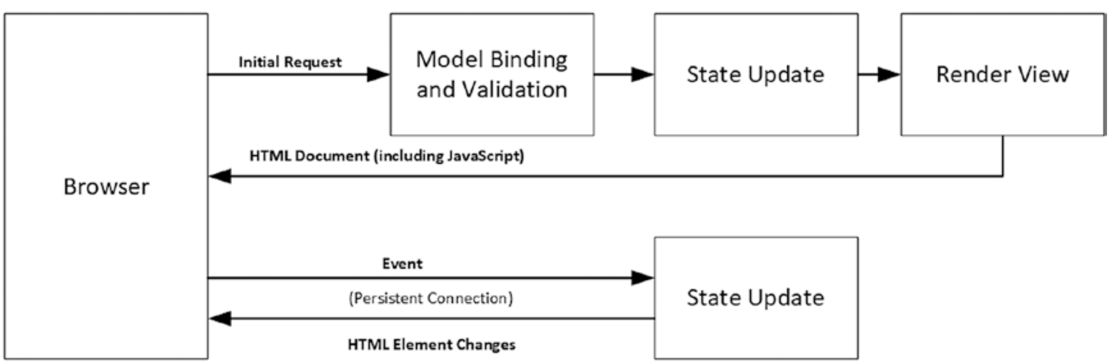

Figure 33-3. *Interacting with Blazor*

The persistent HTTP connection minimizes the delay, and replying with just the differences reduces the amount of data sent between the browser and the server.

Understanding the Blazor Server Advantages

The biggest attraction of Blazor is that it is based on Razor Pages written in C#. This means you can increase efficiency and responsiveness without having to learn a new framework, such as Angular or React, and a new language, such as TypeScript or JavaScript. Blazor is nicely integrated into the rest of ASP.NET Core and is built on features described in earlier chapters, which makes it easy to use (especially when compared to a framework like Angular, which has a dizzyingly steep learning curve).

Understanding the Blazor Server Disadvantages

Blazor requires a modern browser to establish and maintain its persistent HTTP connection. And, because of this connection, applications that use Blazor stop working if the connection is lost, which makes them unsuitable for offline use, where connectivity cannot be relied on or where connections are slow. These issues are addressed by Blazor WebAssembly, described in Chapter 36, but, as I explain, this has its own set of limitations.

Choosing Between Blazor Server and Angular/React/Vue.js

Decisions between Blazor and one of the JavaScript frameworks should be driven by the development team's experience and the users' expected connectivity. If you have no JavaScript expertise and have not used one of the JavaScript frameworks, then you should use Blazor, but only if you can rely on good connectivity and modern browsers. This makes Blazor a good choice for line-of-business applications, for example, where the browser demographic and network quality can be determined in advance.

If you have JavaScript experience and you are writing a public-facing application, then you should use one of the JavaScript frameworks because you won't be able to make assumptions about browsers or network quality. (It doesn't matter which framework you choose—I have written books about Angular, React, and View, and they are all excellent. My advice for choosing a framework is to create a simple app in each of them and pick the one whose development model appeals to you the most.)

If you are writing a public-facing application and you don't have JavaScript experience, then you have two choices. The safest option is to stick to the ASP.NET Core features described in earlier chapters and accept the inefficiencies this can bring. This isn't a terrible choice to make, and you can still produce top-quality applications. A more demanding choice is to learn TypeScript or JavaScript and one Angular, React, or Vue.js—but don't underestimate the amount of time it takes to master JavaScript or the complexity of these frameworks.

Getting Started with Blazor

The best way to get started with Blazor is to jump right in. In the sections that follow, I configure the application to enable Blazor and re-create the functionality offered by the controller and Razor Page. After that, I'll go right back to basics and explain how Razor Components work and the different features they offer.

Configuring ASP.NET Core for Blazor Server

Preparation is required before Blazor can be used. The first step is to add the services and middleware to the `Startup` class, as shown in Listing 33-3.

Listing 33-3. Adding Services and Middleware in the Startup.cs File in the Advanced Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
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 Advanced.Models;

namespace Advanced {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }
```

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

        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<DataContext>(opts => {
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:PeopleConnection"]);
                opts.EnableSensitiveDataLogging(true);
            });
            services.AddControllersWithViews().AddRazorRuntimeCompilation();
            services.AddRazorPages().AddRazorRuntimeCompilation();
            services.AddServerSideBlazor();
        }

        public void Configure(IApplicationBuilder app, DataContext context) {

            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseRouting();

            app.UseEndpoints(endpoints => {
                endpoints.MapControllerRoute("controllers",
                    "controllers/{controller=Home}/{action=Index}/{id?}");
                endpoints.MapDefaultControllerRoute();
                endpoints.MapRazorPages();
                endpoints.MapBlazorHub();
            });

            SeedData.SeedDatabase(context);
        }
    }
}
```

The “hub” in the MapBlazorHub method relates to SignalR, which is the part of ASP.NET Core that handles the persistent HTTP request. I don’t describe SignalR in this book because it is rarely used directly, but it can be useful if you need ongoing communication between clients and the server. See https://docs.microsoft.com/en-gb/aspnet/core/signalr for details. For this book—and most ASP.NET Core applications—it is enough to know that SignalR is used to manage the connections that Blazor relies on.

Adding the Blazor JavaScript File to the Layout

Blazor relies on JavaScript code to communicate with the ASP.NET Core server. Add the elements shown in Listing 33-4 to the _Layout.cshtml file in the Views/Shared folder to add the JavaScript file to the layout used by controller views.

Listing 33-4. Adding Elements in the _Layout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
    <base href="~/" />
</head>
<body>
    <div class="m-2">
        @RenderBody()
    </div>
    <script src="_framework/blazor.server.js"></script>
</body>
</html>
```

The `script` element specifies the name of the JavaScript file, and requests for it are intercepted by the middleware added to the request pipeline in Listing 33-3 so that no additional package is required to add the JavaScript code to the project. The `base` element must also be added to specify the root URL for the application. The same elements must be added to the layout used by Razor Pages, as shown in Listing 33-5.

Listing 33-5. Adding Elements in the _Layout.cshtml File in the Pages Folder

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
    <base href="~/" />
</head>
<body>
    <div class="m-2">
        <h5 class="bg-secondary text-white text-center p-2">Razor Page</h5>
        @RenderBody()
    </div>
    <script src="_framework/blazor.server.js"></script>
</body>
</html>
```

Creating the Blazor Imports File

Blazor requires its own imports file to specify the namespaces that it uses. It is easy to forget to add this file to a project, but, without it, Blazor will silently fail. Add a file named `_Imports.razor` to the `Advanced` folder with the content shown in Listing 33-6. (If you are using Visual Studio, you can use the Razor View Imports template to create this file, but ensure you use the `.razor` file extension.)

Listing 33-6. The Contents of the _Imports.razor File in the Advanced Folder

```
@using Microsoft.AspNetCore.Components
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.JSInterop
@using Microsoft.EntityFrameworkCore
@using Advanced.Models
```

The first five `@using` expressions are for the namespaces required for Blazor. The last two expressions are for convenience in the examples that follow because they will allow me to use Entity Framework Core and the classes in the `Models` namespace.

Creating a Razor Component

There is a clash in terminology: the technology is *Blazor*, but the key building block is called a *Razor Component*. Razor Components are defined in files with the `.razor` extension and must begin with a capital letter. Components can be defined anywhere, but they are usually grouped together to help keep the project organized. Create a `Blazor` folder in the `Advanced` folder and add to it a Razor Component named `PeopleList.razor` with the content shown in Listing 33-7.

Listing 33-7. The Contents of the PeopleList.razor File in the Blazor Folder

```
<table class="table table-sm table-bordered table-striped">
    <thead>
        <tr>
            <th>ID</th><th>Name</th><th>Dept</th><th>Location</th>
        </tr>
    </thead>
    <tbody>
```

```
        @foreach (Person p in People) {
            <tr class="@GetClass(p.Location.City)">
                <td>@p.PersonId</td>
                <td>@p.Surname, @p.Firstname</td>
                <td>@p.Department.Name</td>
                <td>@p.Location.City, @p.Location.State</td>
            </tr>
        }
    </tbody>
</table>

<div class="form-group">
    <label for="city">City</label>
    <select name="city" class="form-control" @bind="SelectedCity">
        <option disabled selected>Select City</option>
        @foreach (string city in Cities) {
            <option value="@city" selected="@(city == SelectedCity)">
                @city
            </option>
        }
    </select>
</div>

@code {

    [Inject]
    public DataContext Context { get; set; }

    public IEnumerable<Person> People =>
        Context.People.Include(p => p.Department).Include(p => p.Location);

    public IEnumerable<string> Cities => Context.Locations.Select(l => l.City);

    public string SelectedCity { get; set; }

    public string GetClass(string city) =>
        SelectedCity == city ? "bg-info text-white" : "";
}
```

Razor Components are similar to Razor Pages. The view section relies on the Razor features you have seen in earlier chapters, with `@` expressions to insert data values into the component's HTML or to generate elements for objects in a sequence, like this:

```
...
@foreach (string city in Cities) {
    <option value="@city" selected="@(city == SelectedCity)">
}
...
```

This `@foreach` expression generates `option` elements for each value in the `Cities` sequence and is identical to the equivalent expression in the controller view and Razor Page created in Chapter 32.

Although Razor Components look familiar, there are some important differences. The first is that there is no page model class and no @model expression. The properties and methods that support a component's HTML are defined directly in an @code expression, which is the counterpart to the Razor Page @functions expression. To define the property that will provide the view section with Person objects, for example, I just define a People property in the @code section, like this:

```
...
public IEnumerable<Person> People =>
    Context.People.Include(p => p.Department).Include(p => p.Location);
...
```

And, because there is no page model class, there is no constructor through which to declare service dependencies. Instead, the dependency injection sets the values of properties that have been decorated with the Inject attribute, like this:

```
...
[Inject]
public DataContext Context { get; set; }
...
```

The most significant difference is the use of a special attribute on the select element.

```
...
<select name="city" class="form-control" @bind="SelectedCity">
    <option disabled selected>Select City</option>
...
```

This Blazor attribute creates a data binding between the value of the select element and the SelectedCity property defined in the @code section.

I describe data bindings in more detail in the "Working with Data Bindings" section, but for now, it is enough to know that the value of the SelectedCity will be updated when the user changes the value of the select element.

Using a Razor Component

Razor components are delivered to the browser as part of a Razor Page or a controller view. Listing 33-8 shows how to use a Razor Component in a controller view.

Listing 33-8. Using a Razor Component in the Index.cshtml File in the Views/Home Folder

```
@model PeopleListViewModel

<h4 class="bg-primary text-white text-center p-2">People</h4>

<component type="typeof(Advanced.Blazor.PeopleList)" render-mode="Server" />
```

Razor Components are applied using the component element, for which there is a tag helper. The component element is configured using the type and render-mode attributes. The type attribute is used to specify the Razor Component. Razor Components are compiled into classes just like controller views and Razor Pages. The PeopleList component is defined in the Blazor folder in the Advanced project, so the type will be Advanced.Blazor.PeopleList, like this:

```
...
<component type="typeof(Advanced.Blazor.PeopleList)" render-mode="Server" />
...
```

The render-mode attribute is used to select how content is produced by the component, using a value from the RenderMode enum, described in Table 33-3.

***Table 33-3.** The RenderMode Values*

Name	Description
`Static`	The Razor Component renders its view section as static HTML with no client-side support.
`Server`	The HTML document is sent to the browser with a placeholder for the component. The HTML displayed by the component is sent to the browser over the persistent HTTP connection and displayed to the user.
`ServerPrerendered`	The view section of the component is included in the HTML and displayed to the user immediately. The HTML content is sent again over the persistent HTTP connection.

For most applications, the `Server` option is a good choice. The `ServerPrerendered` includes a static rendition of the Razor Component's view section in the HTML document sent to the browser. This acts as placeholder content so that the user isn't presented with an empty browser window while the JavaScript code is loaded and executed. Once the persistent HTTP connection has been established, the placeholder content is deleted and replaced with a dynamic version sent by Blazor. The idea of showing static content to the user is a good one, but it can be confusing because the HTML elements are not wired up to the server-side part of the application, and any interaction from the user either doesn't work or will be discarded once the live content arrives.

To see Blazor in action, restart ASP.NET Core and use a browser to request `http://localhost:5000/controllers`. No form submission is required when using Blazor because the data binding will respond as soon as the `select` element's value is changed, as shown in Figure 33-4.

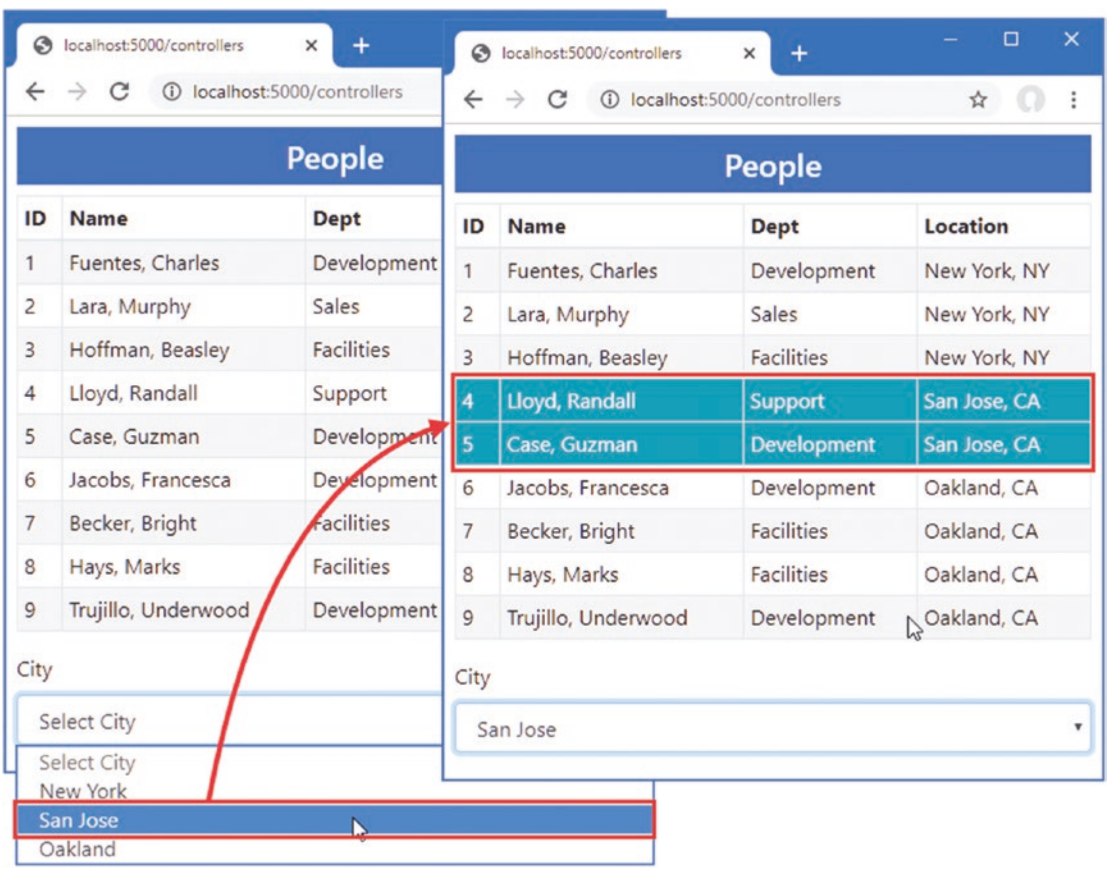

***Figure 33-4.** Using a Razor Component*

When you use the `select` element, the value you choose is sent over the persistent HTTP connection to the ASP.NET Core server, which updates the Razor Component's `SelectedCity` property and rerenders the HTML content. A set of updates is sent to the JavaScript code, which updates the table.

Razor Components can also be used in Razor Pages. Add a Razor Page named `Blazor.cshtml` to the `Pages` folder and add the content shown in Listing 33-9.

Listing 33-9. The Contents of the Blazor.cshtml File in the Pages Folder

```
@page "/pages/blazor"

<script type="text/javascript">
    window.addEventListener("DOMContentLoaded", () => {
        document.getElementById("markElems").addEventListener("click", () => {
            document.querySelectorAll("td:first-child")
                .forEach(elem => {
                    elem.innerText = `M:${elem.innerText}`
                    elem.classList.add("border", "border-dark");
                });
        });
    });
</script>

<h4 class="bg-primary text-white text-center p-2">Blazor People</h4>

<button id="markElems" class="btn btn-outline-primary mb-2">Mark Elements</button>

<component type="typeof(Advanced.Blazor.PeopleList)" render-mode="Server" />
```

The Razor Page in Listing 33-9 contains additional JavaScript code that helps demonstrate that only changes are sent, instead of an entirely new HTML table. Restart ASP.NET Core and request `http://localhost:5000/pages/blazor`. Click the Mark Elements button, and the cells in the ID column will be changed to display different content and a border. Now use the `select` element to pick a different city, and you will see that the elements in the table are modified without being deleted, as shown in Figure 33-5.

UNDERSTANDING BLAZOR CONNECTION MESSAGES

When you stop ASP.NET Core, you will see an error message in the browser window, which indicates the connection to the server has been lost and prevents the user from interacting with the displayed component. Blazor will attempt to reconnect and pick up where it left off when the disconnection is caused by temporary network issues, but it won't be able to do so when the server has been stopped or restarted because the context data for the connection has been lost; you will have to explicitly request a new URL.

There is a default reload link in the connection message, but that goes to the default URL for the website, which isn't useful for this book where I direct you to specific URLs to see the effect of examples. See Chapter 34 for details of how to configure the connection messages.

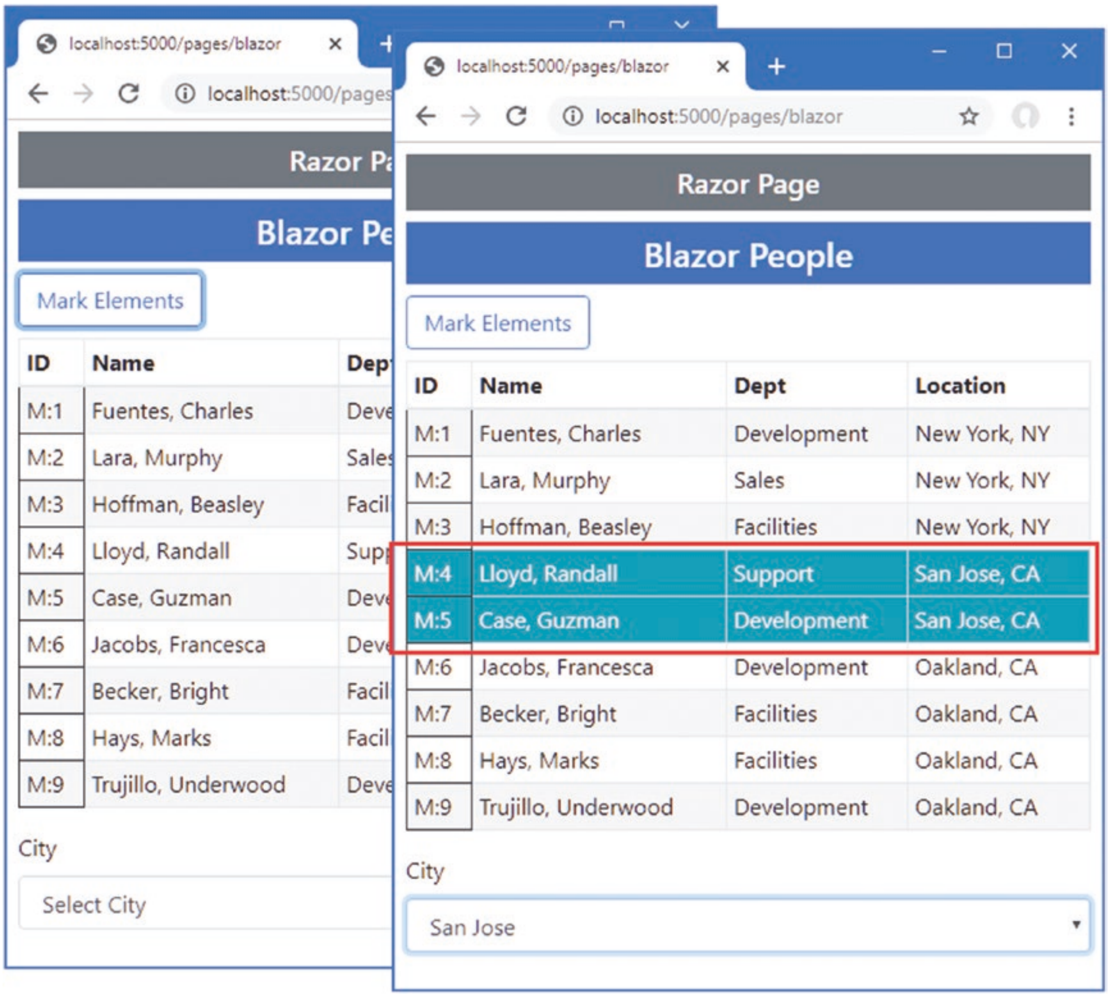

Figure 33-5. *Demonstrating that only changes are used*

Understanding the Basic Razor Component Features

Now that I have demonstrated how Blazor can be used and how it works, it is time to go back to the basics and introduce the features that Razor Components offer. Although the example in the previous section showed how standard ASP.NET Core features can be reproduced using Blazor, there is a much wider set of features available.

Understanding Blazor Events and Data Bindings

Events allow a Razor Component to respond to user interaction, and Blazor uses the persistent HTTP connection to send details of the event to the server where it can be processed. To see Blazor events in action, add a Razor Component named `Events.razor` to the `Blazor` folder with the content shown in Listing 33-10.

Listing 33-10. The Contents of the Events.razor File in the Blazor Folder

```
<div class="m-2 p-2 border">
    <button class="btn btn-primary" @onclick="IncrementCounter">Increment</button>
    <span class="p-2">Counter Value: @Counter</span>
</div>
```

```
@code {
    public int Counter { get; set; } = 1;

    public void IncrementCounter(MouseEventArgs e) {
        Counter++;
    }
}
```

You register a handler for an event by adding an attribute to an HTML element, where the attribute name is `@on`, followed by the event name. In the example, I have set up a handler for the `click` events generated by a `button` element, like this:

```
...
<button class="btn btn-primary" @onclick="IncrementCounter">Increment</button>
...
```

The value assigned to the attribute is the name of the method that will be invoked when the event is triggered. The method can define an optional parameter that is either an instance of the `EventArgs` class or a class derived from `EventArgs` that provides additional information about the event.

For the `onclick` event, the handler method receives a `MouseEventArgs` object, which provides additional details, such as the screen coordinates of the click. Table 33-4 lists the event description events and the events for which they are used.

Table 33-4. *The EventArgs Classes and the Events They Represent*

Class	Events
`ChangeEventArgs`	`onchange`, `oninput`
`ClipboardEventArgs`	`oncopy`, `oncut`, `onpaste`
`DragEventArgs`	`ondrag`, `ondragend`, `ondragenter`, `ondragleave`, `ondragover`, `ondragstart`, `ondrop`
`ErrorEventArgs`	`onerror`
`FocusEventArgs`	`onblur`, `onfocus`, `onfocusin`, `onfocusout`
`KeyboardEventArgs`	`onkeydown`, `onkeypress`, `onkeyup`
`MouseEventArgs`	`onclick`, `oncontextmenu`, `ondblclick`, `onmousedown`, `onmousemove`, `onmouseout`, `onmouseover`, `onmouseup`, `onmousewheel`, `onwheel`
`PointerEventArgs`	`ongotpointercapture`, `onlostpointercapture`, `onpointercancel`, `onpointerdown`, `onpointerenter`, `onpointerleave`, `onpointermove`, `onpointerout`, `onpointerover`, `onpointerup`
`ProgressEventArgs`	`onabort`, `onload`, `onloadend`, `onloadstart`, `onprogress`, `ontimeout`
`TouchEventArgs`	`ontouchcancel`, `ontouchend`, `ontouchenter`, `ontouchleave`, `ontouchmove`, `ontouchstart`
`EventArgs`	`onactivate`, `onbeforeactivate`, `onbeforecopy`, `onbeforecut`, `onbeforedeactivate`, `onbeforepaste`, `oncanplay`, `oncanplaythrough`, `oncuechange`, `ondeactivate`, `ondurationchange`, `onemptied`, `onended`, `onfullscreenchange`, `onfullscreenerror`, `oninvalid`, `onloadeddata`, `onloadedmetadata`, `onpause`, `onplay`, `onplaying`, `onpointerlockchange`, `onpointerlockerror`, `onratechange`, `onreadystatechange`, `onreset`, `onscroll`, `onseeked`, `onseeking`, `onselect`, `onselectionchange`, `onselectstart`, `onstalled`, `onstop`, `onsubmit`, `onsuspend`, `ontimeupdate`, `onvolumechange`, `onwaiting`

The Blazor JavaScript code receives the event when it is triggered and forwards it to the server over the persistent HTTP connection. The handler method is invoked, and the state of the component is updated. Any changes to the content produced by the component's view section will be sent back to the JavaScript code, which will update the content displayed by the browser.

In the example, the `click` event will be handled by the `IncrementCounter` method, which changes the value of the `Counter` property. The value of the `Counter` property is included in the HTML rendered by the component, so Blazor sends the changes to the browser so that the JavaScript code can update the HTML elements displayed to the user. To display the `Events` component, replace the contents of the `Blazor.cshtml` file in the `Pages` folder, as shown in Listing 33-11.

Listing 33-11. Using a New Component in the Blazor.cshtml File in the Pages Folder

```
@page "/pages/blazor"

<h4 class="bg-primary text-white text-center p-2">Events</h4>

<component type="typeof(Advanced.Blazor.Events)" render-mode="Server" />
```

Listing 33-11 changes the type attribute of the component element and removes the custom JavaScript and the button element I used to mark elements in the previous example. Restart ASP.NET Core and request http://localhost:5000/pages/blazor to see the new component. Click the Increment button, and the click event will be received by the Blazor JavaScript code and sent to the server for processing by the IncrementCounter method, as shown in Figure 33-6.

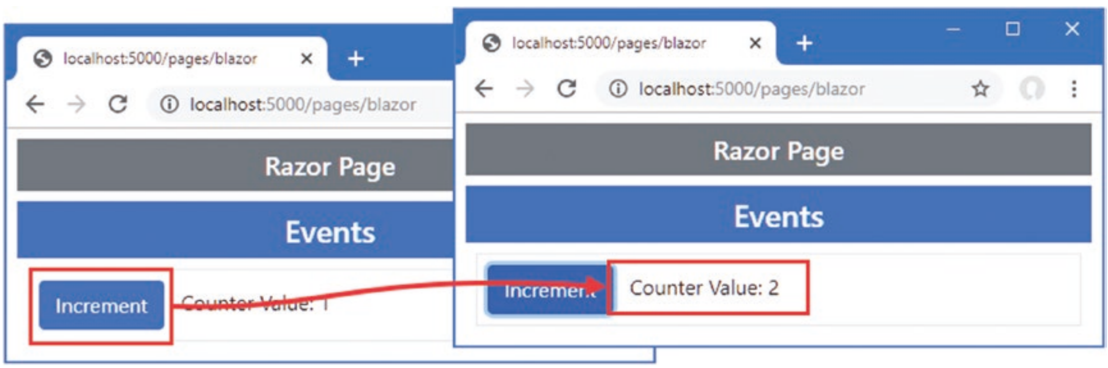

Figure 33-6. *Handling an event*

Handling Events from Multiple Elements

To avoid code duplication, elements from multiple elements can be received by a single handler method, as shown in Listing 33-12.

Listing 33-12. Handling Events in the Events.razor File in the Blazor Folder

```
<div class="m-2 p-2 border">
    <button class="btn btn-primary" @onclick="@(e => IncrementCounter(e, 0))">
        Increment Counter #1
    </button>
    <span class="p-2">Counter Value: @Counter[0]</span>
</div>

<div class="m-2 p-2 border">
    <button class="btn btn-primary" @onclick="@(e => IncrementCounter(e, 1))">
        Increment Counter #2
    </button>
    <span class="p-2">Counter Value: @Counter[1]</span>
</div>

@code {
    public int[] Counter { get; set; } = new int[] { 1, 1 };

    public void IncrementCounter(MouseEventArgs e, int index) {
        Counter[index]++;
    }
}
```

Blazor event attributes can be used with lambda functions that receive the `EventArgs` object and invoke a handler method with additional arguments. In this example, I have added an `index` parameter to the `IncrementCounter` method, which is used to determine which counter value should be updated. The value for the argument is defined in the `@onclick` attribute, like this:

```
...
<button class="btn btn-primary" @onclick="@(e => IncrementCounter(e, 0))">
...
```

This technique can also be used when elements are generated programmatically, as shown in Listing 33-13. In this example, I use an `@for` expression to generate elements and use the loop variable as the argument to the handler method. I have also removed the `EventArgs` parameter from the handler method, which isn't being used.

AVOIDING THE HANDLER METHOD NAME PITFALL

The most common mistake when specifying an event handler method is to include parentheses, like this:

```
...
<button class="btn btn-primary" @onclick="IncrementCounter()">
...
```

The error message this produces will depend on the event handler method. You may see a warning telling you a formal parameter is missing or that `void` cannot be converted to an `EventCallback`. When specifying a handler method, you must specify just the event name, like this:

```
...
<button class="btn btn-primary" @onclick="IncrementCounter">
...
```

You can specify the method name as a Razor expression, like this:

```
...
<button class="btn btn-primary" @onclick="@IncrementCounter">
...
```

Some developers find this easier to parse, but the result is the same. A different set of rules applies when using a lambda function, which must be defined within a Razor expression, like this:

```
...
<button class="btn btn-primary" @onclick="@( ... )">
...
```

Within the Razor expression, the lambda function is defined as it would be in a C# class, which means defining the parameters, followed by the "goes to" arrow, followed by the function body, like this:

```
...
<button class="btn btn-primary" @onclick="@((e) => HandleEvent(e, local))">
...
```

If you don't need to use the `EventArgs` object, then you can omit the parameter from the lambda function, like this:

```
...
<button class="btn btn-primary" @onclick="@(() => IncrementCounter(local))">
...
```

You will quickly become used to these rules as you start to work with Blazor, even if they seem inconsistent at first.

Listing 33-13. Generating Elements in the Events.razor File in the Blazor Folder

```
@for (int i = 0; i < ElementCount; i++) {
    int local = i;
    <div class="m-2 p-2 border">
        <button class="btn btn-primary" @onclick="@(() => IncrementCounter(local))">
            Increment Counter #@(i + 1)
        </button>
        <span class="p-2">Counter Value: @GetCounter(i)</span>
    </div>
}

@code {
    public int ElementCount { get; set; } = 4;

    public Dictionary<int, int> Counters { get; } = new Dictionary<int, int>();

    public int GetCounter(int index) =>
        Counters.ContainsKey(index) ? Counters[index] : 0;

    public void IncrementCounter(int index)  =>
        Counters[index] = GetCounter(index) + 1;
}
```

The important point to understand about event handlers is that the `@onclick` lambda function isn't evaluated until the server receives the `click` event from the browser. This means care must be taken not to use the loop variable `i` as the argument to the `IncrementCounter` method because it will always be the final value produced by the loop, which would be `4` in this case. Instead, you must capture the loop variable in a local variable, like this:

```
...
int local = i;
...
```

The local variable is then used as the argument to the event handler method in the attribute, like this:

```
...
<button class="btn btn-primary" @onclick="@(() => IncrementCounter(local))">
...
```

The local variable fixes the value for the lambda function for each of the generated elements. Restart ASP.NET Core and use a browser to request `http://localhost:5000/pages/blazor`, which will produce the response shown in Figure 33-7. The `click` events produced by all the `button` elements are handled by the same method, but the argument provided by the lambda function ensures that the correct counter is updated.

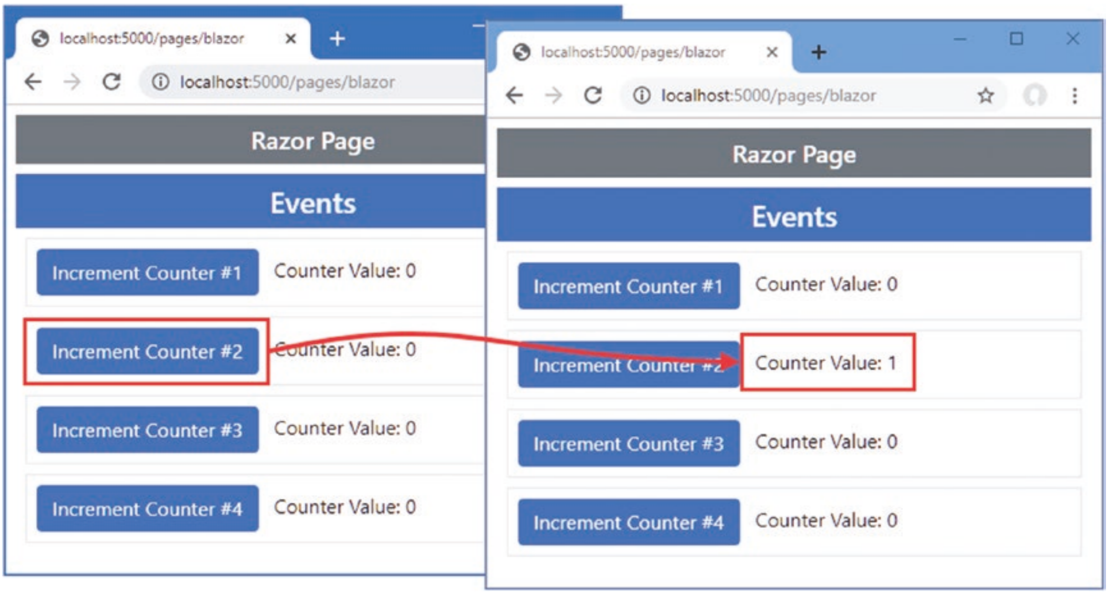

Figure 33-7. *Handling events from multiple elements*

Processing Events Without a Handler Method

Simple event handling can be done directly in a lambda function, without using a handler method, as shown in Listing 33-14.

Listing 33-14. Handling Events in the Events.razor File in the Blazor Folder

```
@for (int i = 0; i < ElementCount; i++) {
    int local = i;
    <div class="m-2 p-2 border">
        <button class="btn btn-primary" @onclick="@(() => IncrementCounter(local))">
            Increment Counter #@(i + 1)
        </button>
        <button class="btn btn-info" @onclick="@(() => Counters.Remove(local))">
                Reset
        </button>
        <span class="p-2">Counter Value: @GetCounter(i)</span>
    </div>
}

@code {
    public int ElementCount { get; set; } = 4;

    public Dictionary<int, int> Counters { get; } = new Dictionary<int, int>();

    public int GetCounter(int index) =>
        Counters.ContainsKey(index) ? Counters[index] : 0;

    public void IncrementCounter(int index)  =>
        Counters[index] = GetCounter(index) + 1;
}
```

Complex handlers should be defined as methods, but this approach is more concise for simple handlers. Restart ASP.NET Core and request `http://localhost:5000/pages/blazor`. The Reset buttons remove values from the `Counters` collection without relying on a method in the `@code` section of the component, as shown in Figure 33-8.

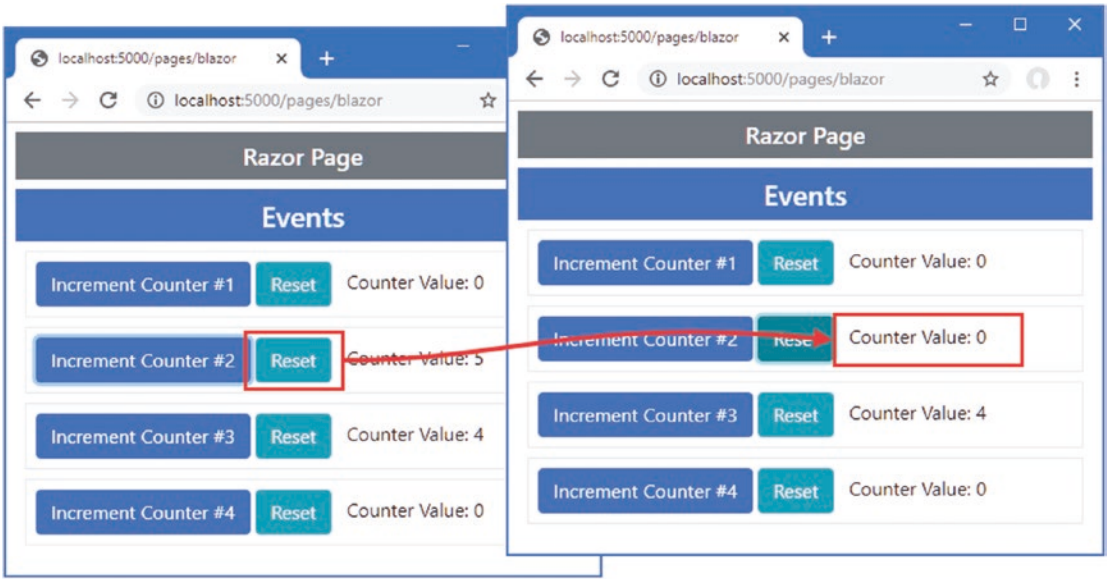

Figure 33-8. *Handling events in a lambda expression*

Preventing Default Events and Event Propagation

Blazor provides two attributes that alter the default behavior of events in the browser, as described in Table 33-5. These attributes, where the name of the event is followed by a colon and then a keyword, are known as *parameters*.

Table 33-5. *The Event Configuration Parameters*

Name	Description
`@on{event}:preventDefault`	This parameter determines whether the default event for an element is triggered.
`@on{event}:stopPropagation`	This parameter determines whether an event is propagated to its ancestor elements.

Listing 33-15 demonstrates what these parameters do and why they are useful.

Listing 33-15. Overriding Event Defaults in the Events.razor File in the Blazor Folder

```
<form action="/pages/blazor" method="get">
    @for (int i = 0; i < ElementCount; i++) {
        int local = i;
        <div class="m-2 p-2 border">
            <button class="btn btn-primary"
                @onclick="@(() => IncrementCounter(local))"
                @onclick:preventDefault="EnableEventParams">
                    Increment Counter #@(i + 1)
            </button>
            <button class="btn btn-info" @onclick="@(() => Counters.Remove(local))">
                    Reset
            </button>
            <span class="p-2">Counter Value: @GetCounter(i)</span>
        </div>
    }
</form>
```

```
<div class="m-2" @onclick="@(() => IncrementCounter(1))">
    <button class="btn btn-primary" @onclick="@(() => IncrementCounter(0))"
            @onclick:stopPropagation="EnableEventParams">Propagation Test</button>
</div>

<div class="form-check m-2">
    <input class="form-check-input" type="checkbox"
           @onchange="@(() => EnableEventParams = !EnableEventParams)" />
    <label class="form-check-label">Enable Event Parameters</label>
</div>

@code {
    public int ElementCount { get; set; } = 4;

    public Dictionary<int, int> Counters { get; } = new Dictionary<int, int>();

    public int GetCounter(int index) =>
        Counters.ContainsKey(index) ? Counters[index] : 0;

    public void IncrementCounter(int index)  =>
        Counters[index] = GetCounter(index) + 1;

    public bool EnableEventParams { get; set; } = false;
}
```

This example creates two situations in which the default behavior of events in the browser can cause problems. The first is caused by adding a `form` element. By default, `button` elements contained in a form will submit that form when they are clicked, even when the `@onclick` attribute is present. This means that whenever one of the Increment Counter buttons is clicked, the browser will send the form data to the ASP.NET Core server, which will respond with the contents of the `Blazor.cshtml` Razor Page.

The second problem is demonstrated by an element whose parent also defines an event handler, like this:

```
...
<div class="m-2" @onclick="@(() => IncrementCounter(1))">
    <button class="btn btn-primary" @onclick="@(() => IncrementCounter(0))"
...
```

Events go through a well-defined lifecycle in the browser, which includes being passed up the chain of ancestor elements. In the example, this means clicking the `button` will cause two counters to be updated, once by the `@onclick` handler for the `button` element and once by the `@onclick` handler for the enclosing `div` element.

To see these problems, restart ASP.NET Core and request `http://localhost:5000/pages/blazor`. Click an Increment Counter button; you will see that the form is submitted and the page is essentially reloaded. Click the Propagation Test button, and you will see that two counters are updated. Figure 33-9 shows both problems.

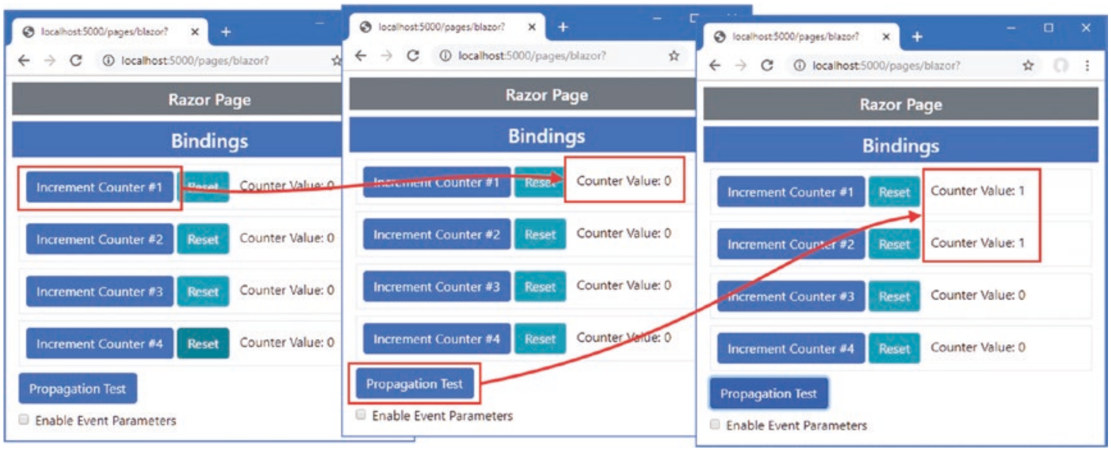

Figure 33-9. *Problems caused by the default behavior of events in the browser*

The checkbox in Listing 33-15 toggles the property that applies the parameters described in Table 33-5, with the effect that the form isn't submitted and only the handler on the button element receives the event. To see the effect, check the checkbox and then click an Increment Counter button and the Propagation Test buttons, which produces the result shown in Figure 33-10.

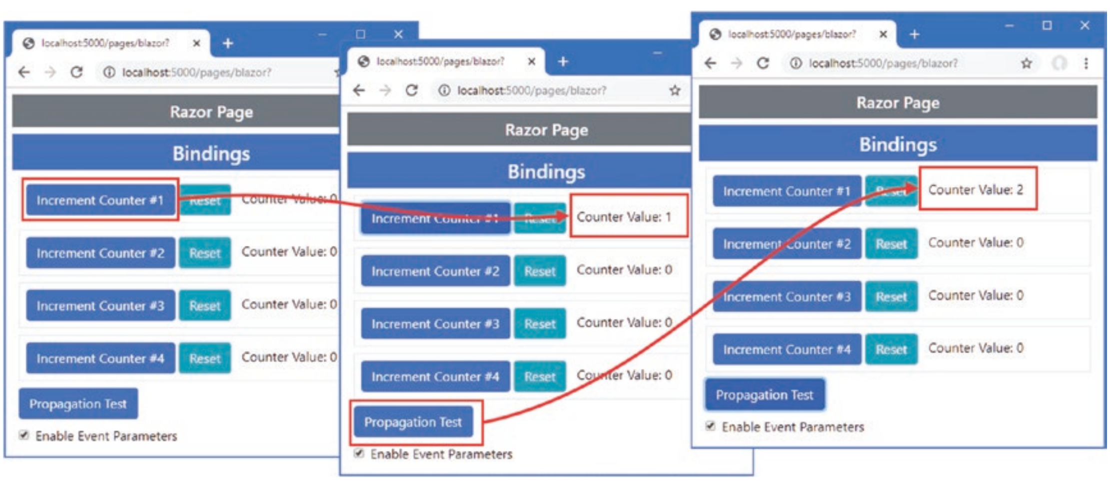

Figure 33-10. *Overriding the default behavior of events in the browser*

Working with Data Bindings

Event handlers and Razor expressions can be used to create a two-way relationship between an HTML element and a C# value, which is useful for elements that allow users to make changes, such as `input` and `select` elements. Add a Razor Component named `Bindings.razor` to the `Blazor` folder with the content shown in Listing 33-16.

Listing 33-16. The Contents of the Bindings.razor File in the Blazor Folder

```
<div class="form-group">
    <label>City:</label>
    <input class="form-control" value="@City" @onchange="UpdateCity" />
</div>
<div class="p-2 mb-2">City Value: @City</div>
```

```
<button class="btn btn-primary" @onclick="@(() => City = "Paris")">Paris</button>
<button class="btn btn-primary" @onclick="@(() => City = "Chicago")">Chicago</button>

@code {

    public string City { get; set; } = "London";

    public void UpdateCity(ChangeEventArgs e) {
        City = e.Value as string;
    }
}
```

The @onchange attribute registers the UpdateCity method as a handler for the change event from the input element. The events are described using the ChangeEventArgs class, which provides a Value property. Each time a change event is received, the City property is updated with the contents of the input element.

The input element's value attribute creates a relationship in the other direction so that when the value of the City property changes, so does the element's value attribute, which changes the text displayed to the user. To apply the new Razor Component, change the component attribute in the Razor Page, as shown in Listing 33-17.

Listing 33-17. Using a Razor Component in the Blazor.cshtml File in the Pages Folder

```
@page "/pages/blazor"

<h4 class="bg-primary text-white text-center p-2">Events</h4>

<component type="typeof(Advanced.Blazor.Bindings)" render-mode="Server" />
```

To see both parts of the relationship defined by the binding in Listing 33-16, restart ASP.NET Core, navigate to http://localhost:5000/pages/blazor, and edit the content of the input element. The change event is triggered only when the input element loses the focus, so once you have finished editing, press the Tab key or click outside of the input element; you will see the value you entered displayed through the Razor expression in the div element, as shown on the left of Figure 33-11. Click one of the buttons, and the City property will be changed to Paris or Chicago, and the selected value will be displayed by both the div element and the input element, as shown on the right of the figure.

Two-way relationships involving the change event can be expressed as data bindings, which allows both the value and the event to be configured with a single attribute, as shown in Listing 33-18.

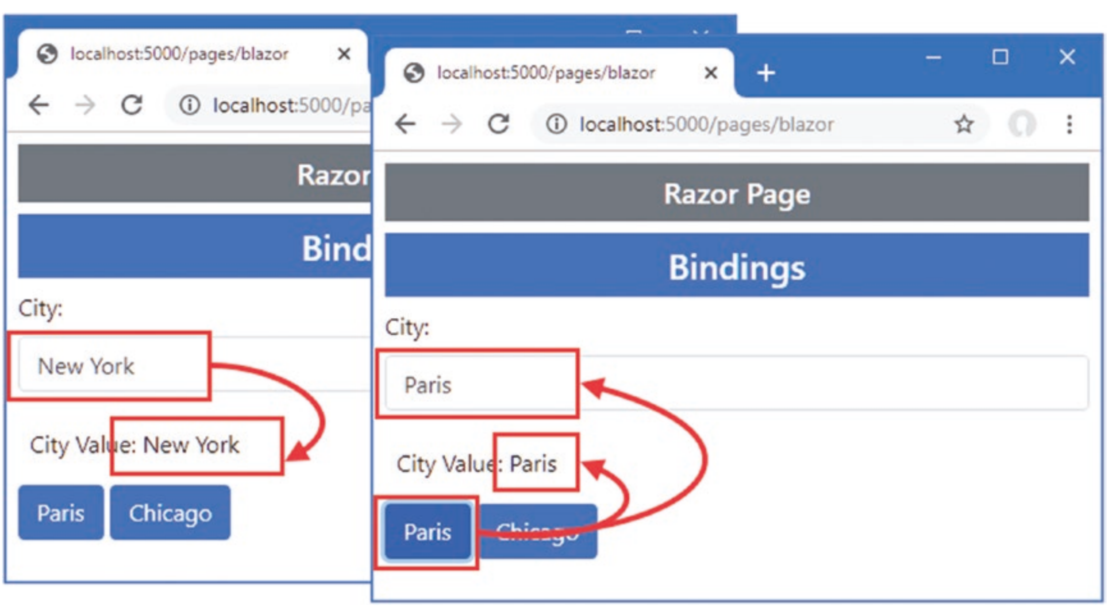

Figure 33-11. *Creating a two-way relationship between an element and a property*

Listing 33-18. Using a Data Binding in the Bindings.razor File in the Blazor Folder

```
<div class="form-group">
    <label>City:</label>
    <input class="form-control" @bind="City" />
</div>
<div class="p-2 mb-2">City Value: @City</div>
<button class="btn btn-primary" @onclick="@(() => City = "Paris")">Paris</button>
<button class="btn btn-primary" @onclick="@(() => City = "Chicago")">Chicago</button>

@code {

    public string City { get; set; } = "London";

    //public void UpdateCity(ChangeEventArgs e) {
    //    City = e.Value as string;
    //}
}
```

The `@bind` attribute is used to specify the property that will be updated when the change event is triggered and that will update the `value` attribute when it changes. The effect in Listing 33-18 is the same as Listing 33-16 but expressed more concisely and without the need for a handler method or a lambda function to update the property.

Changing the Binding Event

By default, the `change` event is used in bindings, which provides reasonable responsiveness for the user without requiring too many updates from the server. The event used in a binding can be changed by using the attributes described in Table 33-6.

These attributes are used instead of `@bind`, as shown in Listing 33-19, but can be used only with events that are represented with the `ChangeEventArgs` class. This means that only the `onchange` and `oninput` events can be used, at least in the current release.

Table 33-6. *The Binding Attributes for Specifying an Event*

Attribute	Description
`@bind-value`	This attribute is used to select the property for the data binding.
`@bind-value:event`	This attribute is used to select the event for the data binding.

Listing 33-19. Specifying an Event for a Binding in the Bindings.razor File in the Blazor Folder

```
<div class="form-group">
    <label>City:</label>
    <input class="form-control" @bind-value="City" @bind-value:event="oninput"  />
</div>
<div class="p-2 mb-2">City Value: @City</div>
<button class="btn btn-primary" @onclick="@(() => City = "Paris")">Paris</button>
<button class="btn btn-primary" @onclick="@(() => City = "Chicago")">Chicago</button>

@code {
    public string City { get; set; } = "London";
}
```

This combination of attributes creates a binding for the `City` property that is updated when the `oninput` event is triggered, which happens after every keystroke, rather than only when the `input` element loses the focus. To see the effect, restart ASP.NET Core, navigate to `http://localhost:5000/pages/blazor`, and start typing into the `input` element. The `City` property will be updated after every keystroke, as shown in Figure 33-12.

Creating DateTime Bindings

Blazor has special support for creating bindings for DateTime properties, allowing them to be expressed using a specific culture or a format string. This feature is applied using the parameters described in Table 33-7.

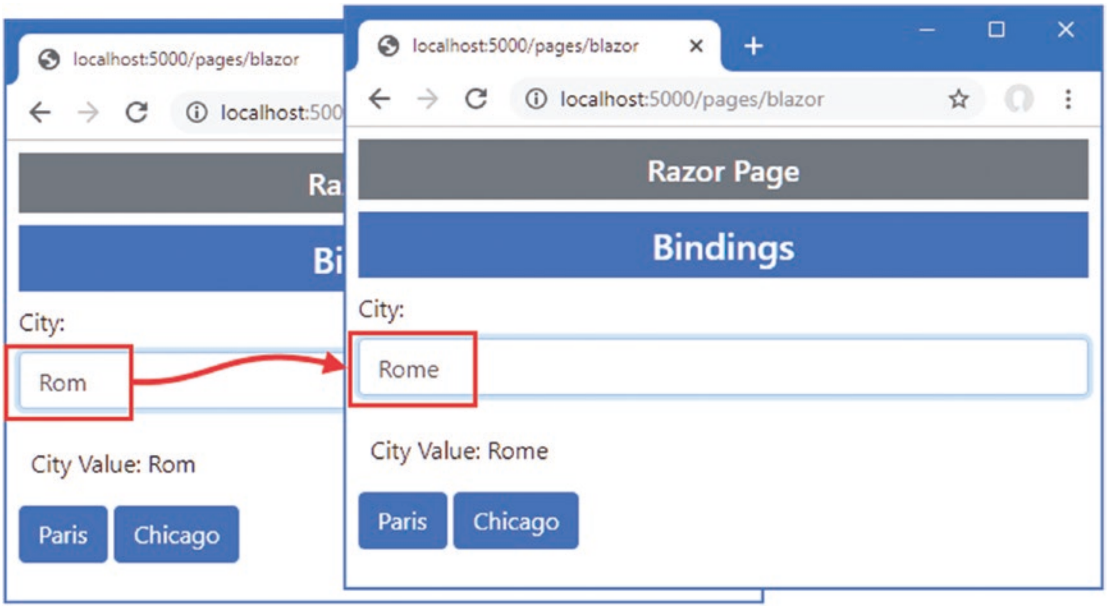

Figure 33-12. Changing the event in a data binding

■ **Tip** If you have used the @bind-value and @bind-value:event attributes to select an event, then you must use the @bind-value:culture and @bind-value:format parameters instead.

Table 33-7. The DateTime Parameters

Name	Description
@bind:culture	This attribute is used to select a CultureInfo object that will be used to format the DateTime value.
@bind:format	This attribute is used to specify a data formatting string that will be used to format the DateTime value.

Listing 33-20 shows the use of these attributes with a DateTime property.

■ **Note** The formatting strings used in these examples are described at https://docs.microsoft.com/en-us/dotnet/api/system.datetime?view=netcore-3.1.

Listing 33-20. Using a DateTime Property in the Bindings.razor File in the Blazor Folder

```
@using System.Globalization

<div class="form-group">
    <label>City:</label>
    <input class="form-control" @bind-value="City" @bind-value:event="oninput"  />
</div>
<div class="p-2 mb-2">City Value: @City</div>
<button class="btn btn-primary" @onclick="@(() => City = "Paris")">Paris</button>
<button class="btn btn-primary" @onclick="@(() => City = "Chicago")">Chicago</button>

<div class="form-group mt-2">
    <label>Time:</label>
    <input class="form-control my-1" @bind="Time" @bind:culture="Culture"
        @bind:format="MMM-dd" />
    <input class="form-control my-1" @bind="Time" @bind:culture="Culture" />
    <input class="form-control" type="date" @bind="Time" />
</div>
<div class="p-2 mb-2">Time Value: @Time</div>

<div class="form-group">
    <label>Culture:</label>
    <select class="form-control" @bind="Culture">
        <option value="@CultureInfo.GetCultureInfo("en-us")">en-US</option>
        <option value="@CultureInfo.GetCultureInfo("en-gb")">en-GB</option>
        <option value="@CultureInfo.GetCultureInfo("fr-fr")">fr-FR</option>
    </select>
</div>

@code {
    public string City { get; set; } = "London";

    public DateTime Time { get; set; } = DateTime.Parse("2050/01/20 09:50");

    public CultureInfo Culture { get; set; } = CultureInfo.GetCultureInfo("en-us");
}
```

There are three input elements that are used to display the same DataTime value, two of which have been configured using the attributes from Table 33-7. The first element has been configured with a culture and a format string, like this:

```
...
<input class="form-control my-1" @bind="Time" @bind:culture="Culture"
        @bind:format="MMM-dd" />
...
```

The DateTime property is displayed using the culture picked in the select element and with a format string that displays an abbreviated month name and the numeric date. The second input element specifies just a culture, which means the default formatting string will be used.

```
...
<input class="form-control my-1" @bind="Time" @bind:culture="Culture" />
...
```

To see how dates are displayed, restart ASP.NET Core, request http://localhost:5000/pages/blazor, and use the select element to pick different culture settings. The settings available represent English as it is used in the United States, English as it used in the United Kingdom, and French as it is used in France. Figure 33-13 shows the formatting each produces.

The initial locale in this example is en-US. When you switch to en-GB, the order in which the month and date appear changes. When you switch to en-FR, the abbreviated month name changes.

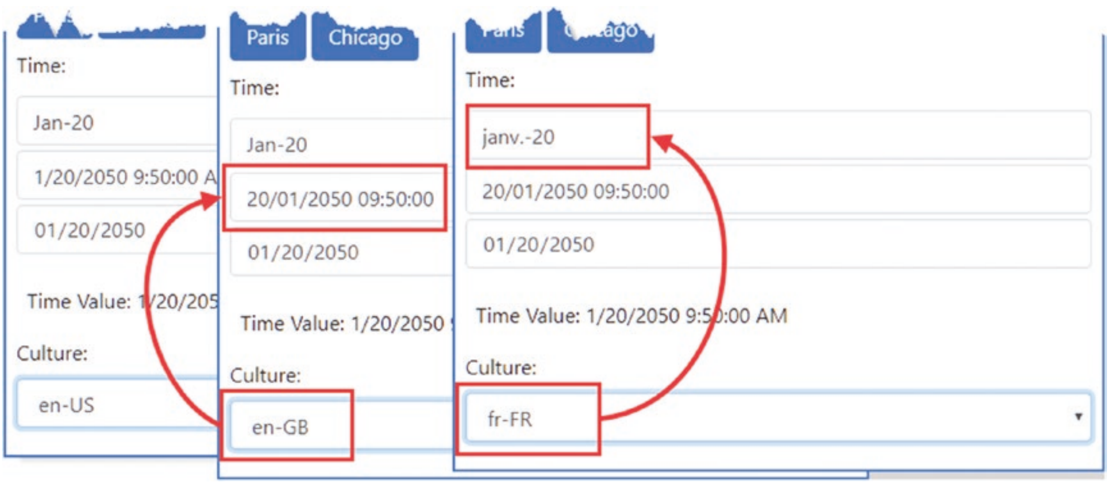

Figure 33-13. *Formatting DateTime values*

LETTING THE BROWSER FORMAT DATES

Notice that the value displayed by the third `input` element in Listing 33-20 doesn't change, regardless of the locale you choose. This `input` element has neither of the attributes described in Table 33-7 but does have its `type` attribute set to `date`, like this:

```
...
<input class="form-control" type="date" @bind="Time" />
...
```

You should not specify a culture or a format string when setting the type attribute to `date`, `datetime-local`, `month`, or `time`, because Blazor will automatically format date values into a culture-neutral format that the browser translates into the user's locale. Figure 33-11 shows how the date is formatted in the `en-US` locale but the user will see the date expressed in their local convention.

Using Class Files to Define Components

If you don't like the mix of code and markup that Razor Components support, you can use C# class files to define part, or all, of the component.

Using a Code-Behind Class

The `@code` section of a Razor Component can be defined in a separate class file, known as a *code-behind class* or *code-behind file*. Code-behind classes for Razor Components are defined as `partial` classes with the same name as the component they provide code for.

Add a Razor Component named `Split.razor` to the `Blazor` folder with the content shown in Listing 33-21.

Listing 33-21. The Contents of the Split.razor File in the Blazor Folder

```
<ul class="list-group">
    @foreach (string name in Names) {
        <li class="list-group-item">@name</li>
    }
</ul>
```

This file contains only HTML content and Razor expressions and renders a list of names that it expects to receive through a `Names` property. To provide the component with its code, add a class file named `Split.razor.cs` to the `Blazor` folder and use it to define the partial class shown in Listing 33-22.

Listing 33-22. The Contents of the Split.razor.cs File in the Blazor Folder

```
using Advanced.Models;
using Microsoft.AspNetCore.Components;
using System.Collections.Generic;
using System.Linq;

namespace Advanced.Blazor {

    public partial class Split {

        [Inject]
        public DataContext Context { get; set; }

        public IEnumerable<string> Names => Context.People.Select(p => p.Firstname);
    }
}
```

The partial class must be defined in the same namespace as its Razor Component and have the same name. For this example, that means the namespace is `Advanced.Blazor`, and the class name is `Splt`. Code-behind classes do not define constructors and receive services using the `Inject` attribute. Listing 33-23 applies the new component.

Listing 33-23. Applying a New Component in the Blazor.cshtml File in the Pages Folder

```
@page "/pages/blazor"

<h4 class="bg-primary text-white text-center p-2">Code-Behind</h4>

<component type="typeof(Advanced.Blazor.Split)" render-mode="Server" />
```

Restart ASP.NET Core and request `http://localhost:5000/pages/blazor`, and you will see the response shown in Figure 33-14.

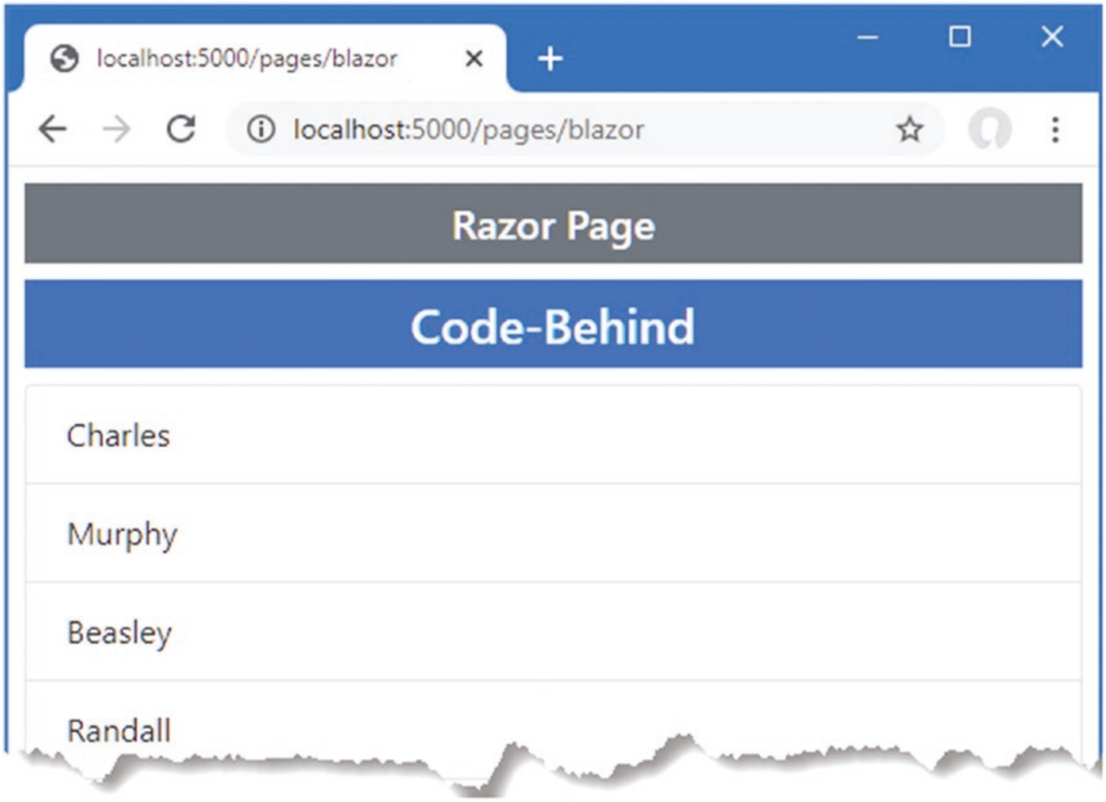

Figure 33-14. *Using a code-behind class to define a Razor Component*

Defining a Razor Component Class

Razor Components can be defined entirely in a class file, although this can be less expressive than using Razor expressions. Add a class file named `CodeOnly.cs` to the `Blazor` folder and use it to define the class shown in Listing 33-24.

Listing 33-24. The Contents of the CodeOnly.cs File in the Blazor Folder

```
using Advanced.Models;
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Rendering;
using Microsoft.AspNetCore.Components.Web;
using System.Collections.Generic;
using System.Linq;

namespace Advanced.Blazor {

    public class CodeOnly : ComponentBase {

        [Inject]
        public DataContext Context { get; set; }

        public IEnumerable<string> Names => Context.People.Select(p => p.Firstname);

        public bool Ascending { get; set; } = false;

        protected override void BuildRenderTree(RenderTreeBuilder builder) {
            IEnumerable<string> data = Ascending
                ? Names.OrderBy(n => n) : Names.OrderByDescending(n => n);
```

```
            builder.OpenElement(1, "button");
            builder.AddAttribute(2, "class", "btn btn-primary mb-2");
            builder.AddAttribute(3, "onclick",
                EventCallback.Factory.Create<MouseEventArgs>(this,
                    () => Ascending = !Ascending));
            builder.AddContent(4, new MarkupString("Toggle"));
            builder.CloseElement();

            builder.OpenElement(5, "ul");
            builder.AddAttribute(6, "class", "list-group");
            foreach (string name in data) {
                builder.OpenElement(7, "li");
                builder.AddAttribute(8, "class", "list-group-item");
                builder.AddContent(9, new MarkupString(name));
                builder.CloseElement();
            }
            builder.CloseElement();
        }
    }
}
```

The base class for components is ComponentBase. The content that would normally be expressed as annotated HTML elements is created by overriding the BuildRenderTree method and using the RenderTreeBuilder parameter. Creating content can be awkward because each element is created and configured using multiple code statements, and each statement must have a sequence number that the compiler uses to match up code and content. The OpenElement method starts a new element, which is configured using the AddElement and AddContent methods and then completed with the CloseElement method. All the features available in regular Razor Components are available, including events and bindings, which are set up by adding attributes to elements, just as if they were defined literally in a .razor file. The component in Listing 33-24 displays a list of sorted names, with the sort direction altered when a button element is clicked. Listing 33-25 applies the component so that it will be displayed to the user.

Listing 33-25. Applying a New Component in the Blazor.cshtml File in the Pages Folder

```
@page "/pages/blazor"

<h4 class="bg-primary text-white text-center p-2">Class Only</h4>

<component type="typeof(Advanced.Blazor.CodeOnly)" render-mode="Server" />
```

Restart ASP.NET Core and request http://localhost:5000/pages/blazor to see the content produced by the class-based Razor Component. When you click the button, the sort direction of the names in the list is changed, as shown in Figure 33-15.

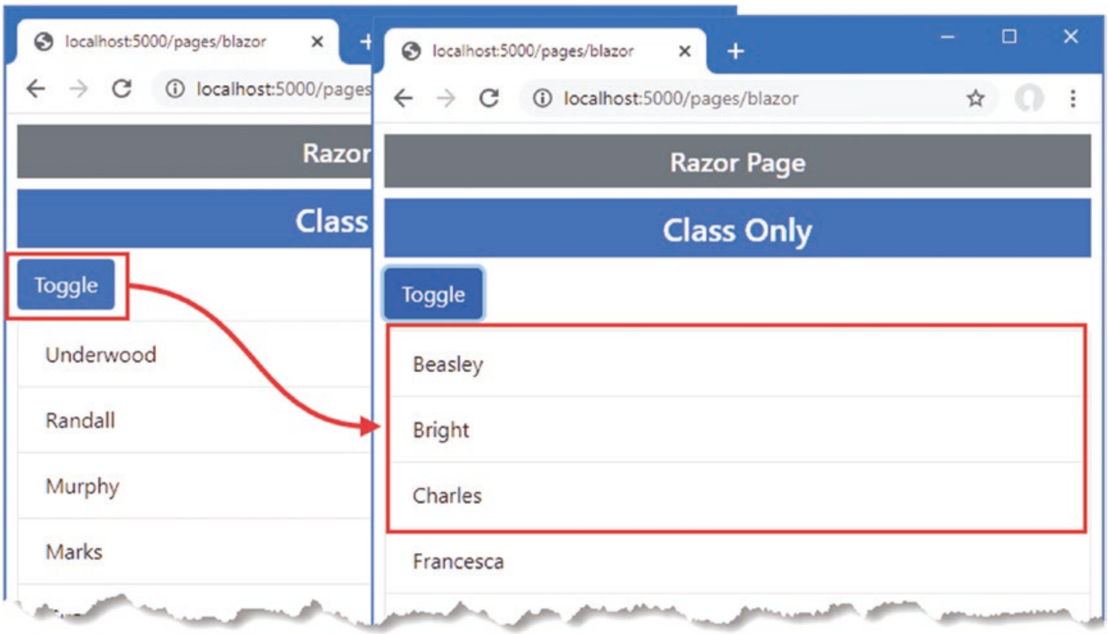

Figure 33-15. *Defining a component entirely in code*

Summary

In this chapter, I introduced Blazor Server, explained the problem it solves, and described the advantages and disadvantages it presents. I showed you how to configure an ASP.NET Core application to enable Blazor Server and showed you the basic features that are available when using Razor Components, which are the Blazor building blocks. In the next chapter, I continue to describe the features provided by Blazor.

CHAPTER 34

Using Blazor Server, Part 2

In this chapter, I continue to describe Blazor Server, focusing on the way that Razor Components can be used together to create more complex features. Table 34-1 summarizes the chapter.

Table 34-1. *Chapter Summary*

Problem	Solution	Listing
Creating complex features using Blazor	Combine components to reduce duplication	3, 4
Configuring a component	Use the `Parameter` attribute to receive a value from an attribute	5–10
Defining custom events and bindings	Use `EventCallbacks` to receive the handler for the event and follow the convention to create bindings	11–14
Displaying child content in a component	Use a `RenderFragment` named `ChildContent`	15, 16
Creating templates	Use named `RenderFragment` properties	17, 25
Distributing configuration settings widely	Use a cascading parameter	26, 27
Responding to connection errors	Use the connection element and classes	28, 29
Responding to unhandled errors	Use the error element and classes	30, 31

Preparing for This Chapter

This chapter uses the Advanced project from Chapter 33. No changes are required to prepare for this chapter.

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-3`. See Chapter 1 for how to get help if you have problems running the examples.

Open a new PowerShell command prompt, navigate to the folder that contains the `Advanced.csproj` file, and run the command shown in Listing 34-1 to drop the database.

Listing 34-1. Dropping the Database

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

Select Start Without Debugging or Run Without Debugging from the Debug menu or use the PowerShell command prompt to run the command shown in Listing 34-2.

A. Freeman, *Pro ASP.NET Core 3*, https://doi.org/10.1007/978-1-4842-5440-0_34

Listing 34-2. Running the Example Application

```
dotnet run
```

Use a browser to request `http://localhost:5000/controllers`, which will display a list of data items. Request `http://localhost:5000/pages/blazor`, and you will see the component from Chapter 33 I used to demonstrate data bindings. Figure 34-1 shows both responses.

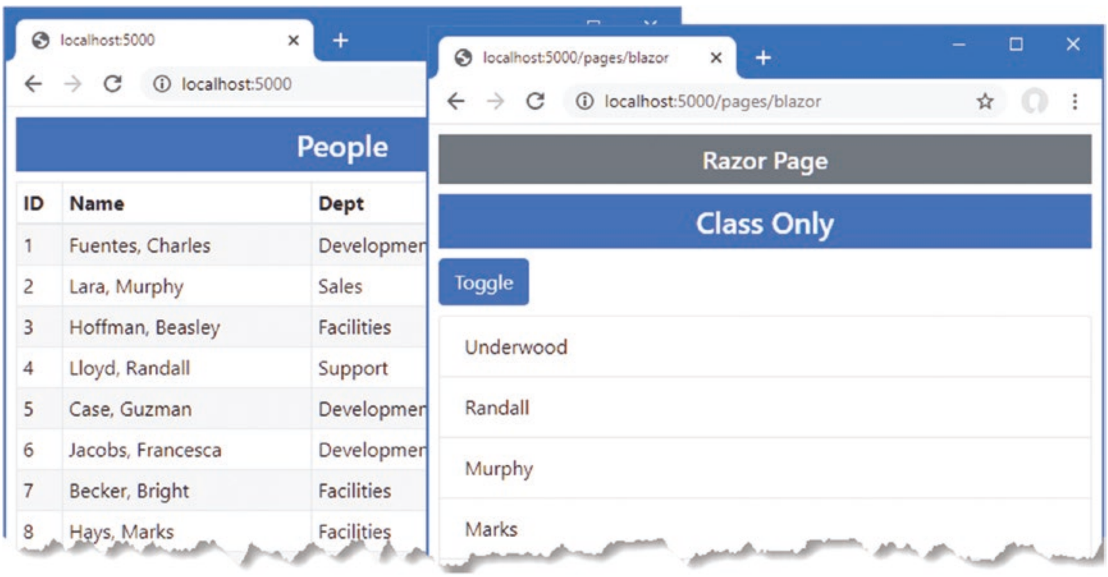

Figure 34-1. *Running the example application*

Combining Components

Blazor components can be combined to create more complex features. In the sections that follow, I show you how multiple components can be used together and how components can communicate. To get started, add a Razor Component named `SelectFilter.razor` to the `Blazor` folder with the content shown in Listing 34-3.

Listing 34-3. The Contents of the SelectFilter.razor File in the Blazor Folder

```
<div class="form-group">
    <label for="select-@Title">@Title</label>
    <select name="select-@Title" class="form-control" @bind="SelectedValue">
        <option disabled selected>Select @Title</option>
        @foreach (string val in Values) {
            <option value="@val" selected="@(val == SelectedValue)">
                @val
            </option>
        }
    </select>
</div>

@code {
```

```
    public IEnumerable<string> Values { get; set; } = Enumerable.Empty<string>();

    public string SelectedValue { get; set; }

    public string Title { get; set; } = "Placeholder";
}
```

The component renders a `select` element that will allow the user to choose a city. In Listing 34-4, I have applied the `SelectFilter` component, replacing the existing `select` element.

Listing 34-4. Applying a Component in the PeopleList.razor File in the Blazor Folder

```
<table class="table table-sm table-bordered table-striped">
    <thead><tr><th>ID</th><th>Name</th><th>Dept</th><th>Location</th></tr></thead>
    <tbody>
        @foreach (Person p in People) {
            <tr class="@GetClass(p.Location.City)">
                <td>@p.PersonId</td>
                <td>@p.Surname, @p.Firstname</td>
                <td>@p.Department.Name</td>
                <td>@p.Location.City, @p.Location.State</td>
            </tr>
        }
    </tbody>
</table>

<SelectFilter />

@code {

    [Inject]
    public DataContext Context { get; set; }

    public IEnumerable<Person> People =>
        Context.People.Include(p => p.Department).Include(p => p.Location);

    public IEnumerable<string> Cities => Context.Locations.Select(l => l.City);

    public string SelectedCity { get; set; }

    public string GetClass(string city) =>
        SelectedCity == city ? "bg-info text-white" : "";
}
```

When a component is added to the content rendered by a controller view or Razor Page, the `component` element is used, as shown in Chapter 33. When a component is added to the content rendered by another component, then the name of the component is used as an element instead. In this case, I am adding the `SelectFilter` component to the content rendered by the `PeopleList` component, which I do with a `SelectFilter` element. It is important to pay close attention to the capitalization, which must match exactly.

When combining components, the effect is that one component delegates responsibility for part of its layout to another. In this case, I have removed the `select` element that the `PeopleList` component used to present the user with a choice of cities and replaced it with the `SelectFilter` component, which will provide the same feature. The components form a parent/child relationship; the `PeopleList` component is the parent, and the `SelectFilter` component is the child.

Additional work is required before everything is properly integrated, but you can see that adding the `SelectFilter` element displays the `SelectFilter` component by restarting ASP.NET Core and requesting `http://localhost:5000/controllers`, which produces the response shown in Figure 34-2.

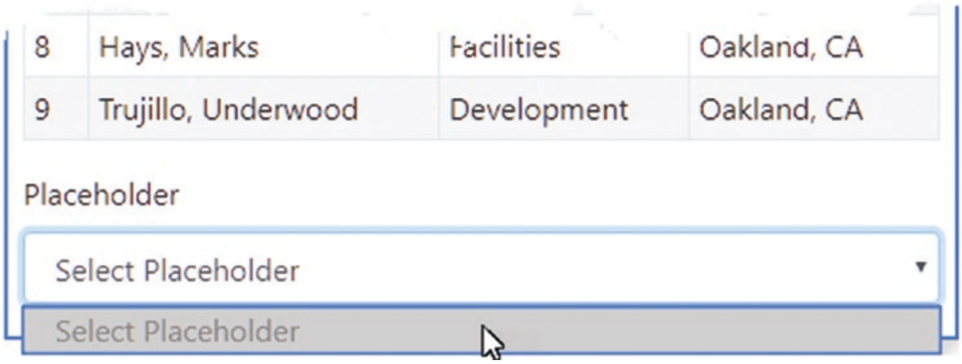

Figure 34-2. *Adding one component to the content rendered by another*

Configuring Components with Attributes

My goal with the `SelectList` component is to create a general-purpose feature that I can use throughout the application, configuring the values it displays each time it is used. Razor Components are configured using attributes added to the HTML element that applies them. The values assigned to the HTML element attributes are assigned to the component's C# properties. The `Parameter` attribute is applied to the C# properties that a component allows to be configured, as shown in Listing 34-5.

Listing 34-5. Declaring Configurable Properties in the SelectFilter.razor File in the Blazor Folder

```
<div class="form-group">
    <label for="select-@Title">@Title</label>
    <select name="select-@Title" class="form-control" @bind="SelectedValue">
        <option disabled selected>Select @Title</option>
        @foreach (string val in Values) {
            <option value="@val" selected="@(val == SelectedValue)">
                @val
            </option>
        }
    </select>
</div>

@code {

    [Parameter]
    public IEnumerable<string> Values { get; set; } = Enumerable.Empty<string>();

    public string SelectedValue { get; set; }

    [Parameter]
    public string Title { get; set; } = "Placeholder";
}
```

Components can be selective about the properties they allow to be configured. In this case, the `Parameter` attribute has been applied to two of the properties defined by the `SelectFilter` component. In Listing 34-6, I have modified the element the `PeopleList` component uses to apply the `SelectFilter` component to add configuration attributes.

Listing 34-6. Configuring a Component in the PeopleList.razor File in the Blazor Folder

```
<table class="table table-sm table-bordered table-striped">
    <thead><tr><th>ID</th><th>Name</th><th>Dept</th><th>Location</th></tr></thead>
    <tbody>
        @foreach (Person p in People) {
```

```
            <tr class="@GetClass(p.Location.City)">
                <td>@p.PersonId</td>
                <td>@p.Surname, @p.Firstname</td>
                <td>@p.Department.Name</td>
                <td>@p.Location.City, @p.Location.State</td>
            </tr>
        }
    </tbody>
</table>

<SelectFilter values="@Cities" title="City" />

@code {

    [Inject]
    public DataContext Context { get; set; }

    public IEnumerable<Person> People =>
        Context.People.Include(p => p.Department).Include(p => p.Location);

    public IEnumerable<string> Cities => Context.Locations.Select(l => l.City);

    public string SelectedCity { get; set; }

    public string GetClass(string city) =>
        SelectedCity == city ? "bg-info text-white" : "";
}
```

For each property that should be configured, an attribute of the same name is added to the parent's HTML element. The attribute values can be fixed values, such as the `City` string assigned to the `title` attribute, or Razor expressions, such as `@Cities`, which assigns the sequence of objects from the `Cities` property to the `values` attribute.

Setting and Receiving Bulk Configuration Settings

Defining individual properties to receive values can be error-prone if there are many configuration settings, especially if those values are being received by a component so they can be passed on, either to a child component or to a regular HTML element. In these situations, a single property can be designated to receive any attribute values that have not been matched by other properties, which can then be applied as a set, as shown in Listing 34-7.

Listing 34-7. Receiving Bulk Attributes in the SelectFilter.razor File in the Blazor Folder

```
<div class="form-group">
    <label for="select-@Title">@Title</label>
    <select name="select-@Title" class="form-control"
            @bind="SelectedValue" @attributes="Attrs">
        <option disabled selected>Select @Title</option>
        @foreach (string val in Values) {
            <option value="@val" selected="@(val == SelectedValue)">
                @val
            </option>
        }
    </select>
</div>

@code {
```

```
    [Parameter]
    public IEnumerable<string> Values { get; set; } = Enumerable.Empty<string>();

    public string SelectedValue { get; set; }

    [Parameter]
    public string Title { get; set; } = "Placeholder";

    [Parameter(CaptureUnmatchedValues = true)]
    public Dictionary<string, object> Attrs { get; set; }
}
```

Setting the `Parameter` attribute's `CaptureUnmatchedValues` argument to `true` identifies a property as the catchall for attributes that are not otherwise matched. The type of the property must be `Dictionary<string, object>`, which allows the attribute names and values to be represented.

Properties whose type is `Dictionary<string, object>` can be applied to elements using the `@attribute` expression, like this:

```
...
<select name="select-@Title" class="form-control" @bind="SelectedValue"
    @attributes="Attrs">
...
```

This is known as *attribute splatting*, and it allows a set of attributes to be applied in one go. The effect of the changes in Listing 34-7 means that the `SelectFilter` component will receive the `Values` and `Title` attribute values and that any other attributes will be assigned to the `Attrs` property and passed on to the `select` element. Listing 34-8 adds some attributes to demonstrate the effect.

Listing 34-8. Adding Element Attributes in the PeopleList.razor File in the Blazor Folder

```
<table class="table table-sm table-bordered table-striped">
    <thead><tr><th>ID</th><th>Name</th><th>Dept</th><th>Location</th></tr></thead>
    <tbody>
        @foreach (Person p in People) {
            <tr class="@GetClass(p.Location.City)">
                <td>@p.PersonId</td>
                <td>@p.Surname, @p.Firstname</td>
                <td>@p.Department.Name</td>
                <td>@p.Location.City, @p.Location.State</td>
            </tr>
        }
    </tbody>
</table>

<SelectFilter values="@Cities" title="City" autofocus="true" name="city"
    required="true" />

@code {

    // ...statements omitted for brevity...
}
```

Restart ASP.NET Core and navigate to `http://localhost:5000/controllers`. The attributes passed on to the `select` element do not affect appearance, but if you right-click the `select` element and select Inspect from the popup menu, you will see the attributes added to the `SelectFilter` element in the `PeopleList` component have been added to the element rendered by the `SelectFilter` component, like this:

```
...
<select class="form-control" autofocus="true" name="city" required="true">
...
```

Configuring a Component in a Controller View or Razor Page

Attributes are also used to configure components when they are applied using the `component` element. In Listing 34-9, I have added properties to the `PeopleList` component that specify how many items from the database should be displayed and a string value that will be passed on to the `SelectFilter` component.

Listing 34-9. Adding Configuration Properties in the PeopleList.razor File in the Blazor Folder

```
<table class="table table-sm table-bordered table-striped">
    <thead><tr><th>ID</th><th>Name</th><th>Dept</th><th>Location</th></tr></thead>
    <tbody>
        @foreach (Person p in People) {
            <tr class="@GetClass(p.Location.City)">
                <td>@p.PersonId</td>
                <td>@p.Surname, @p.Firstname</td>
                <td>@p.Department.Name</td>
                <td>@p.Location.City, @p.Location.State</td>
            </tr>
        }
    </tbody>
</table>

<SelectFilter values="@Cities" title="@SelectTitle" />

@code {

    [Inject]
    public DataContext Context { get; set; }

    public IEnumerable<Person> People => Context.People.Include(p => p.Department)
            .Include(p => p.Location).Take(ItemCount);

    public IEnumerable<string> Cities => Context.Locations.Select(l => l.City);

    public string SelectedCity { get; set; }

    public string GetClass(string city) =>
        SelectedCity == city ? "bg-info text-white" : "";

    [Parameter]
    public int ItemCount { get; set; } = 4;

    [Parameter]
    public string SelectTitle { get; set; }
}
```

Values for the C# properties are provided by adding attributes whose name begins with `param-`, followed by the property name, to the `component` element, as shown in Listing 34-10.

Listing 34-10. Adding Configuration Attributes in the Index.cshtml File in the Views/Home Folder

```
@model PeopleListViewModel

<h4 class="bg-primary text-white text-center p-2">People</h4>

<component type="typeof(Advanced.Blazor.PeopleList)" render-mode="Server"
           param-itemcount="5" param-selecttitle="@("Location")" />
```

The param-itemcount attribute provides a value for the ItemCount property, and the param-selecttitle attribute provides a value for the SelectTitle property.

When using the component element, attributes values that can be parsed into numeric or bool values are handled as literal values and not Razor expressions, which is why I am able to specify the value for the ItemCount property as 4. Other values are assumed to be Razor expressions and not literal values, even though they are not prefixed with @. This oddity means that since I want to specify the value for the SelectTitle property as a literal string, I need a Razor expression, like this:

```
...
<component type="typeof(Advanced.Blazor.PeopleList)" render-mode="Server"
           param-itemcount="5" param-selecttitle="@("Location")" />
...
```

To see the effect of the configuration attributes, restart ASP.NET Core and request http://localhost:5000/controllers, which will produce the response shown in Figure 34-3.

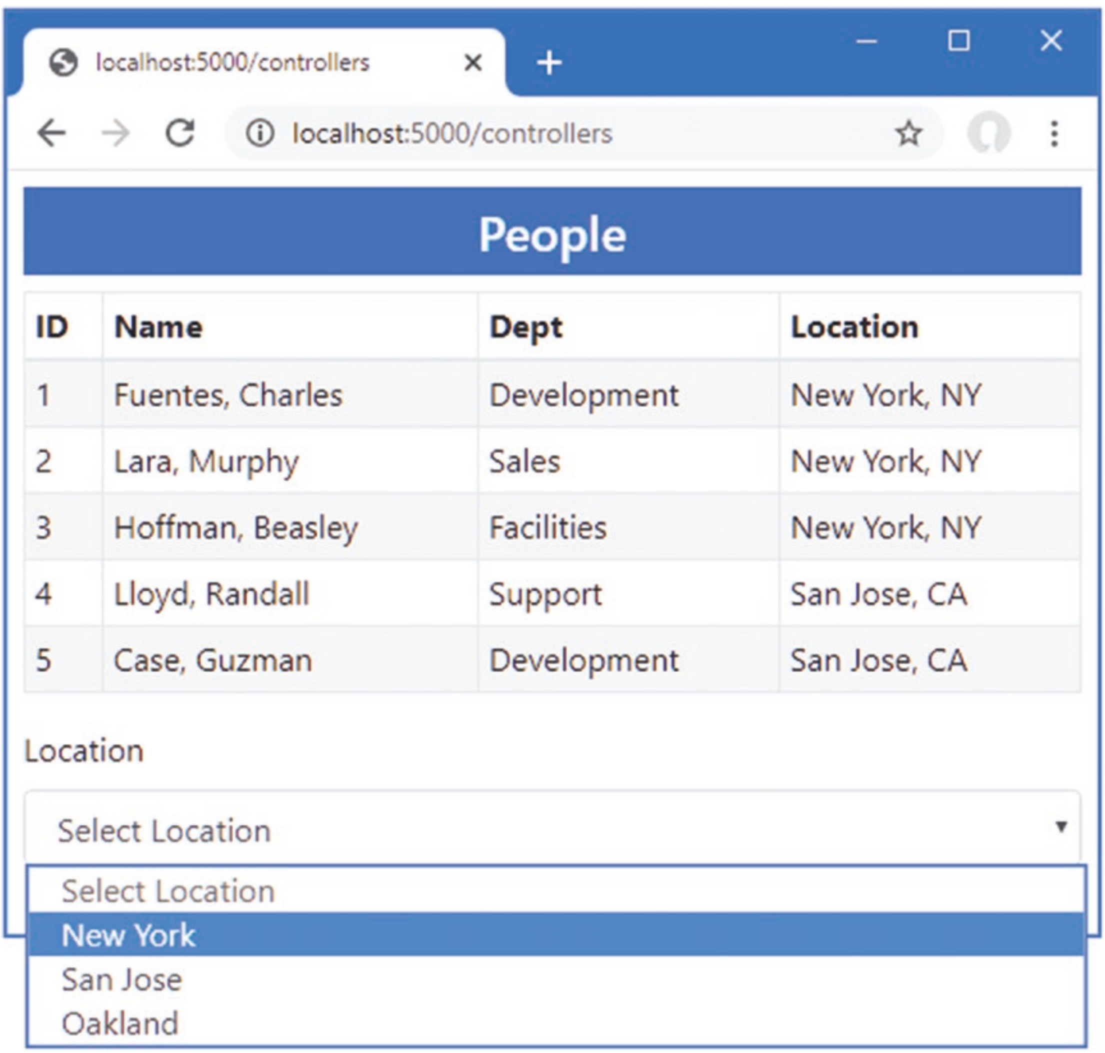

Figure 34-3. *Configuring components with attributes*

Creating Custom Events and Bindings

The SelectFilter component receives its data values from its parent component, but it has no way to indicate when the user makes a selection. For this, I need to create a custom event for which the parent component can register a handler method, just as it would for events from regular HTML elements. Listing 34-11 adds a custom event to the SelectFilter component.

Listing 34-11. Creating an Event in the SelectFilter.razor File in the Blazor Folder

```
<div class="form-group">
    <label for="select-@Title">@Title</label>
    <select name="select-@Title" class="form-control"
            @onchange="HandleSelect" value="@SelectedValue">
        <option disabled selected>Select @Title</option>
        @foreach (string val in Values) {
            <option value="@val" selected="@(val == SelectedValue)">
                @val
            </option>
        }
    </select>
</div>

@code {

    [Parameter]
    public IEnumerable<string> Values { get; set; } = Enumerable.Empty<string>();

    public string SelectedValue { get; set; }

    [Parameter]
    public string Title { get; set; } = "Placeholder";

    [Parameter(CaptureUnmatchedValues = true)]
    public Dictionary<string, object> Attrs { get; set; }

    [Parameter]
    public EventCallback<string> CustomEvent { get; set; }

    public async Task HandleSelect(ChangeEventArgs e) {
        SelectedValue = e.Value as string;
        await CustomEvent.InvokeAsync(SelectedValue);
    }
}
```

The custom event is defined by adding a property whose type is EventCallback<T>. The generic type argument is the type that will be received by the parent's event handler and is string in this case. I have changed the select element so the @onchange attribute registers the HandleSelect method when the select element triggers its onchange event.

The HandleSelect method updates the SelectedValue property and triggers the custom event by invoking the EventCallback<T>.InvokeAsync method, like this:

```
...
await CustomEvent.InvokeAsync(SelectedValue);
...
```

The argument to the InvokeAsync method is used to trigger the event using the value received from the ChangeEventArgs object that was received from the select element. Listing 34-12 changes the PeopleList component so that it receives the custom event emitted by the SelectList component.

Listing 34-12. Handling an Event in the PeopleList.razor File in the Blazor Folder

```
<table class="table table-sm table-bordered table-striped">
    <thead><tr><th>ID</th><th>Name</th><th>Dept</th><th>Location</th></tr></thead>
    <tbody>
        @foreach (Person p in People) {
            <tr class="@GetClass(p.Location.City)">
                <td>@p.PersonId</td>
                <td>@p.Surname, @p.Firstname</td>
                <td>@p.Department.Name</td>
                <td>@p.Location.City, @p.Location.State</td>
            </tr>
        }
    </tbody>
</table>

<SelectFilter values="@Cities" title="@SelectTitle" CustomEvent="@HandleCustom" />

@code {

    [Inject]
    public DataContext Context { get; set; }

    public IEnumerable<Person> People => Context.People.Include(p => p.Department)
            .Include(p => p.Location).Take(ItemCount);

    public IEnumerable<string> Cities => Context.Locations.Select(l => l.City);

    public string SelectedCity { get; set; }

    public string GetClass(string city) =>
        SelectedCity as string == city ? "bg-info text-white" : "";

    [Parameter]
    public int ItemCount { get; set; } = 4;

    [Parameter]
    public string SelectTitle { get; set; }

    public void HandleCustom(string newValue) {
        SelectedCity = newValue;
    }
}
```

To set up the event handler, an attribute is added to the element that applies the child component using the name of its `EventCallback<T>` property. The value of the attribute is a Razor expression that selects a method that receives a parameter of type `T`.

Restart ASP.NET Core, request `http://localhost:5000/controllers`, and select a value from the list of cities. The custom event completes the relationship between the parent and child components. The parent configures the child through its attributes to specify the title and the list of data values that will be presented to the user. The child component uses a custom event to tell the parent when the user selects a value, allowing the parent to highlight the corresponding rows in its HTML table, as shown in Figure 34-4.

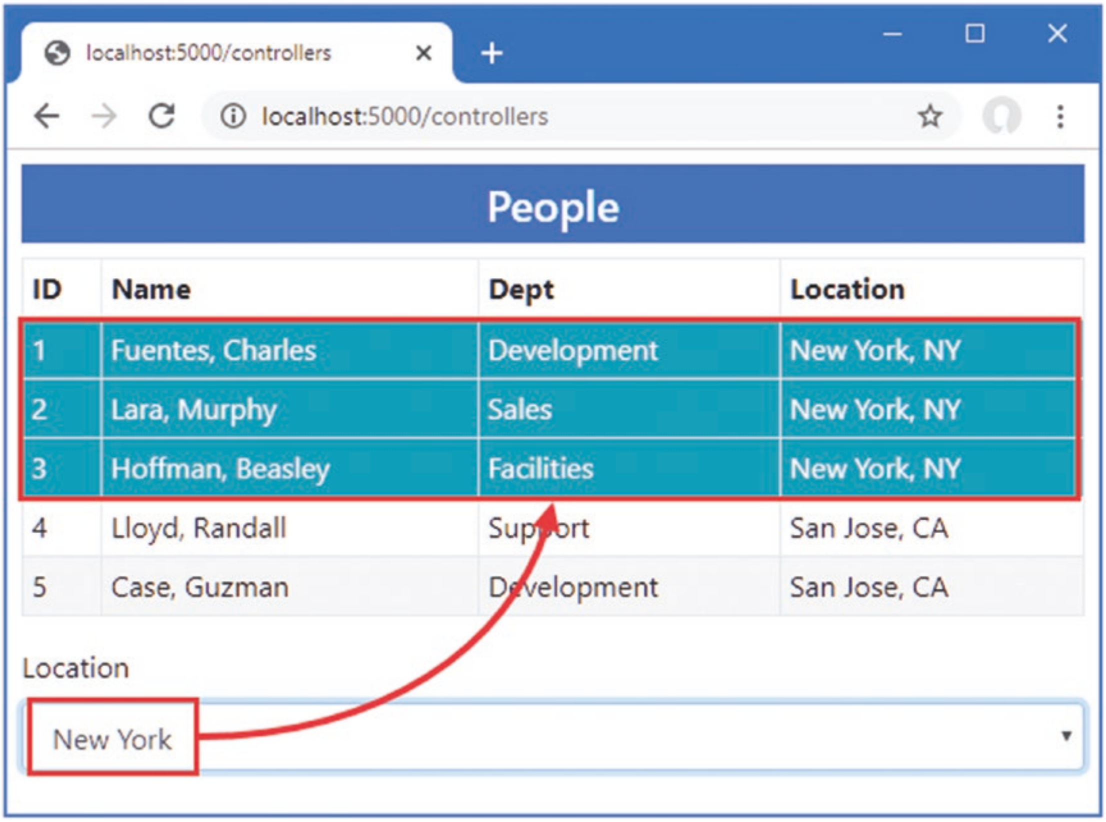

Figure 34-4. *Using a custom event*

Creating a Custom Binding

A parent component can create a binding on a child component if it defines a pair of properties, one of which is assigned a data value and the other of which is a custom event. The names of the property are important: the name of the event property must be the same as the data property plus the word Changed. Listing 34-13 updates the `SelectFilter` component so it presents the properties required for the binding.

Listing 34-13. Preparing for Custom Binding in the SelectFilter.razor File in the Blazor Folder

```
<div class="form-group">
    <label for="select-@Title">@Title</label>
    <select name="select-@Title" class="form-control"
            @onchange="HandleSelect" value="@SelectedValue">
        <option disabled selected>Select @Title</option>
        @foreach (string val in Values) {
            <option value="@val" selected="@(val == SelectedValue)">
                @val
            </option>
        }
    </select>
</div>

@code {

    [Parameter]
    public IEnumerable<string> Values { get; set; } = Enumerable.Empty<string>();
```

```
    [Parameter]
    public string SelectedValue { get; set; }

    [Parameter]
    public string Title { get; set; } = "Placeholder";

    [Parameter(CaptureUnmatchedValues = true)]
    public Dictionary<string, object> Attrs { get; set; }

    [Parameter]
    public EventCallback<string> SelectedValueChanged { get; set; }

    public async Task HandleSelect(ChangeEventArgs e) {
        SelectedValue = e.Value as string;
        await SelectedValueChanged.InvokeAsync(SelectedValue);
    }
}
```

Notice that the Parameter attribute must be applied to both the SelectedValue and SelectedValueChanged properties. If either attribute is omitted, the data binding won't work as expected.

The parent component binds to the child with the @bind-<name> attribute, where <name> corresponds to the property defined by the child component. In this example, the name of the child component's property is SelectedValue, and the parent can create a binding using @bind-SelectedValue, as shown in Listing 34-14.

Listing 34-14. Using a Custom Binding in the PeopleList.razor File in the Blazor Folder

```
<table class="table table-sm table-bordered table-striped">
    <thead><tr><th>ID</th><th>Name</th><th>Dept</th><th>Location</th></tr></thead>
    <tbody>
        @foreach (Person p in People) {
            <tr class="@GetClass(p.Location.City)">
                <td>@p.PersonId</td>
                <td>@p.Surname, @p.Firstname</td>
                <td>@p.Department.Name</td>
                <td>@p.Location.City, @p.Location.State</td>
            </tr>
        }
    </tbody>
</table>

<SelectFilter values="@Cities" title="@SelectTitle"
    @bind-SelectedValue="SelectedCity" />

<button class="btn btn-primary"
    @onclick="@(() => SelectedCity = "San Jose")">
        Change
</button>

@code {

    [Inject]
    public DataContext Context { get; set; }

    public IEnumerable<Person> People => Context.People.Include(p => p.Department)
            .Include(p => p.Location).Take(ItemCount);

    public IEnumerable<string> Cities => Context.Locations.Select(l => l.City);
```

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

    public string GetClass(string city) =>
        SelectedCity as string == city ? "bg-info text-white" : "";

    [Parameter]
    public int ItemCount { get; set; } = 4;

    [Parameter]
    public string SelectTitle { get; set; }

    //public void HandleCustom(string newValue) {
    //    SelectedCity = newValue;
    //}
}
```

Restart ASP.NET Core, request `http://localhost:5000/controllers`, and select New York from the list of cities. The custom binding will cause the value chosen in the `select` element to be reflected by the highlighting in the table. Click the Change button to test the binding in the other direction, and you will see the highlighted city change, as shown in Figure 34-5.

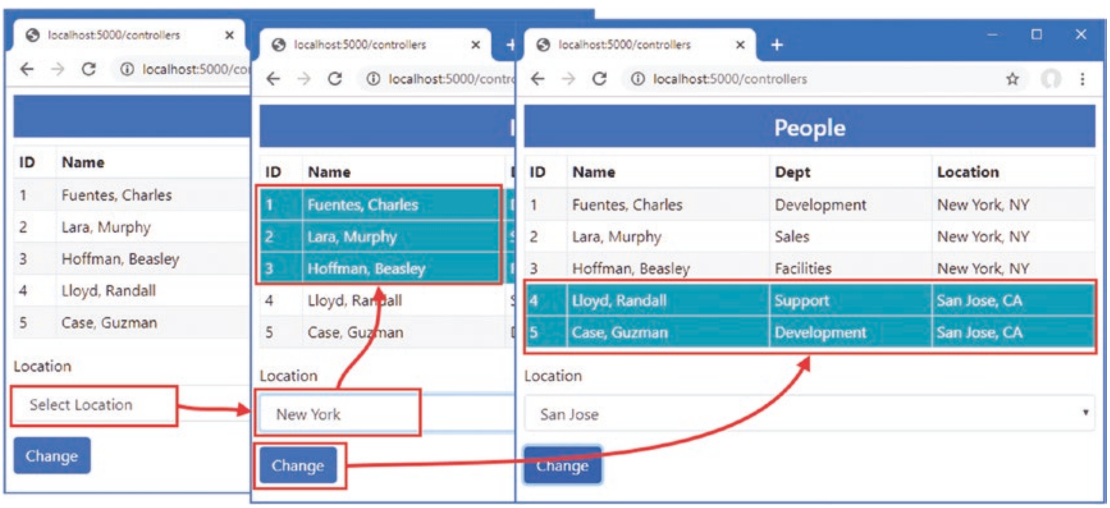

Figure 34-5. *Using a custom binding*

Displaying Child Content in a Component

Components that display child content act as wrappers around elements provided by their parents. To see how child content is managed, add a Razor Component named `ThemeWrapper.razor` to the `Blazor` folder with the content shown in Listing 34-15.

Listing 34-15. The Contents of the ThemeWrapper.razor File in the Blazor Folder

```
<div class="p-2 bg-@Theme border text-white">
    <h5 class="text-center">@Title</h5>
    @ChildContent
</div>

@code {
    [Parameter]
    public string Theme { get; set; }
```

```
    [Parameter]
    public string Title { get; set; }

    [Parameter]
    public RenderFragment ChildContent { get; set; }
}
```

To receive child content, a component defines a property named `ChildContent` whose type is `RenderFragment` and that has been decorated with the `Parameter` attribute. The `@ChildContent` expression includes the child content in the component's HTML output. The component in the listing wraps its child content in a `div` element that is styled using a Bootstrap theme color and that displays a title. The name of the theme color and the text of the title are also received as parameters.

RESTRICTING ELEMENT REUSE

When updating the content presented to the user, Blazor will reuse elements if it can because creating new elements is a relatively expensive operation. This is particularly true when displaying elements for a sequence of values, such as with `@for` or `@foreach` expressions. If the sequence changes, Blazor will reuse the elements it created for the old data values to display the new data.

This can cause problems if changes have been made to the elements outside of the control of Blazor, such as with custom JavaScript code. Blazor isn't aware of the changes, which will persist when the elements are reused. Although this is a rare situation, you can restrict the reuse of elements by using an `@key` attribute and providing an expression that associates the element with one of the data values in the sequence, like this:

```
...
@foreach (Person p in People) {
    <tr @key="p.PersonId" class="@GetClass(p.Location.City)">

        <td>@p.PersonId</td>
        <td>@p.Surname, @p.Firstname</td>
        <td>@p.Department.Name</td>
        <td>@p.Location.City, @p.Location.State</td>
    </tr>
}
...
```

Blazor will reuse an element only if there is a data item that has the same key. For other values, new elements will be created.

Child content is defined by adding HTML elements between the start and end tags when applying the component, as shown in Listing 34-16.

Listing 34-16. Defining Child Content in the PeopleList.razor File in the Blazor Folder

```
<table class="table table-sm table-bordered table-striped">
    <thead><tr><th>ID</th><th>Name</th><th>Dept</th><th>Location</th></tr></thead>
    <tbody>
        @foreach (Person p in People) {
            <tr class="@GetClass(p.Location.City)">
                <td>@p.PersonId</td>
                <td>@p.Surname, @p.Firstname</td>
                <td>@p.Department.Name</td>
                <td>@p.Location.City, @p.Location.State</td>
            </tr>
        }
    </tbody>
</table>
```

```
<ThemeWrapper Theme="info" Title="Location Selector">
    <SelectFilter values="@Cities" title="@SelectTitle"
        @bind-SelectedValue="SelectedCity" />
    <button class="btn btn-primary"
        @onclick="@(() => SelectedCity = "San Jose")">
            Change
     </button>
</ThemeWrapper>

@code {

    // ...statements omitted for brevity...
}
```

No additional attributes are required to configure the child content, which is processed and assigned to the ChildContent property automatically. To see how the ThemeWrapper component presents its child content, restart ASP.NET Core and request http://localhost:5000/controllers. You will see the configuration attributes that selected the theme and the title text used to produce the response shown in Figure 34-6.

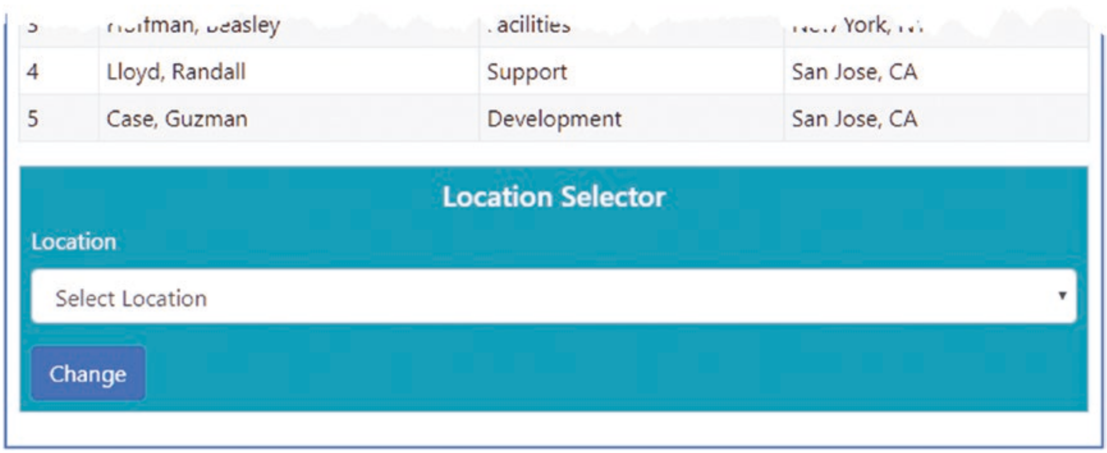

Figure 34-6. *Using child content*

Creating Template Components

Template components bring more structure to the presentation of child content, allowing multiple sections of content to be displayed. Template components are a good way of consolidating features that are used throughout an application to prevent the duplication of code and content.

To see how this works, add a Razor Component named TableTemplate.razor to the Blazor folder with the content shown in Listing 34-17.

Listing 34-17. The Contents of the TableTemplate.razor File in the Blazor Folder

```
<table class="table table-sm table-bordered table-striped">
    @if (Header != null) {
        <thead>@Header</thead>
    }
    <tbody>@Body</tbody>
</table>

@code {
    [Parameter]
    public RenderFragment Header { get; set; }
```

```
    [Parameter]
    public RenderFragment Body { get; set; }
}
```

The component defines a `RenderFragment` property for each region of child content it supports. The `TableTemplate` component defines two `RenderFragment` properties, named `Header` and `Body`, which represent the content sections of a table. Each region of child content is rendered using a Razor expression, `@Header` and `@Body`, and you can check to see whether content has been provided for a specific section by checking to see whether the property value is `null`, which this component does for the `Header` section.

When using a template component, the content for each region is enclosed in an HTML element whose tag matches the name of the corresponding `RenderFragment` property, as shown in Listing 34-18.

Listing 34-18. Applying a Template Component in the PeopleList.razor File in the Blazor Folder

```
<TableTemplate>
    <Header>
        <tr><th>ID</th><th>Name</th><th>Dept</th><th>Location</th></tr>
    </Header>
    <Body>
        @foreach (Person p in People) {
            <tr class="@GetClass(p.Location.City)">
                <td>@p.PersonId</td>
                <td>@p.Surname, @p.Firstname</td>
                <td>@p.Department.Name</td>
                <td>@p.Location.City, @p.Location.State</td>
            </tr>
        }
    </Body>
</TableTemplate>

<ThemeWrapper Theme="info" Title="Location Selector">
    <SelectFilter values="@Cities" title="@SelectTitle"
        @bind-SelectedValue="SelectedCity" />
    <button class="btn btn-primary"
        @onclick="@(() => SelectedCity = "San Jose")">
            Change
     </button>
</ThemeWrapper>

@code {

    // ...statements omitted for brevity...
}
```

The child content is structured into sections that correspond to the template component's properties, `Header` and `Body`, which leaves the `TableTemplate` component responsible for the table structure and the `PeopleList` component responsible for providing the detail. Restart ASP.NET Core and request `http://localhost:5000/controllers`, and you will see the output produced by the template component, as shown in Figure 34-7.

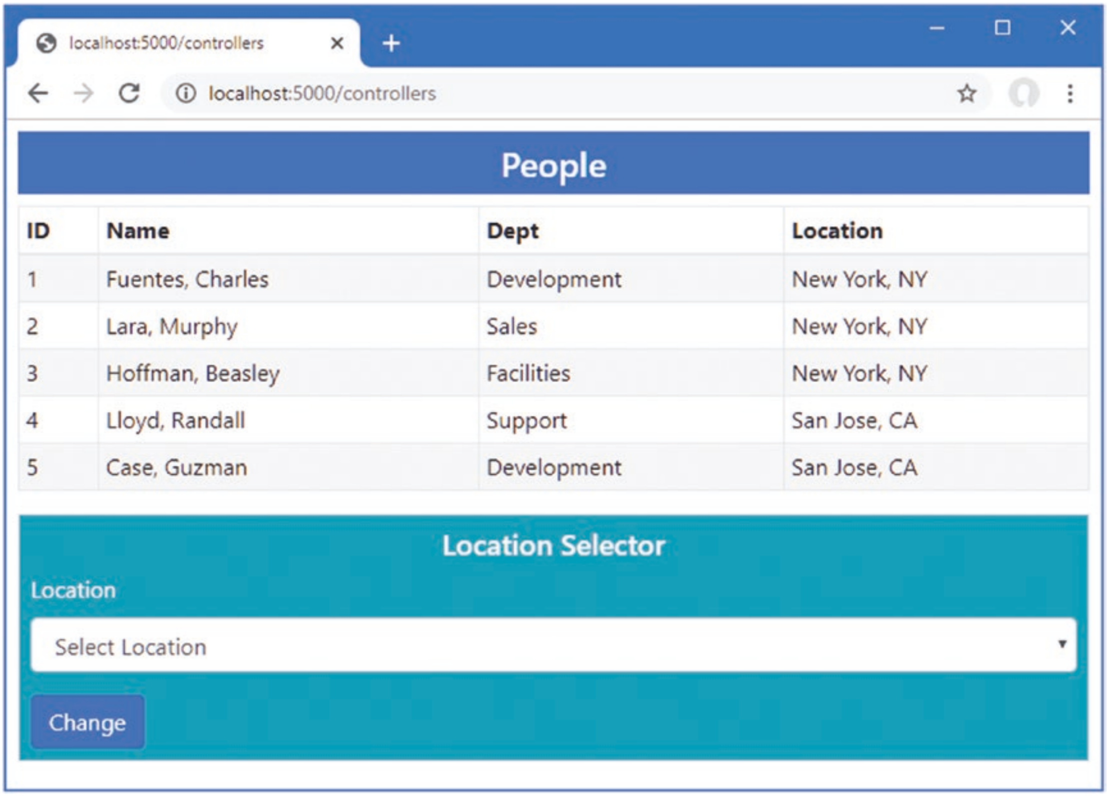

Figure 34-7. *Using a template component*

Using Generic Type Parameters in Template Components

The template component I created in the previous section is useful, in the sense that it provides a consistent representation of a table that I can use throughout the example application. But it is also limited because it relies on the parent component to take responsibility for generating the rows for the table body. The template component doesn't have any insight into the content it presents, which means it cannot do anything with that content other than display it.

Template components can be made data-aware with the use of a generic type parameter, which allows the parent component to provide a sequence of data objects and a template for presenting them. The template component becomes responsible for generating the content for each data object and, consequently, can provide more useful functionality. As a demonstration, I am going to add support to the template component for selecting how many table rows are displayed and for selecting table rows. The first step is to add a generic type parameter to the component and use it to render the content for the table body, as shown in Listing 34-19.

Listing 34-19. Adding a Generic Type Parameter in the TableTemplate.razor File in the Blazor Folder

```
@typeparam RowType

<table class="table table-sm table-bordered table-striped">
    @if (Header != null) {
        <thead>@Header</thead>
    }
    <tbody>
        @foreach (RowType item in RowData) {
            <tr>@RowTemplate(item)</tr>
        }
    </tbody>
</table>
```

```
@code {
    [Parameter]
    public RenderFragment Header { get; set; }

    [Parameter]
    public RenderFragment<RowType> RowTemplate{ get; set; }

    [Parameter]
    public IEnumerable<RowType> RowData { get; set; }
}
```

The generic type parameter is specified using the `@typeparam` attribute, and, in this case, I have given the parameter the name `RowType` because it will refer to the data type for which the component will generate table rows.

The data the component will process is received by adding a property whose type is a sequence of objects of the generic type. I have named the property `RowData`, and its type is `IEnumerable<RowType>`. The content the component will display for each object is received using a `RenderFragment<T>` property. I have named this property `RowTemplate`, and its type is `RenderFragment<RowType>`, reflecting the name I selected for the generic type parameter.

When a component receives a content section through a `RenderFragment<T>` property, it can render it for a single object by invoking the section as a method and using the object as the argument, like this:

```
...
@foreach (RowType item in RowData) {
    <tr>@RowTemplate(item)</tr>
}
...
```

This fragment of code enumerates the `RowType` objects in the `RowData` sequence and renders the content section received through the `RowTemplate` property for each of them.

Using a Generic Template Component

I have simplified the `PeopleList` component so it only uses the template component to produce a table of `Person` objects, and I have removed earlier features, as shown in Listing 34-20.

Listing 34-20. Using a Generic Template Component in the PeopleList.razor File in the Blazor Folder

```
<TableTemplate RowType="Person" RowData="People">
    <Header>
        <tr><th>ID</th><th>Name</th><th>Dept</th><th>Location</th></tr>
    </Header>
    <RowTemplate Context="p">
        <td>@p.PersonId</td>
        <td>@p.Surname, @p.Firstname</td>
        <td>@p.Department.Name</td>
        <td>@p.Location.City, @p.Location.State</td>
    </RowTemplate>
</TableTemplate>

@code {

    [Inject]
    public DataContext Context { get; set; }

    public IEnumerable<Person> People => Context.People
            .Include(p => p.Department)
            .Include(p => p.Location);
}
```

The RowType attribute is used to specify the value for the generic type argument. The RowData attribute specifies the data the template component will process.

The RowTemplate element denotes the elements that will be produced for each data object. When defining a content section for a RenderFragment<T> property, the Context attribute is used to assign a name to the current object being processed. In this case, the Context attribute is used to assign the name p to the current object, which is then referred to in the Razor expressions used to populate the content section's elements.

The overall effect is that the template component is configured to display Person objects. The component will generate a table row for each Person, which will contain td elements whose content is set using the current Person object's properties.

Since I removed properties that were decorated with the Parameter attribute in Listing 34-20, I need to remove the corresponding attributes from the element that applies the PepleList component, as shown in Listing 34-21.

Listing 34-21. Removing Attributes in the Index.cshtml File in the Views/Home Folder

```
@model PeopleListViewModel

<h4 class="bg-primary text-white text-center p-2">People</h4>

<component type="typeof(Advanced.Blazor.PeopleList)" render-mode="Server" />
```

To see the generic template component, restart ASP.NET Core and request http://localhost:5000/controllers. The data and content sections provided by the PeopleList component have been used by the TableTemplate component to produce the table shown in Figure 34-8.

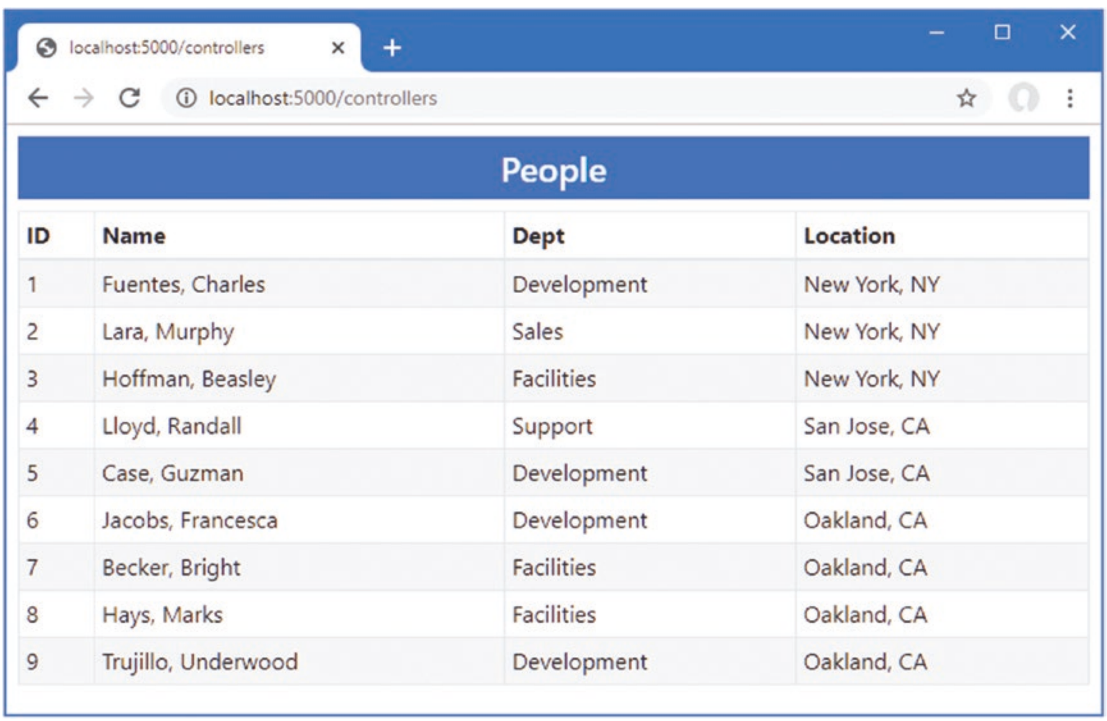

Figure 34-8. *Using a generic template component*

Adding Features to the Generic Template Component

This may feel like a step backward, but, as you will see, giving the template component insight into the data it handles sets the foundation for adding features, as shown in Listing 34-22.

Listing 34-22. Adding a Feature in the TableTemplate.razor File in the Blazor Folder

```
@typeparam RowType

<div class="container-fluid">
    <div class="row">
        <div class="col">
            <SelectFilter Title="@("Sort")" Values="@SortDirectionChoices"
                @bind-SelectedValue="SortDirectionSelection" />
        </div>
        <div class="col">
            <SelectFilter Title="@("Highlight")" Values="@HighlightChoices()"
                @bind-SelectedValue="HighlightSelection" />
        </div>
    </div>
</div>

<table class="table table-sm table-bordered table-striped">
    @if (Header != null) {
        <thead>@Header</thead>
    }
    <tbody>
        @foreach (RowType item in SortedData()) {
            <tr class="@IsHighlighted(item)">@RowTemplate(item)</tr>
        }
    </tbody>
</table>

@code {
    [Parameter]
    public RenderFragment Header { get; set; }

    [Parameter]
    public RenderFragment<RowType> RowTemplate{ get; set; }

    [Parameter]
    public IEnumerable<RowType> RowData { get; set; }

    [Parameter]
    public Func<RowType, string> Highlight { get; set; }

    public IEnumerable<string> HighlightChoices() =>
        RowData.Select(item => Highlight(item)).Distinct();

    public string HighlightSelection { get; set; }

    public string IsHighlighted(RowType item) =>
        Highlight(item) == HighlightSelection ? "bg-dark text-white": "";

    [Parameter]
    public Func<RowType, string> SortDirection { get; set; }

    public string[] SortDirectionChoices =
        new string[] { "Ascending", "Descending" };

    public string SortDirectionSelection{ get; set; } = "Ascending";
```

```
    public IEnumerable<RowType> SortedData() =>
        SortDirectionSelection == "Ascending"
            ? RowData.OrderBy(SortDirection)
            : RowData.OrderByDescending(SortDirection);
}
```

The changes present the user with two `select` elements that are presented using the `SelectFilter` component created earlier in the chapter. These new elements allow the user to sort the data in ascending and descending order and to select a value used to highlight rows in the table. The parent component provides additional parameters that give the template component functions that select the properties used for sorting and highlighting, as shown in Listing 34-23.

Listing 34-23. Configuring Template Component Features in the PeopleList.razor File in the Blazor Folder

```
<TableTemplate RowType="Person" RowData="People"
        Highlight="@(p => p.Location.City)" SortDirection="@(p => p.Surname)">
    <Header>
        <tr><th>ID</th><th>Name</th><th>Dept</th><th>Location</th></tr>
    </Header>
    <RowTemplate Context="p">
        <td>@p.PersonId</td>
        <td>@p.Surname, @p.Firstname</td>
        <td>@p.Department.Name</td>
        <td>@p.Location.City, @p.Location.State</td>
    </RowTemplate>
</TableTemplate>

@code {

    [Inject]
    public DataContext Context { get; set; }

    public IEnumerable<Person> People => Context.People
            .Include(p => p.Department)
            .Include(p => p.Location);
}
```

The `Highlight` attribute provides the template component with a function that selects the property used for highlighting table rows, and the `SortDirection` attribute provides a function that selects a property used for sorting. To see the effect, restart ASP.NET Core and request `http://localhost:5000/controllers`. The response will contain the new `select` elements, which can be used to change the sort order or select a city for filtering, as shown in Figure 34-9.

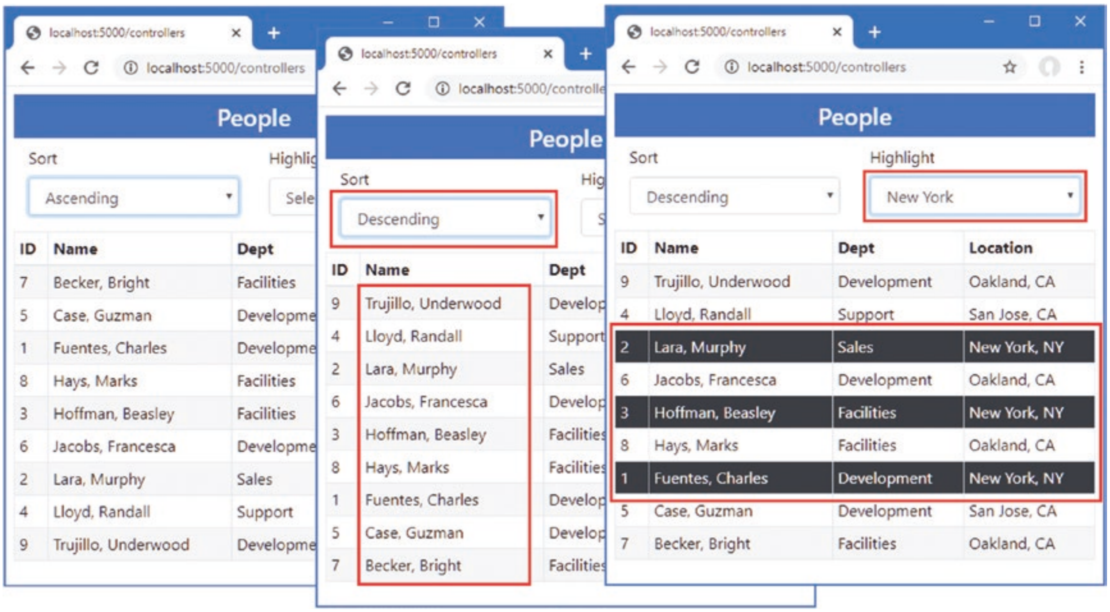

Figure 34-9 *Adding features to a template component*

Reusing a Generic Template Component

The features added to the template component all relied on the generic type parameter, which allows the component to modify the content it presents without being tied to a specific class. The result is a component that can be used to display, sort, and highlight any data type wherever a table is required. Add a Razor Component named `DepartmentList.razor` to the `Blazor` folder with the content shown in Listing 34-24.

Listing 34-24. The Contents of the DepartmentList.razor File in the Blazor Folder

```
<TableTemplate RowType="Department" RowData="Departments"
    Highlight="@(d => d.Name)"
    SortDirection="@(d => d.Name)">
    <Header>
        <tr><th>ID</th><th>Name</th><th>People</th><th>Locations</th></tr>
    </Header>
    <RowTemplate Context="d">
        <td>@d.Departmentid</td>
        <td>@d.Name</td>
        <td>@(String.Join(", ", d.People.Select(p => p.Surname)))</td>
        <td>
            @(String.Join(", ", d.People.Select(p => p.Location.City).Distinct()))
        </td>
    </RowTemplate>
</TableTemplate>

@code {

    [Inject]
    public DataContext Context { get; set; }

    public IEnumerable<Department> Departments => Context.Departments
            .Include(d => d.People).ThenInclude(p => p.Location);
}
```

The TableTemplate component is used to present the user with a list of the Department objects in the database, along with details of the related Person and Location objects, which are queried with the Entity Framework Core Include and ThenInclude methods. Listing 34-25 changes the Razor Component displayed by the Razor Page named Blazor.

Listing 34-25. Changing the Component in the Blazor.cshtml File in the Pages Folder

```
@page "/pages/blazor"

<h4 class="bg-primary text-white text-center p-2">Departments</h4>

<component type="typeof(Advanced.Blazor.DepartmentList)" render-mode="Server" />
```

Restart ASP.NET Core and request http://localhost:5000/pages/blazor. The response will be presented using the templated component, as shown in Figure 34-10.

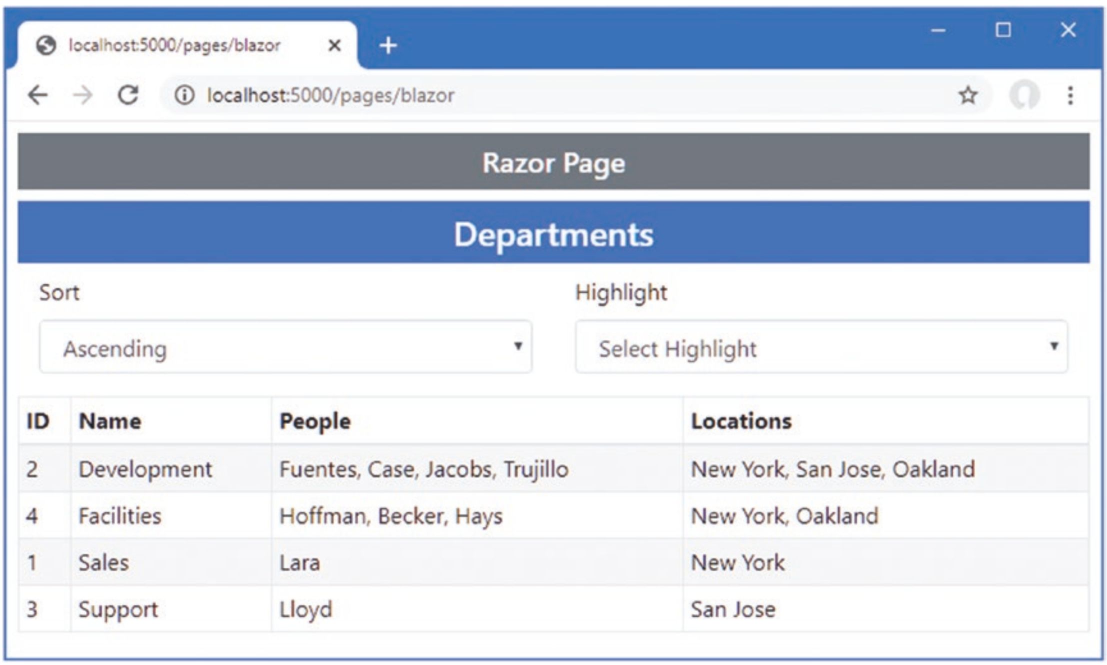

Figure 34-10. *Reusing a generic template component*

Cascading Parameters

As the number of components increases, it can be useful for a component to provide configuration data to descendants deep in the hierarchy of components. This can be done by having each component in the chain receive the data and pass it on to all of its children, but that is error-prone and requires every component to participate in the process, even if none of its descendants uses the data it passes on.

Blazor provides a solution to this problem by supporting *cascading parameters*, in which a component provides data values that are available directly to any of its descendants, without being relayed by intermediate components. Cascading parameters are defined using the CascadingValue component, which is used to wrap a section of content, as shown in Listing 34-26.

Listing 34-26. Creating a Cascading Parameter in the DepartmentList.razor File in the Blazor Folder

```
<CascadingValue Name="BgTheme" Value="Theme" IsFixed="false" >
    <TableTemplate RowType="Department" RowData="Departments"
        Highlight="@(d => d.Name)"
        SortDirection="@(d => d.Name)">
        <Header>
```

```
            <tr><th>ID</th><th>Name</th><th>People</th><th>Locations</th></tr>
        </Header>
        <RowTemplate Context="d">
            <td>@d.Departmentid</td>
            <td>@d.Name</td>
            <td>@(String.Join(", ", d.People.Select(p => p.Surname)))</td>
            <td>
                @(String.Join(", ",
                    d.People.Select(p => p.Location.City).Distinct()))
            </td>
        </RowTemplate>
    </TableTemplate>
</CascadingValue>

<SelectFilter Title="@("Theme")" Values="Themes" @bind-SelectedValue="Theme" />

@code {

    [Inject]
    public DataContext Context { get; set; }

    public IEnumerable<Department> Departments => Context.Departments
            .Include(d => d.People).ThenInclude(p => p.Location);

    public string Theme { get; set; } = "info";
    public string[] Themes = new string[] { "primary", "info", "success" };
}
```

The CascadingValue element makes a value available to the components it encompasses and their descendants. The Name attribute specifies the name of the parameter, the Value attribute specifies the value, and the isFixed attribute is used to specify whether the value will change. The CascadingValue element has been used in Listing 34-26 to create a cascading parameter named BgTheme, whose value is set by an instance of the SelectFilter component that presents the user with a selection of Bootstrap CSS theme names.

■ **Tip** Each CascadingValue element creates one cascading parameter. If you need to pass on multiple values, then you can nest the CascadingValue or create a simple parameter that provides multiple settings through a dictionary.

Cascading parameters are received directly by the components that require them with the CascadingParameter attribute, as shown in Listing 34-27.

Listing 34-27. Receiving a Cascading Parameter in the SelectFilter.razor File in the Blazor Folder

```
<div class="form-group p-2 bg-@Theme @TextColor()">
    <label for="select-@Title">@Title</label>
    <select name="select-@Title" class="form-control"
            @onchange="HandleSelect" value="@SelectedValue">
        <option disabled selected>Select @Title</option>
        @foreach (string val in Values) {
            <option value="@val" selected="@(val == SelectedValue)">
                @val
            </option>
        }
    </select>
</div>
```

```
@code {

    [Parameter]
    public IEnumerable<string> Values { get; set; } = Enumerable.Empty<string>();

    [Parameter]
    public string SelectedValue { get; set; }

    [Parameter]
    public string Title { get; set; } = "Placeholder";

    [Parameter(CaptureUnmatchedValues = true)]
    public Dictionary<string, object> Attrs { get; set; }

    [Parameter]
    public EventCallback<string> SelectedValueChanged { get; set; }

    public async Task HandleSelect(ChangeEventArgs e) {
        SelectedValue = e.Value as string;
        await SelectedValueChanged.InvokeAsync(SelectedValue);
    }

    [CascadingParameter(Name ="BgTheme")]
    public string Theme { get; set; }

    public string TextColor() => Theme == null ? "" : "text-white";
}
```

The `CascadingParameter` attribute's `Name` argument is used to specify the name of the cascading parameter. The `BgTheme` parameter defined in Listing 34-26 is received by the `Theme` property in Listing 34-27 and used to set the background for the component. Restart ASP.NET Core and request `http://localhost:5000/pages/blazor`, which produces the response shown in Figure 34-11.

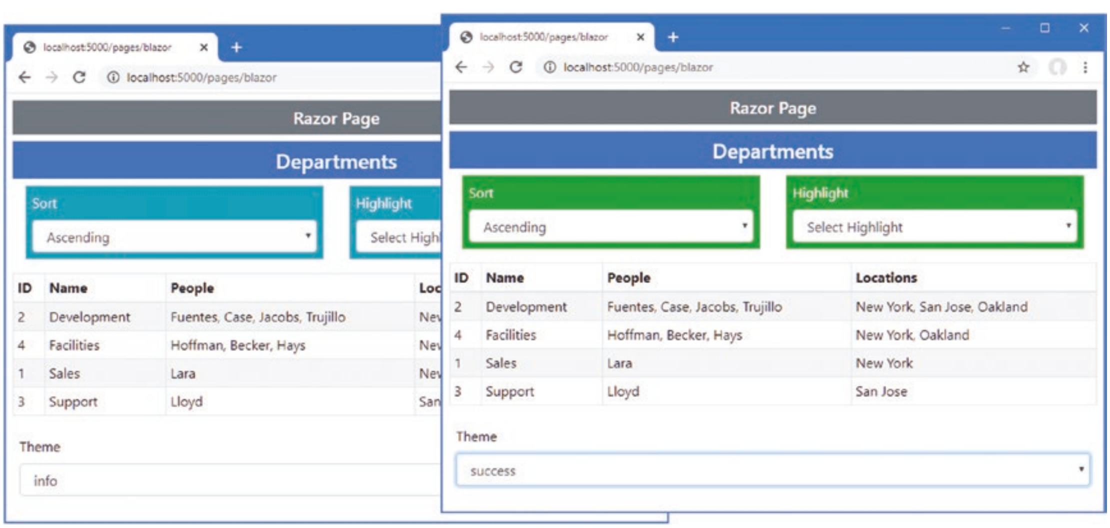

Figure 34-11. *Using a cascading parameter*

There are three instances of the `SelectFilter` component used in this example, but only two of them are within the hierarchy contained by the `CascadingValue` element. The other instance is defined outside of the `CascadingValue` element and does not receive the cascading value.

Handling Errors

In the following sections, I describe the features Blazor provides for dealing with connection errors and unhandled application errors.

Handling Connection Errors

Blazor relies on its persistent HTTP connection between the browser and the ASP.NET Core server. The application cannot function when the connection is disrupted, and a modal error message is displayed that prevents the user from interacting with components.

Blazor allows the connection errors to be customized by defining an element with a specific id, as shown in Listing 34-28.

Listing 34-28. Defining a Connection Error Element in the Blazor.cshtml File in the Pages Folder

```
@page "/pages/blazor"

<h4 class="bg-primary text-white text-center p-2">Departments</h4>

<link rel="stylesheet" href="connectionErrors.css" />

<div id="components-reconnect-modal"
    class="h4 bg-dark text-white text-center my-2 p-2 components-reconnect-hide">
    Blazor Connection Lost
    <div class="reconnect">
        Trying to reconnect...
    </div>
    <div class="failed">
        Reconnection Failed.
        <button class="btn btn-light" onclick="window.Blazor.reconnect()">
            Reconnect
        </button>
    </div>
    <div class="rejected">
        Reconnection Rejected.
        <button class="btn btn-light" onclick="location.reload()">
            Reload
        </button>
    </div>
</div>

<component type="typeof(Advanced.Blazor.DepartmentList)" render-mode="Server" />
```

The id attribute of the custom error element must be components-reconnect-modal. When there is a connection error, Blazor locates this element and adds it to one of four classes, described in Table 34-2.

Table 34-2. *The Connection Error Classes*

Name	Description
components-reconnect-show	The element is added to this class when the connection has been lost and Blazor is attempting a reconnection. The error message should be displayed to the user, and interaction with the Blazor content should be prevented.
components-reconnect-hide	The element is added to this class if the connection is reestablished. The error message should be hidden, and interaction should be permitted.
components-reconnect-failed	The element is added to this class if Blazor reconnection fails. The user can be presented with a button that invokes window.Blazor.reconnect() to attempt reconnection again.
components-reconnect-rejected	The element is added to this class if Blazor is able to reach the server, but the user's connection state has been lost. This typically happens when the server has been restarted. The user can be presented with a button that invokes location.reload() to reload the application and try again.

The element isn't added to any of these classes initially, so I have explicitly added it to the `components-reconnect-hide` class so that it isn't visible until a problem occurs.

I want to present specific messages to the user for each of the conditions that can arise during reconnection. To this end, I added elements that display a message for each condition. To manage their visibility, add a CSS Stylesheet named `connectionErrors.css` to the `wwwroot` folder and use it to define the styles shown in Listing 34-29.

Listing 34-29. The Contents of the connectionErrors.css File in the wwwroot Folder

```
#components-reconnect-modal {
    position: fixed; top: 0; right: 0; bottom: 0;
    left: 0; z-index: 1000; overflow: hidden; opacity: 0.9;
}

.components-reconnect-hide { display: none; }
.components-reconnect-show { display: block; }

.components-reconnect-show > .reconnect { display: block; }
.components-reconnect-show > .failed,
.components-reconnect-show > .rejected {
    display: none;
}

.components-reconnect-failed > .failed {
    display: block;
}
.components-reconnect-failed > .reconnect,
.components-reconnect-failed > .rejected {
    display: none;
}

.components-reconnect-rejected > .rejected {
    display: block;
}
.components-reconnect-rejected > .reconnect,
.components-reconnect-rejected > .failed {
    display: none;
}
```

These styles show the `components-reconnect-modal` element as a modal item, with its visibility determined by the `components-reconnect-hide` and `components-reconnect-show` classes. The visibility of the specific messages is toggled based on the application of the classes in Table 34-2.

To see the effect, restart ASP.NET Core and request `http://localhost:5000/pages/blazor`. Wait until the component is displayed and then stop the ASP.NET Core server. You will see an initial error message as Blazor attempts to reconnect. After a few seconds, you will see the message that indicates that reconnection has failed.

Restart ASP.NET Core and request `http://localhost:5000/pages/blazor`. Wait until the component is displayed and then restart ASP.NET Core. This time Blazor will be able to connect to the server, but the connection will be rejected because the server restart has caused the connection state to be lost. Figure 34-12 shows both sequences of error messages.

■ **Tip** It is not possible to test successful connection recovery with just the browser because there is no way to interrupt the persistent HTTP connection. I use the excellent Fiddler proxy, `https://www.telerik.com/fiddler`, which allows me to terminate the connection without stopping the ASP.NET Core server.

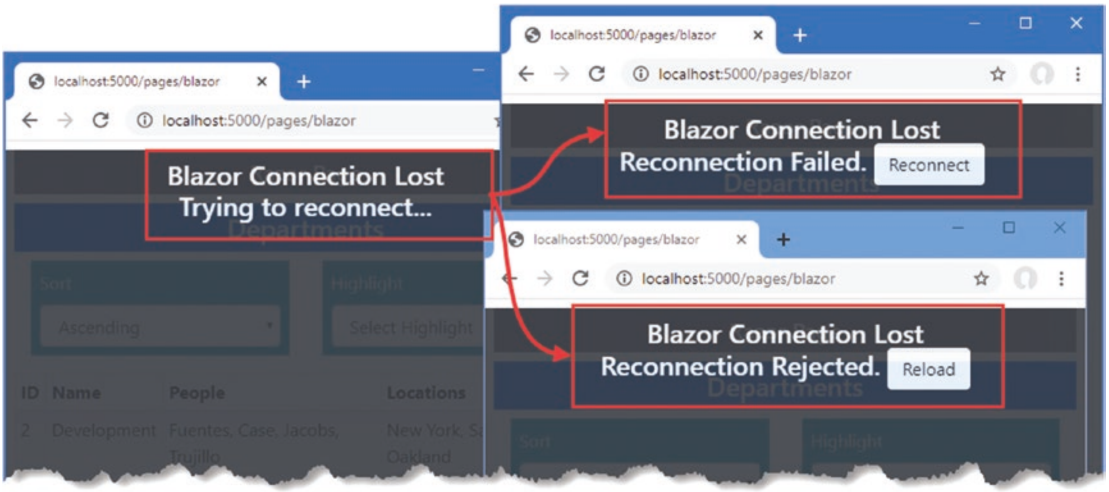

Figure 34-12. *Handling connection errors*

Handling Uncaught Application Errors

Blazor does not respond well to uncaught application errors, which are almost always treated as terminal. To see the default error behavior, add the elements shown in Listing 34-30 to the DepartmentList component.

Listing 34-30. Adding Elements in the DepartmentList.razor File in the Blazor Folder

```
<CascadingValue Name="BgTheme" Value="Theme" IsFixed="false" >
    <TableTemplate RowType="Department" RowData="Departments"
        Highlight="@(d => d.Name)"
        SortDirection="@(d => d.Name)">
        <Header>
            <tr><th>ID</th><th>Name</th><th>People</th><th>Locations</th></tr>
        </Header>
        <RowTemplate Context="d">
            <td>@d.Departmentid</td>
            <td>@d.Name</td>
            <td>@(String.Join(", ", d.People.Select(p => p.Surname)))</td>
            <td>
                @(String.Join(", ",
                    d.People.Select(p => p.Location.City).Distinct()))
            </td>
        </RowTemplate>
    </TableTemplate>
</CascadingValue>

<SelectFilter Title="@("Theme")" Values="Themes" @bind-SelectedValue="Theme" />

<button class="btn btn-danger" @onclick="@(() => throw new Exception())">
    Error
</button>

@code {

    // ...statements omitted for brevity...
}
```

Restart ASP.NET Core, request `http://localhost:5000/pages/blazor`, and click the Error button. There is no visible change in the browser, but the exception thrown at the server when the button was clicked has proved fatal: the user can still choose values using the select elements because these are presented by the browser, but the event handlers that respond to selections no longer work, and the application is essentially dead.

When there is an unhandled application error, Blazor looks for an element whose `id` is `blazor-error-ui` and sets its CSS `display` property to `block`. Listing 34-31 adds an element with this `id` to the `Blazor.cshtml` file styled to present a useful message.

Listing 34-31. Adding an Error Element in the Blazor.cshtml File in the Pages Folder

```
@page "/pages/blazor"

<h4 class="bg-primary text-white text-center p-2">Departments</h4>

<link rel="stylesheet" href="connectionErrors.css" />

<div id="components-reconnect-modal"
    class="h4 bg-dark text-white text-center my-2 p-2 components-reconnect-hide">
    Blazor Connection Lost
    <div class="reconnect">
        Trying to reconnect...
    </div>
    <div class="failed">
        Reconnection Failed.
        <button class="btn btn-light" onclick="window.Blazor.reconnect()">
            Reconnect
        </button>
    </div>
    <div class="rejected">
        Reconnection Rejected.
        <button class="btn btn-light" onclick="location.reload()">
            Reload
        </button>
    </div>
</div>

<div id="blazor-error-ui"
     class="text-center bg-danger h6 text-white p-2 fixed-top w-100"
     style="display:none">
    An error has occurred. This application will not respond until reloaded.
    <button class="btn btn-sm btn-primary" onclick="location.reload()">
        Reload
    </button>
</div>

<component type="typeof(Advanced.Blazor.DepartmentList)" render-mode="Server" />
```

When the element is shown, the user will be presented with a warning and a button that reloads the browser. To see the effect, restart ASP.NET Core, request `http://localhost:5000/pages/blazor`, and click the Error button, which will display the message shown in Figure 34-13.

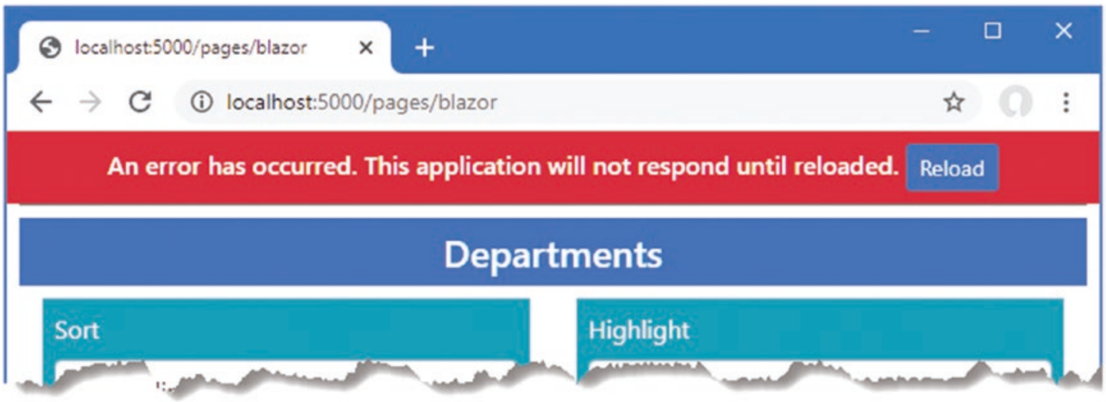

Figure 34-13. *Displaying an error message*

Summary

In this chapter, I showed you how to combine Razor Components to create more complex features. I showed you how to create parent/child relationships between components, how to configure components with attributes, and how to create custom events to signal when important changes occur. I also showed you how a component can receive content from its parent and how to generate content consistently using template components, which can be defined with one or more generic type parameters. I finished the chapter by demonstrating how Blazor applications can react to connection and application errors. In the next chapter, I describe the advanced features that Blazor provides.

CHAPTER 35

Advanced Blazor Features

In this chapter, I explain how Blazor supports URL routing so that multiple components can be displayed through a single request. I show you how to set up the routing system, how to define routes, and how to create common content in a layout.

This chapter also covers the component lifecycle, which allows components to participate actively in the Blazor environment, which is especially important once you start using the URL routing feature. Finally, this chapter explains the different ways that components can interact outside of the parent/child relationships described in earlier chapters. Table 35-1 puts these features in context.

Table 35-1. *Putting Blazor Routing and Lifecycle Component Interactions in Context*

Question	Answer
What are they?	The routing feature allows components to respond to changes in the URL without requiring a new HTTP connection. The lifecycle feature allows components to define methods that are invoked as the application executes, and the interaction features provide useful ways of communicating between components and with other JavaScript code.
Why are they useful?	These features allow the creation of complex applications that take advantage of the Blazor architecture.
How are they used?	URL routing is set up using built-in components and configured using `@page` directives. The lifecycle features are used by overriding methods in a component's `@code` section. The interaction features are used in different ways depending on what a component is interacting with.
Are there any pitfalls or limitations?	These are advanced features that must be used with care, especially when creating interactions outside of Blazor.
Are there any alternatives?	All of the features described in this chapter are optional, but it is hard to create complex applications without them.

Table 35-2 summarizes the chapter.

Table 35-2. *Chapter Summary*

Problem	Solution	Listing
Selecting components based on the current URL	Use URL routing	6–12
Defining content that will be used by multiple components	Use a layout	13, 14
Responding to the stages of the component's lifecycle	Implement the lifecycle notification methods	15–17
Coordinating the activities of multiple components	Retain references with the `@ref` expression	18–19
Coordinating with code outside of Blazor	Use the interoperability features	20–35

A. Freeman, *Pro ASP.NET Core 3*, https://doi.org/10.1007/978-1-4842-5440-0_35

Preparing for This Chapter

This chapter uses the Advanced project from Chapter 35. No changes are required for this chapter.

■ **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-3. See Chapter 1 for how to get help if you have problems running the examples.

Open a new PowerShell command prompt, navigate to the folder that contains the `Advanced.csproj` file, and run the command shown in Listing 35-1 to drop the database.

Listing 35-1. Dropping the Database

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

Select Start Without Debugging or Run Without Debugging from the Debug menu or use the PowerShell command prompt to run the command shown in Listing 35-2.

Listing 35-2. Running the Example Application

```
dotnet run
```

Use a browser to request `http://localhost:5000/controllers`, which will display a list of data items. Request `http://localhost:5000/pages/blazor`, and you will see the component from Chapter 34 that I used to demonstrate bindings. Figure 35-1 shows both responses.

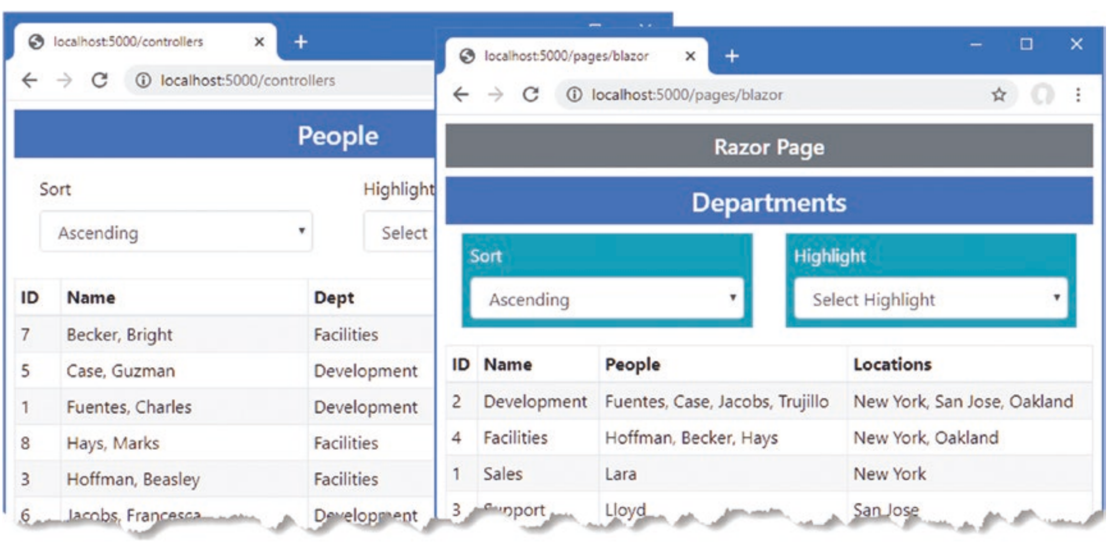

Figure 35-1. *Running the example application*

Using Component Routing

Blazor includes support for selecting the components to display to the user based on the ASP.NET Core routing system so that the application responds to changes in the URL by displaying different Razor Components. To get started, add a Razor Component named `Routed.razor` to the `Blazor` folder with the content shown in Listing 35-3.

Listing 35-3. The Contents of the Routed.razor File in the Blazor Folder

```
<Router AppAssembly="typeof(Startup).Assembly">
    <Found>
        <RouteView RouteData="@context" />
    </Found>
    <NotFound>
        <h4 class="bg-danger text-white text-center p-2">
            No Matching Route Found
        </h4>
    </NotFound>
</Router>
```

The `Router` component is included with ASP.NET Core and provides the link between Blazor and the ASP.NET Core routing features. `Router` is a generic template component that defines `Found` and `NotFound` sections.

The `Router` component requires the `AppAssembly` attribute, which specifies the .NET assembly to use. For most projects this is the current assembly, which is specified like this:

```
...
<Router AppAssembly="typeof(Startup).Assembly">
...
```

The type of the `Router` component's `Found` property is `RenderFragment<RouteData>`, which is passed on to the `RouteView` component through its `RouteData` property, like this:

```
...
<Found>
    <RouteView RouteData="@context" />
</Found>
...
```

The `RouteView` component is responsible for displaying the component matched by the current route and, as I explain shortly, for displaying common content through layouts. The type of the `NotFound` property is `RenderFragment`, without a generic type argument, and displays a section of content when no component can be found for the current route.

Preparing the Razor Page

Individual components can be displayed in existing controller views and Razor Pages, as previous chapters have shown. But when using component routing, it is preferable to create a set of URLs that are distinct to working with Blazor because the way that URLs are supported is limited and leads to tortured workarounds. Add a Razor Page named `_Host.cshtml` to the `Pages` folder and add the content shown in Listing 35-4.

Listing 35-4. The Contents of the _Host.cshtml File in the Pages Folder

```
@page "/"
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
    <base href="~/" />
</head>
<body>
```

```
    <div class="m-2">
        <component type="typeof(Advanced.Blazor.Routed)" render-mode="Server" />
    </div>
    <script src="_framework/blazor.server.js"></script>
</body>
</html>
```

This page contains a `component` element that applies the `Routed` component defined in Listing 35-4 and a `script` element for the Blazor JavaScript code. There is also a `link` element for the Bootstrap CSS stylesheet. Alter the configuration for the example application to use the `_Host.cshtml` file as a fallback when requests are not matched by the existing URL routes, as shown in Listing 35-5.

Listing 35-5. Adding the Fallback in the Startup.cs File in the Advanced Folder

```
...
public void Configure(IApplicationBuilder app, DataContext context) {

    app.UseDeveloperExceptionPage();
    app.UseStaticFiles();
    app.UseRouting();

    app.UseEndpoints(endpoints => {
        endpoints.MapControllerRoute("controllers",
            "controllers/{controller=Home}/{action=Index}/{id?}");
        endpoints.MapDefaultControllerRoute();
        endpoints.MapRazorPages();
        endpoints.MapBlazorHub();
        endpoints.MapFallbackToPage("/_Host");
    });

    SeedData.SeedDatabase(context);
}
...
```

The `MapFallbackToPage` method configures the routing system to use the `_Host` page as a last resort for unmatched requests.

Adding Routes to Components

Components declare the URLs for which they should be displayed using `@page` directives. Listing 35-6 adds the `@page` directive to the `PeopleList` component.

Listing 35-6. Adding a Directive in the PeopleList.razor File in the Blazor Folder

```
@page "/people"

<TableTemplate RowType="Person" RowData="People"
        Highlight="@(p => p.Location.City)" SortDirection="@(p => p.Surname)">
    <Header>
        <tr><th>ID</th><th>Name</th><th>Dept</th><th>Location</th></tr>
    </Header>
    <RowTemplate Context="p">
        <td>@p.PersonId</td>
        <td>@p.Surname, @p.Firstname</td>
        <td>@p.Department.Name</td>
        <td>@p.Location.City, @p.Location.State</td>
    </RowTemplate>
</TableTemplate>

@code {
```

```
    [Inject]
    public DataContext Context { get; set; }

    public IEnumerable<Person> People => Context.People
            .Include(p => p.Department)
            .Include(p => p.Location);
}
```

The directive in Listing 35-6 means the `PeopleList` component will be displayed for the `http://localhost:5000/people` URL. Components can declare support for more than one route using multiple `@page` directives. Listing 35-7 adds `@page` directives to the `DepartmentList` component to support two URLs.

Listing 35-7. Adding a Directive in the DepartmentList.razor File in the Blazor Folder

```
@page "/departments"
@page "/depts"

<CascadingValue Name="BgTheme" Value="Theme" IsFixed="false" >
    <TableTemplate RowType="Department" RowData="Departments"
        Highlight="@(d => d.Name)"
        SortDirection="@(d => d.Name)">
        <Header>
            <tr><th>ID</th><th>Name</th><th>People</th><th>Locations</th></tr>
        </Header>
        <RowTemplate Context="d">
            <td>@d.Departmentid</td>
            <td>@d.Name</td>
            <td>@(String.Join(", ", d.People.Select(p => p.Surname)))</td>
            <td>
                @(String.Join(", ",
                    d.People.Select(p => p.Location.City).Distinct()))
            </td>
        </RowTemplate>
    </TableTemplate>
</CascadingValue>

<SelectFilter Title="@("Theme")" Values="Themes" @bind-SelectedValue="Theme" />

<button class="btn btn-danger" @onclick="@(() => throw new Exception())">
    Error
</button>

@code {

    [Inject]
    public DataContext Context { get; set; }

    public IEnumerable<Department> Departments => Context.Departments
            .Include(d => d.People).ThenInclude(p => p.Location);

    public string Theme { get; set; } = "info";
    public string[] Themes = new string[] { "primary", "info", "success" };
}
```

Most of the routing pattern features described in Chapter 13 can be used in `@page` expressions, except catchall segment variables and optional segment variables. Using two `@page` expressions, one with a segment variable, can be used to re-create the optional variable feature, as demonstrated in Chapter 36, where I show you how to implement a CRUD application using Blazor.

To see the basic Razor Component routing feature at work, restart ASP.NET Core and request `http://localhost:5000/people` and `http://localhost:5000/depts`. Each URL displays one of the components in the application, as shown in Figure 35-2.

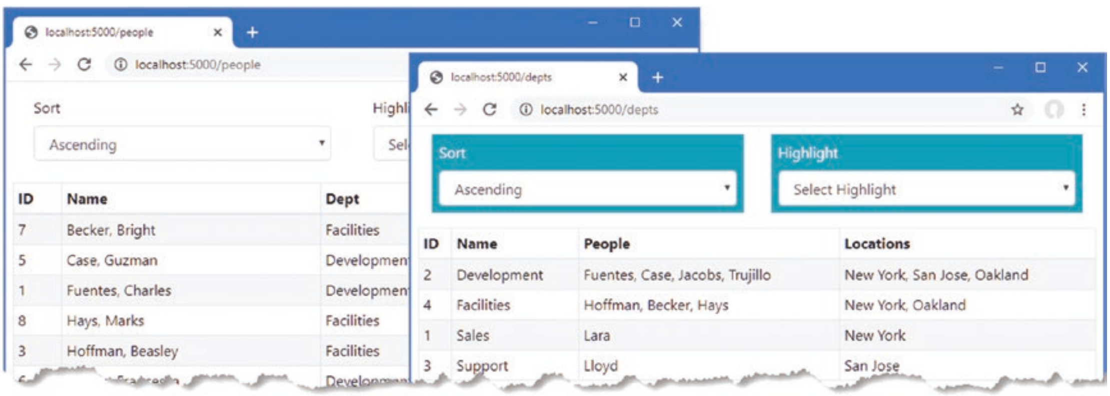

Figure 35-2. *Enabling Razor Component routing in the example application*

Setting a Default Component Route

The configuration change in Listing 35-5 set up the fallback route for requests in the `Startup` class. A corresponding route is required in one of the application's components to identify the component that should be displayed for the application's default URL, `http://localhost:5000`, as shown in Listing 35-8.

Listing 35-8. Defining the Default Route in the PeopleList.razor File in the Blazor Folder

```
@page "/"
@page "/people"

<TableTemplate RowType="Person" RowData="People"
        Highlight="@(p => p.Location.City)" SortDirection="@(p => p.Surname)">
    <Header>
        <tr><th>ID</th><th>Name</th><th>Dept</th><th>Location</th></tr>
    </Header>
    <RowTemplate Context="p">
        <td>@p.PersonId</td>
        <td>@p.Surname, @p.Firstname</td>
        <td>@p.Department.Name</td>
        <td>@p.Location.City, @p.Location.State</td>
    </RowTemplate>
</TableTemplate>

@code {

    [Inject]
    public DataContext Context { get; set; }

    public IEnumerable<Person> People => Context.People
            .Include(p => p.Department)
            .Include(p => p.Location);
}
```

Restart ASP.NET Core and request `http://localhost:5000`, and you will see the content produced by the `PeopleList` component, as shown in Figure 35-3.

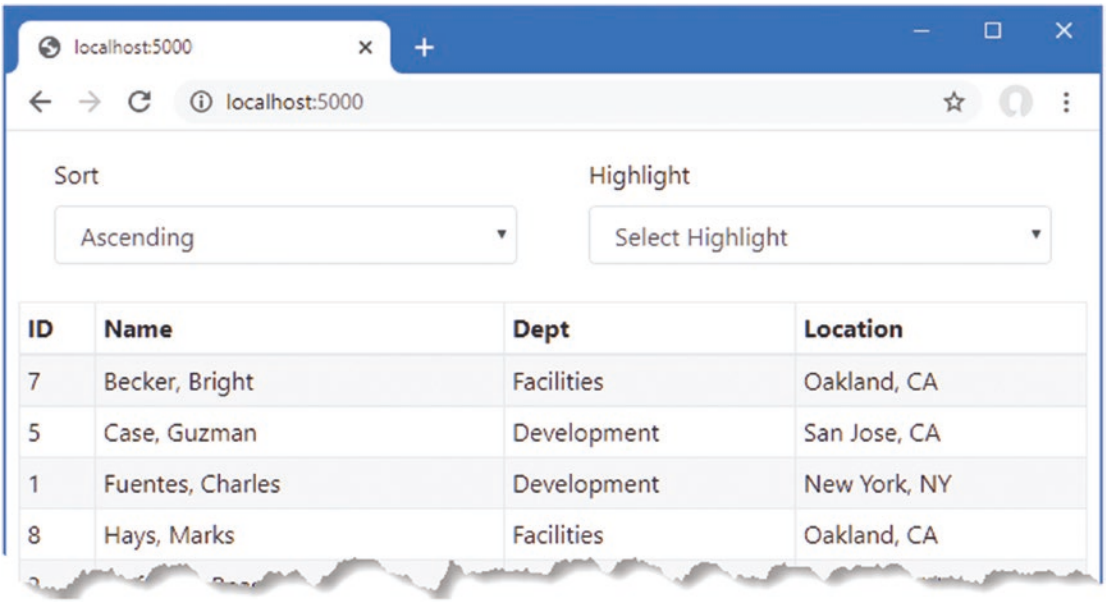

Figure 35-3. *Displaying a component for the default URL*

Navigating Between Routed Components

The basic routing configuration is in place, but it may not be obvious why using routes offers any advantages over the independent components demonstrated in earlier chapters. Improvements come through the `NavLink` component, which renders anchor elements that are wired into the routing system. Listing 35-9 adds `NavLink` to the `PeopleList` component.

Listing 35-9. Adding Navigation in the PeopleList.razor File in the Blazor Folder

```
@page "/"
@page "/people"

<TableTemplate RowType="Person" RowData="People"
        Highlight="@(p => p.Location.City)" SortDirection="@(p => p.Surname)">
    <Header>
        <tr><th>ID</th><th>Name</th><th>Dept</th><th>Location</th></tr>
    </Header>
    <RowTemplate Context="p">
        <td>@p.PersonId</td>
        <td>@p.Surname, @p.Firstname</td>
        <td>@p.Department.Name</td>
        <td>@p.Location.City, @p.Location.State</td>
    </RowTemplate>
</TableTemplate>

<NavLink class="btn btn-primary" href="/depts">Departments</NavLink>

@code {

    [Inject]
    public DataContext Context { get; set; }
```

```
    public IEnumerable<Person> People => Context.People
            .Include(p => p.Department)
            .Include(p => p.Location);
}
```

Unlike the anchor elements used in other parts of ASP.NET Core, `Navlink` components are configured using URLs and not component, page, or action names. The `NavLink` in this example navigates to the URL supported by the `@page` directive of the `DepartmentList` component.

Navigation can also be performed programmatically, which is useful when a component responds to an event and then needs to navigate to a different URL, as shown in Listing 35-10.

Listing 35-10. Navigating Programmatically in the DepartmentList.razor File in the Blazor Folder

```
@page "/departments"
@page "/depts"

<CascadingValue Name="BgTheme" Value="Theme" IsFixed="false" >
    <TableTemplate RowType="Department" RowData="Departments"
        Highlight="@(d => d.Name)"
        SortDirection="@(d => d.Name)">
        <Header>
            <tr><th>ID</th><th>Name</th><th>People</th><th>Locations</th></tr>
        </Header>
        <RowTemplate Context="d">
            <td>@d.Departmentid</td>
            <td>@d.Name</td>
            <td>@(String.Join(", ", d.People.Select(p => p.Surname)))</td>
            <td>
                @(String.Join(", ",
                    d.People.Select(p => p.Location.City).Distinct()))
            </td>
        </RowTemplate>
    </TableTemplate>
</CascadingValue>

<SelectFilter Title="@("Theme")" Values="Themes" @bind-SelectedValue="Theme" />

<button class="btn btn-primary" @onclick="HandleClick">People</button>

@code {

    [Inject]
    public DataContext Context { get; set; }

    public IEnumerable<Department> Departments => Context.Departments
            .Include(d => d.People).ThenInclude(p => p.Location);

    public string Theme { get; set; } = "info";
    public string[] Themes = new string[] { "primary", "info", "success" };

    [Inject]
    public NavigationManager NavManager { get; set; }

    public void HandleClick() => NavManager.NavigateTo("/people");
}
```

The `NavigationManager` class provides programmatic access to navigation. Table 35-3 describes the most important members provided by the `NavigationManager` class.

Table 35-3. *Useful NavigationManager Members*

Name	Description
`NavigateTo(url)`	This method navigates to the specified URL without sending a new HTTP request.
`ToAbsoluteUri(path)`	This method converts a relative path to a complete URL.
`ToBaseRelativePath(url)`	This method gets a relative path from a complete URL.
`LocationChanged`	This event is triggered when the location changes.
`Uri`	This property returns the current URL.

The `NavigationManager` class is provided as a service and is received by Razor Components using the `Inject` attribute, which provides access to the dependency injection features described in Chapter 14.

The `NavigationManager.NavigateTo` method navigates to a URL and is used in this example to navigate to the `/people` URL, which will be handled by the `PeopleList` component.

To see why routing and navigation are important, restart ASP.NET Core and request `http://localhost:5000/people`. Click the Departments link, which is styled as a button, and the `DepartmentList` component will be displayed. Click the People link, and you will return to the `PeopleList` component, as shown in Figure 35-4.

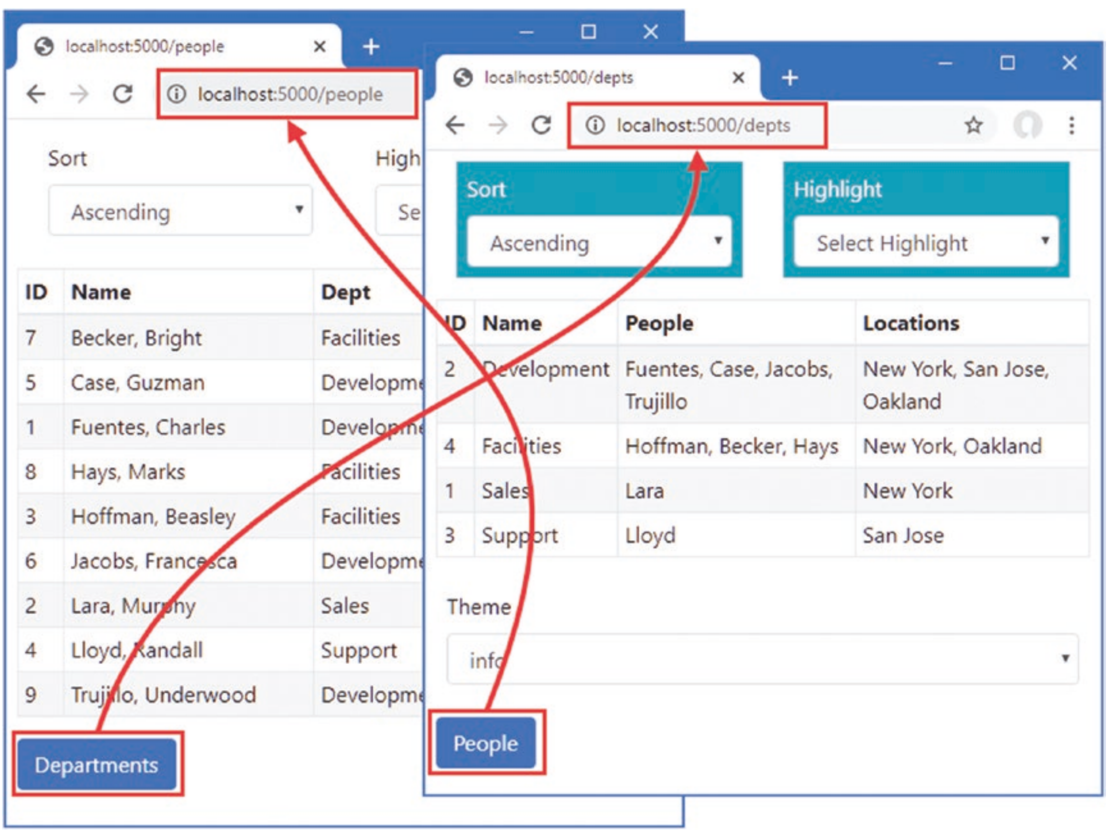

Figure 35-4. *Navigating between routed components*

If you perform this sequence with the F12 developer tools open, you will see that the transition from one component to the next is done without needing a separate HTTP request, even though the URL displayed by the browser changes. Blazor delivers the content rendered by each component over the persistent HTTP connection that is established when the first component is displayed and uses a JavaScript API to navigate without loading a new HTML document.

■ **Tip** The `NavigationManager.NavigateTo` method accepts an optional argument that, when `true`, forces the browser to send a new HTTP request and reload the HTML document.

Receiving Routing Data

Components can receive segment variables by decorating a property with the `Parameter` attribute. To demonstrate, add a Razor Component named `PersonDisplay.razor` to the `Blazor` folder with the content shown in Listing 35-11.

Listing 35-11. The Contents of the PersonDisplay.razor in the Blazor Folder

```
@page "/person"
@page "/person/{id:long}"

<h5>Editor for Person: @Id</h5>

<NavLink class="btn btn-primary" href="/people">Return</NavLink>

@code {

    [Parameter]
    public long Id { get; set; }
}
```

This component doesn't do anything other than displaying the value it receives from the routing data until I add features later in the chapter. The `@page` expression includes a segment variable named `id`, whose type is specified as `long`. The component receives the value assigned to the segment variable by defining a property with the same name and decorating it with the `Parameter` attribute.

■ **Tip** If you don't specify a type for segment variables in the `@page` expression, then you must set the type of the property to be `string`.

Listing 35-12 uses the `NavLink` component to create navigation links for each of the `Person` objects displayed by the `PeopleList` component.

Listing 35-12. Adding Navigation Links in the PeopleList.razor File in the Blazor Folder

```
@page "/"
@page "/people"

<TableTemplate RowType="Person" RowData="People"
        Highlight="@(p => p.Location.City)" SortDirection="@(p => p.Surname)">
    <Header>
        <tr><th>ID</th><th>Name</th><th>Dept</th><th>Location</th>
            <td></td>
        </tr>
    </Header>
    <RowTemplate Context="p">
        <td>@p.PersonId</td>
        <td>@p.Surname, @p.Firstname</td>
        <td>@p.Department.Name</td>
        <td>@p.Location.City, @p.Location.State</td>
        <td>
            <NavLink class="btn btn-sm btn-info" href="@GetEditUrl(p.PersonId)">
                Edit
            </NavLink>
```

```
        </td>
    </RowTemplate>
</TableTemplate>

<NavLink class="btn btn-primary" href="/depts">Departments</NavLink>

@code {

    [Inject]
    public DataContext Context { get; set; }

    public IEnumerable<Person> People => Context.People
            .Include(p => p.Department)
            .Include(p => p.Location);

    public string GetEditUrl(long id) => $"/person/{id}";
}
```

Razor Components do no support mixing static content and Razor expressions in attribute values. Instead, I have defined the GetEditUrl method to generate the navigation URLs for each Person object, which is called to produce the value for the NavLink href attributes.

Restart ASP.NET Core, request http://localhost:5000/people, and click one of the Edit buttons. The browser will navigate to the new URL without reloading the HTML document and display the placeholder content generated by the PersonDisplay component, as shown in Figure 35-5, which shows how a component can receive data from the routing system.

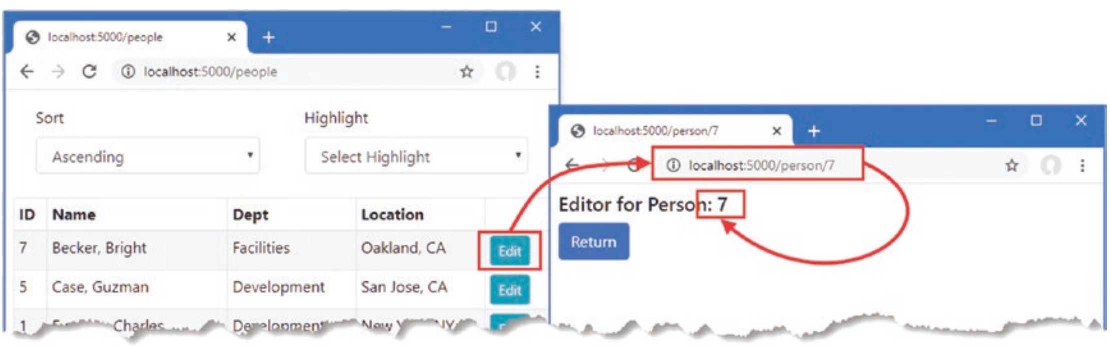

Figure 35-5. *Receiving data from the routing system in a Razor Component*

Defining Common Content Using Layouts

Layouts are template components that provide common content for Razor Components. To create a layout, add a Razor Component called NavLayout.razor to the Blazor folder and add the content shown in Listing 35-13.

Listing 35-13. The Contents of the NavLayout.razor File in the Blazor Folder

```
@inherits LayoutComponentBase

<div class="container-fluid">
    <div class="row">
        <div class="col-3">
            @foreach (string key in NavLinks.Keys) {
                <NavLink class="btn btn-outline-primary btn-block"
                         href="@NavLinks[key]"
                         ActiveClass="btn-primary text-white"
                         Match="NavLinkMatch.Prefix">
```

```
                    @key
                </NavLink>
            }
        </div>
        <div class="col">
            @Body
        </div>
    </div>
</div>

@code {

    public Dictionary<string, string> NavLinks
        = new Dictionary<string, string> {
            {"People", "/people" },
            {"Departments", "/depts" },
            {"Details", "/person" }
        };
}
```

Layouts use `@inherits` expression to specify the `LayoutComponentBase` class as the base for the class generated from the Razor Component. The `LayoutComponentBase` class defines a `RenderFragment` class named `Body` that is used to specify the content from components within the common content displayed by the layout. In this example, the layout component creates a grid layout that displays a set of `NavLink` components for each of the components in the application. The `NavLink` components are configured with two new attributes, described in Table 35-4.

Table 35-4. *The NavLink Configuration Attributes*

Name	Description
`ActiveClass`	This attribute specifies one or more CSS classes that the anchor element rendered by the `NavLink` component will be added to when the current URL matches the `href` attribute value.
`Match`	This attribute specifies how the current URL is matched to the `href` attribute, using a value from the `NavLinkMatch` enum. The values are `Prefix`, which considers a match if the `href` matches the start of the URL, and `All`, which requires the entire URL to be the same.

The `NavLink` components are configured to use `Prefix` matching and to add the anchor elements they render to the Bootstrap `btn-primary` and `text-white` classes when there is a match.

Applying a Layout

There are three ways that a layout can be applied. A component can select its own layout using an `@layout` expression. A parent can use a layout for its child components by wrapping them in the built-in `LayoutView` component. A layout can be applied to all components by setting the `DefaultLayout` attribute of the `RouteView` component, as shown in Listing 35-14.

Listing 35-14. Applying a Layout in the Routed.razor File in the Blazor Folder

```
<Router AppAssembly="typeof(Startup).Assembly">
    <Found>
        <RouteView RouteData="@context" DefaultLayout="typeof(NavLayout)" />
    </Found>
    <NotFound>
        <h4 class="bg-danger text-white text-center p-2">
            Not Matching Route Found
        </h4>
    </NotFound>
</Router>
```

Restart ASP.NET Core and request `http://localhost:5000/people`. The layout will be displayed with the content rendered by the `PeopleList` component. The navigation buttons on the left side of the layout can be used to navigate through the application, as shown in Figure 35-6.

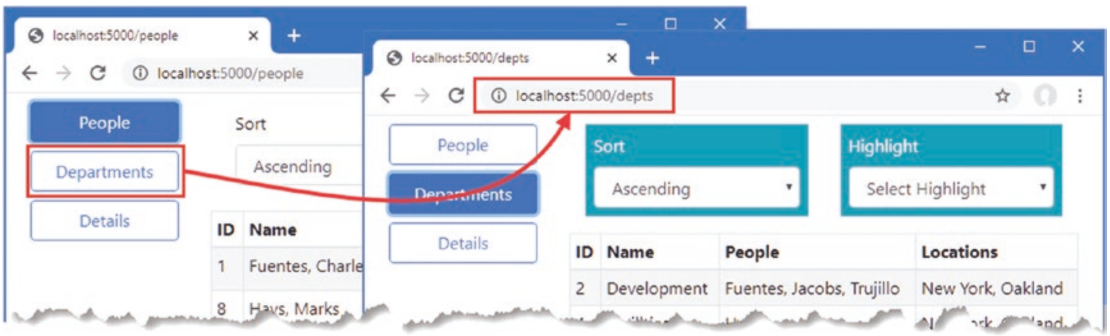

Figure 35-6. *Using a layout component*

■ **Note** If you request `http://localhost:5000`, you will see the content from the `PeopleList` component, but the corresponding navigation button will not be highlighted. I show you how to resolve this problem in the next section.

Understanding the Component Lifecycle Methods

Razor Components have a well-defined lifecycle, which is represented with methods that components can implement to receive notifications of key transitions. Table 35-5 describes the lifecycle methods.

Table 35-5. *The Razor Component Lifecycle Methods*

Name	Description
`OnInitialized()` `OnInitializedAsync()`	These methods are invoked when the component is first initialized.
`OnParametersSet()` `OnParametersSetAsync()`	These methods are invoked after the values for properties decorated with the `Parameter` attribute have been applied.
`ShouldRender()`	This method is called before the component's content is rendered to update the content presented to the user. If the method returns `false`, the component's content will not be rendered, and the update is suppressed. This method does not suppress the initial rendering for the component.
`OnAfterRender(first)` `OnAfterRenderAsync(first)`	This method is invoked after the component's content is rendered. The `bool` parameter is `true` when Blazor performs the initial render for the component.

Using either the `OnInitialized` or `OnParameterSet` method is useful for setting the initial state of the component. The layout defined in the previous section doesn't deal with the default URL because the `NavLink` component matches only a single URL. The same issue exists for the `DepartmentList` component, which can be requested using the `/departments` and `/depts` paths.

UNDERSTANDING LIFECYCLES FOR ROUTED COMPONENTS

When using URL routing, components can be removed from the display when the URL changes. Components can implement the `System.IDisposable` interface, and Blazor will call the method when the component is removed.

Creating a component that matches multiple URLs requires the use of lifecycle methods. To understand why, add a Razor Component named `MultiNavLink.razor` to the `Blazor` folder with the content shown in Listing 35-15.

Listing 35-15. The Contents of the MultiNavLink.razor File in the Blazor Folder

```
<a class="@ComputedClass" @onclick="HandleClick"  href="">
    @ChildContent
</a>

@code {

    [Inject]
    public NavigationManager NavManager { get; set; }

    [Parameter]
    public IEnumerable<string> Href { get; set; }

    [Parameter]
    public string Class { get; set; }

    [Parameter]
    public string ActiveClass { get; set; }

    [Parameter]
    public NavLinkMatch? Match { get; set; }

    public NavLinkMatch ComputedMatch { get =>
            Match ?? (Href.Count() == 1 ? NavLinkMatch.Prefix : NavLinkMatch.All); }

    [Parameter]
    public RenderFragment ChildContent { get; set; }

    public string ComputedClass { get; set; }

    public void HandleClick() {
        NavManager.NavigateTo(Href.First());
    }

    private void CheckMatch(string currentUrl) {
        string path = NavManager.ToBaseRelativePath(currentUrl);
        path = path.EndsWith("/") ? path.Substring(0, path.Length - 1) : path;
        bool match = Href.Any(href => ComputedMatch == NavLinkMatch.All
                ? path == href : path.StartsWith(href));
        ComputedClass = match ? $"{Class} {ActiveClass}" : Class;
    }

    protected override void OnParametersSet() {
        ComputedClass = Class;
        NavManager.LocationChanged += (sender, arg) => CheckMatch(arg.Location);
        Href = Href.Select(h => h.StartsWith("/") ? h.Substring(1) : h);
        CheckMatch(NavManager.Uri);
    }
}
```

This component works in the same way as a regular `NavLink` but accepts an array of paths to match. The component relies on the `OnParametersSet` lifecycle method because some initial setup is required that cannot be performed until after values have been assigned to the properties decorated with the `Parameter` attribute, such as extracting the individual paths.

This component responds to changes in the current URL by listening for the `LocationChanged` event defined by the `NavigationManager` class. The event's `Location` property provides the component with the current URL, which is used to alter the classes for the anchor element. Listing 35-16 applies the new component in the layout.

■ **Tip** Notice that I have removed the `Match` attribute in Listing 35-14. The new component supports this attribute but defaults to matching based on the number of paths that it receives through the `href` attribute.

Listing 35-16. Applying a New Component in the NavLayout.razor File in the Blazor Folder

```
@inherits LayoutComponentBase

<div class="container-fluid">
    <div class="row">
        <div class="col-3">
            @foreach (string key in NavLinks.Keys) {
                <MultiNavLink class="btn btn-outline-primary btn-block"
                         href="@NavLinks[key]" ActiveClass="btn-primary text-white">
                    @key
                </MultiNavLink>
            }
        </div>
        <div class="col">
            @Body
        </div>
    </div>
</div>

@code {

    public Dictionary<string, string[]> NavLinks
        = new Dictionary<string, string[]> {
            {"People", new string[] {"/people", "/" } },
            {"Departments", new string[] {"/depts", "/departments" } },
            {"Details", new string[] { "/person" } }
        };
}
```

Restart ASP.NET Core and request `http://localhost:5000` and `http://localhost:5000/departments`. Both URLs are recognized, and the corresponding navigation buttons are highlighted, as shown in Figure 35-7.

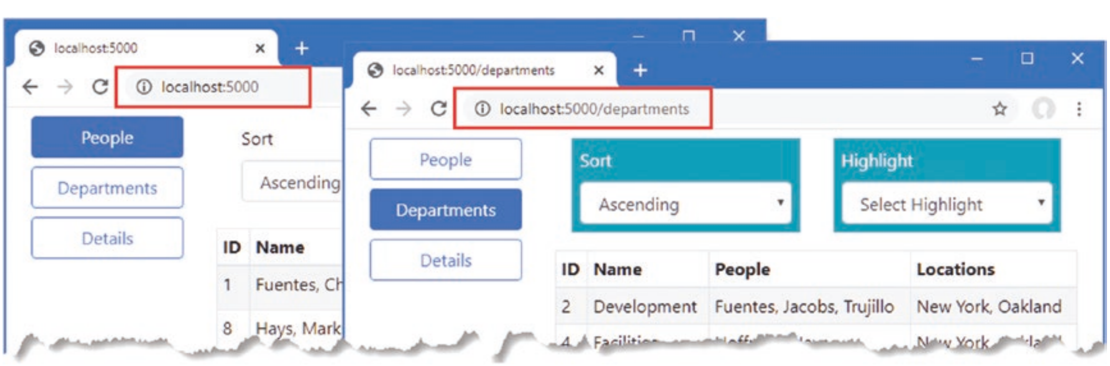

Figure 35-7. *Using the lifecycle methods*

Using the Lifecycle Methods for Asynchronous Tasks

The lifecycle methods are also useful for performing tasks that may complete after the initial content from the component has been rendered, such as querying the database. Listing 35-17 replaces the placeholder content in the PersonDisplay component and uses the lifecycle methods to query the database using values received as parameters.

Listing 35-17. Querying for Data in the PersonDisplay.razor File in the Blazor Folder

```
@page "/person"
@page "/person/{id:long}"

@if (Person == null) {
    <h5 class="bg-info text-white text-center p-2">Loading...</h5>
} else {
    <table class="table table-striped table-bordered">
        <tbody>
            <tr><th>Id</th><td>@Person.PersonId</td></tr>
            <tr><th>Surname</th><td>@Person.Surname</td></tr>
            <tr><th>Firstname</th><td>@Person.Firstname</td></tr>
        </tbody>
    </table>
}

<button class="btn btn-outline-primary" @onclick="@(() => HandleClick(false))">
    Previous
</button>
<button class="btn btn-outline-primary" @onclick="@(() => HandleClick(true))">
    Next
</button>

@code {

    [Inject]
    public DataContext Context { get; set; }

    [Inject]
    public NavigationManager NavManager { get; set; }

    [Parameter]
    public long Id { get; set; } = 0;

    public Person Person { get; set; }

    protected async override Task OnParametersSetAsync() {
        await Task.Delay(1000);
        Person = await Context.People
            .FirstOrDefaultAsync(p => p.PersonId == Id) ?? new Person();
    }

    public void HandleClick(bool increment) {
        Person = null;
        NavManager.NavigateTo($"/person/{(increment ? Id + 1 : Id -1)}");
    }
}
```

The component can't query the database until the parameter values have been set and so the value of the Person property is obtained in the OnParametersSetAsync method. Since the database is running alongside the ASP.NET Core server, I have added a one-second delay before querying the database to help emphasize the way the component works.

The value of the Person property is null until the query has completed, at which point it will be either an object representing the query result or a new Person object if the query doesn't produce a result. A loading message is displayed while the Person object is null.

Restart ASP.NET Core and request http://localhost:5000. Click one of the Edit buttons presented in the table, and the PersonDisplay component will display a summary of the data. Click the Previous and Next buttons to query for the objects with the adjacent primary key values, producing the results shown in Figure 35-8.

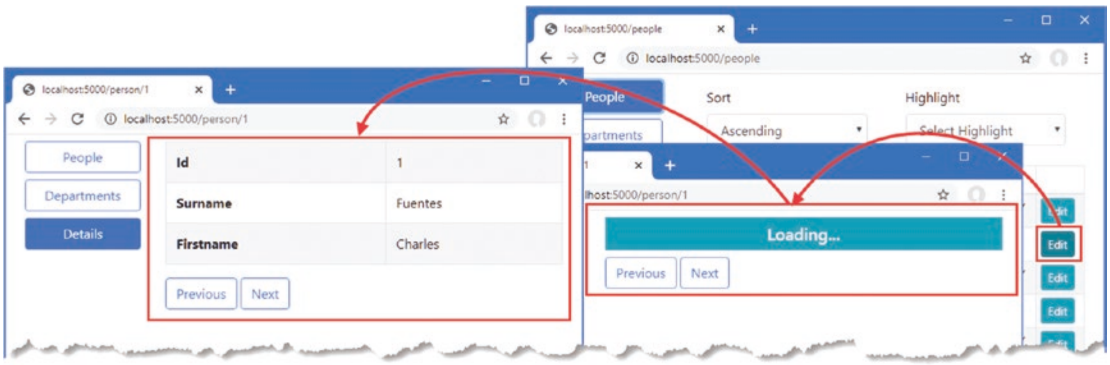

Figure 35-8. *Performing asynchronous tasks in a component*

Notice that Blazor doesn't wait for the Task performed in the OnParametersSetAsync method to complete before displaying content to the user, which is why a loading message is useful when the Person property is null. Once the Task is complete and a value has been assigned to the Person property, the component's view is automatically re-rendered, and the changes are sent to the browser over the persistent HTTP connection to be displayed to the user.

Managing Component Interaction

Most components work together through parameters and events, allowing the user's interaction to drive changes in the application. Blazor also provides advanced options for managing interaction with components, which I describe in the following sections.

Using References to Child Components

A parent component can obtain a reference to a child component and use it to consume the properties and methods it defines. In preparation, Listing 35-18 adds a disabled state to the MultiNavLink component.

Listing 35-18. Adding a Feature in the MultiNavLink.razor File in the Blazor Folder

```
<a class="@ComputedClass" @onclick="HandleClick"  href="">
    @if (Enabled) {
        @ChildContent
    } else {
        @("Disabled")
    }
</a>

@code {

    [Inject]
    public NavigationManager NavManager { get; set; }

    [Parameter]
    public IEnumerable<string> Href { get; set; }
```

```
    [Parameter]
    public string Class { get; set; }

    [Parameter]
    public string ActiveClass { get; set; }

    [Parameter]
    public string DisabledClasses { get; set; }

    [Parameter]
    public NavLinkMatch? Match { get; set; }

    public NavLinkMatch ComputedMatch { get =>
            Match ?? (Href.Count() == 1 ? NavLinkMatch.Prefix : NavLinkMatch.All); }

    [Parameter]
    public RenderFragment ChildContent { get; set; }

    public string ComputedClass { get; set; }

    public void HandleClick() {
        NavManager.NavigateTo(Href.First());
    }

    private void CheckMatch(string currentUrl) {
        string path = NavManager.ToBaseRelativePath(currentUrl);
        path = path.EndsWith("/") ? path.Substring(0, path.Length - 1) : path;
        bool match = Href.Any(href => ComputedMatch == NavLinkMatch.All
                ? path == href : path.StartsWith(href));
        if (!Enabled) {
            ComputedClass = DisabledClasses;
        } else {
            ComputedClass = match ? $"{Class} {ActiveClass}" : Class;
        }
    }

    protected override void OnParametersSet() {
        ComputedClass = Class;
        NavManager.LocationChanged += (sender, arg) => CheckMatch(arg.Location);
        Href = Href.Select(h => h.StartsWith("/") ? h.Substring(1) : h);
        CheckMatch(NavManager.Uri);
    }

    private bool Enabled { get; set; } = true;

    public void SetEnabled(bool enabled) {
        Enabled = enabled;
        CheckMatch(NavManager.Uri);
    }
}
```

In Listing 35-19, I have updated the shared layout so that it retains references to the `MultiNavLink` components and a `button` that toggles their `Enabled` property value.

Listing 35-19. Retaining References in the NavLayout.razor File in the Blazor Folder

```
@inherits LayoutComponentBase

<div class="container-fluid">
    <div class="row">
        <div class="col-3">
            @foreach (string key in NavLinks.Keys) {
                <MultiNavLink class="btn btn-outline-primary btn-block"
                         href="@NavLinks[key]"
                         ActiveClass="btn-primary text-white"
                         DisabledClasses="btn btn-dark text-light btn-block disabled"
                         @ref="Refs[key]">
                    @key
                </MultiNavLink>
            }
            <button class="btn btn-secondary btn-block mt-5 " @onclick="ToggleLinks">
                Toggle Links
            </button>
        </div>
        <div class="col">
            @Body
        </div>
    </div>
</div>

@code {

    public Dictionary<string, string[]> NavLinks
        = new Dictionary<string, string[]> {
            {"People", new string[] {"/people", "/" } },
            {"Departments", new string[] {"/depts", "/departments" } },
            {"Details", new string[] { "/person" } }
        };

    public Dictionary<string, MultiNavLink> Refs
        = new Dictionary<string, MultiNavLink>();

    private bool LinksEnabled = true;

    public void ToggleLinks() {
        LinksEnabled = !LinksEnabled;
        foreach (MultiNavLink link in Refs.Values) {
            link.SetEnabled(LinksEnabled);
        }
    }
}
```

References to components are created by adding an `@ref` attribute and specifying the name of a field or property to which the component should be assigned. Since the `MultiNavLink` components are created in a `@foreach` loop driven by a `Dictionary`, the simplest way to retain references is also in a `Dictionary`, like this:

```
...
<MultiNavLink class="btn btn-outline-primary btn-block"
    href="@NavLinks[key]" ActiveClass="btn-primary text-white"
    DisabledClasses="btn btn-dark text-light btn-block disabled"
    @ref="Refs[key]">
...
```

As each MultiNavLink component is created, it is added to the Refs dictionary. Razor Components are compiled into standard C# classes, which means that a collection of MultiNavLink components is a collection of MultiNavlink objects.

```
...
public Dictionary<string, MultiNavLink> Refs
        = new Dictionary<string, MultiNavLink>();
...
```

Restart ASP.NET Core, request http://localhost:5000, and click the Toggle Links button. The event handler invokes the ToggleLinks method, which sets the value of the Enabled property for each of the MultiNavLink components, as shown in Figure 35-9.

■ **Caution** References can be used only after the component's content has been rendered and the OnAfterRender/OnAfterRenderAsync lifecycle methods have been invoked. This makes references ideal for use in event handlers but not the earlier lifecycle methods.

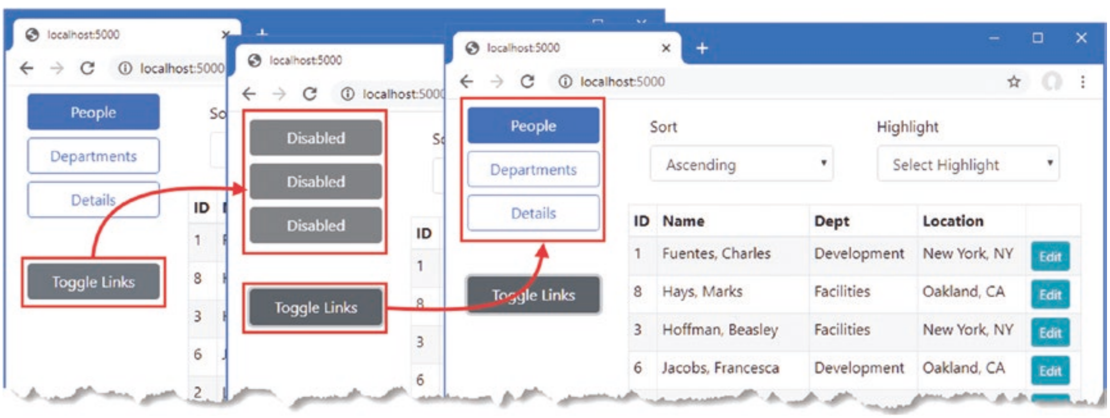

Figure 35-9. *Retaining references to components*

Interacting with Components from Other Code

Components can be used by other code in the ASP.NET Core application, allowing a richer interaction between parts of complex projects. Listing 35-20 alters the method in the MultiNavLink component so it can be invoked by other parts of the ASP.NET Core application to enable and disable navigation.

Listing 35-20. Replacing a Method in the MultiNavLink.razor File in the Blazor Folder

```
<a class="@ComputedClass" @onclick="HandleClick"  href="">
    @if (Enabled) {
        @ChildContent
    } else {
        @("Disabled")
    }
</a>

@code {

    // ...other properties and methods omitted for brevity...

    public void SetEnabled(bool enabled) {
        InvokeAsync(() => {
            Enabled = enabled;
            CheckMatch(NavManager.Uri);
```

```
            StateHasChanged();
        });
    }
}
```

Razor Components provide two methods that are used in code that is invoked outside of the Blazor environment, as described in Table 35-6.

Table 35-6. *The Razor Component External Invocation Methods*

Name	Description
`InvokeAsync(func)`	This method is used to execute a function inside the Blazor environment.
`StateHasChanged()`	This method is called when a change occurs outside of the normal lifecycle, as shown in the next section.

The `InvokeAsync` method is used to invoke a function within the Blazor environment, ensuring that changes are processed correctly. The `StateHasChanged` method is invoked when all the changes have been applied, triggering a Blazor update and ensuring changes are reflected in the component's output.

To create a service that will be available throughout the application, create the `Advanced/Services` folder and add to it a class file named `ToggleService.cs`, with the code shown in Listing 35-21.

Listing 35-21. The Contents of the ToggleService.cs File in the Services Folder

```
using Advanced.Blazor;
using System.Collections.Generic;

namespace Advanced.Services {
    public class ToggleService {
        private List<MultiNavLink> components = new List<MultiNavLink>();
        private bool enabled = true;

        public void EnrolComponents(IEnumerable<MultiNavLink> comps) {
            components.AddRange(comps);
        }

        public bool ToggleComponents() {
            enabled = !enabled;
            components.ForEach(c => c.SetEnabled(enabled));
            return enabled;
        }
    }
}
```

This service managed a collection of components and invokes the `SetEnabled` method on all of them when its `ToggleComponents` method is called. There is nothing specific to Blazor in this service, which relies on the C# classes that are produced when Razor Component files are compiled. Listing 35-22 updates the application configuration to configure the `ToggleService` class as a singleton service.

Listing 35-22. Configuring a Service in the Startup.cs File in the Advanced Folder

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddDbContext<DataContext>(opts => {
        opts.UseSqlServer(Configuration[
            "ConnectionStrings:PeopleConnection"]);
        opts.EnableSensitiveDataLogging(true);
    });
    services.AddControllersWithViews().AddRazorRuntimeCompilation();
```

```
    services.AddRazorPages().AddRazorRuntimeCompilation();
    services.AddServerSideBlazor();
    services.AddSingleton<Services.ToggleService>();
}
...
```

Listing 35-23 updates the Blazor layout so that references to the `MultiNavLink` components are retained and registered with the new service.

Listing 35-23. Using the Service in the NavLayout.razor File in the Blazor Folder

```
@inherits LayoutComponentBase
@using Advanced.Services

<div class="container-fluid">
    <div class="row">
        <div class="col-3">
            @foreach (string key in NavLinks.Keys) {
                <MultiNavLink class="btn btn-outline-primary btn-block"
                         href="@NavLinks[key]"
                         ActiveClass="btn-primary text-white"
                         DisabledClasses="btn btn-dark text-light btn-block disabled"
                         @ref="Refs[key]">
                    @key
                </MultiNavLink>
            }
            <button class="btn btn-secondary btn-block mt-5 " @onclick="ToggleLinks">
                Toggle Links
            </button>
        </div>
        <div class="col">
            @Body
        </div>
    </div>
</div>

@code {

    [Inject]
    public ToggleService Toggler { get; set; }

    public Dictionary<string, string[]> NavLinks
        = new Dictionary<string, string[]> {
            {"People", new string[] {"/people", "/" } },
            {"Departments", new string[] {"/depts", "/departments" } },
            {"Details", new string[] { "/person" } }
        };

    public Dictionary<string, MultiNavLink> Refs
        = new Dictionary<string, MultiNavLink>();

    protected override void OnAfterRender(bool firstRender) {
        if (firstRender) {
            Toggler.EnrolComponents(Refs.Values);
        }
    }
```

```
    public void ToggleLinks() {
        Toggler.ToggleComponents();
    }
}
```

As noted in the previous section, component references are not available until after the content has been rendered. Listing 35-23 uses the `OnAfterRender` lifecycle method to register the component references with the service, which is received via dependency injection.

The final step is to use the service from a different part of the ASP.NET Core application. Listing 35-24 adds a simple action method to the `Home` controller that invokes the `ToggleService.ToggleComponents` method every time it handles a request.

Listing 35-24. Adding an Action Method in the HomeController.cs File in the Controllers Folder

```
using Advanced.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using Advanced.Services;

namespace Advanced.Controllers {
    public class HomeController : Controller {
        private DataContext context;
        private ToggleService toggleService;

        public HomeController(DataContext dbContext, ToggleService ts) {
            context = dbContext;
            toggleService = ts;
        }

        public IActionResult Index([FromQuery] string selectedCity) {
            return View(new PeopleListViewModel {
                People = context.People
                    .Include(p => p.Department).Include(p => p.Location),
                Cities = context.Locations.Select(l => l.City).Distinct(),
                SelectedCity = selectedCity
            });
        }

        public string Toggle() => $"Enabled: { toggleService.ToggleComponents() }";
    }

    public class PeopleListViewModel {
        public IEnumerable<Person> People { get; set; }
        public IEnumerable<string> Cities { get; set; }
        public string SelectedCity { get; set; }

        public string GetClass(string city) =>
            SelectedCity == city ? "bg-info text-white" : "";
    }
}
```

Restart ASP.NET Core and request `http://localhost:5000`. Open a separate browser window and request `http://localhost:5000/controllers/home/toggle`. When the second request is processed by the ASP.NET Core application, the action method will use the service, which toggles the state of the navigation button. Each time you request `/controllers/home/toggle`, the state of the navigation buttons will change, as shown in Figure 35-10.

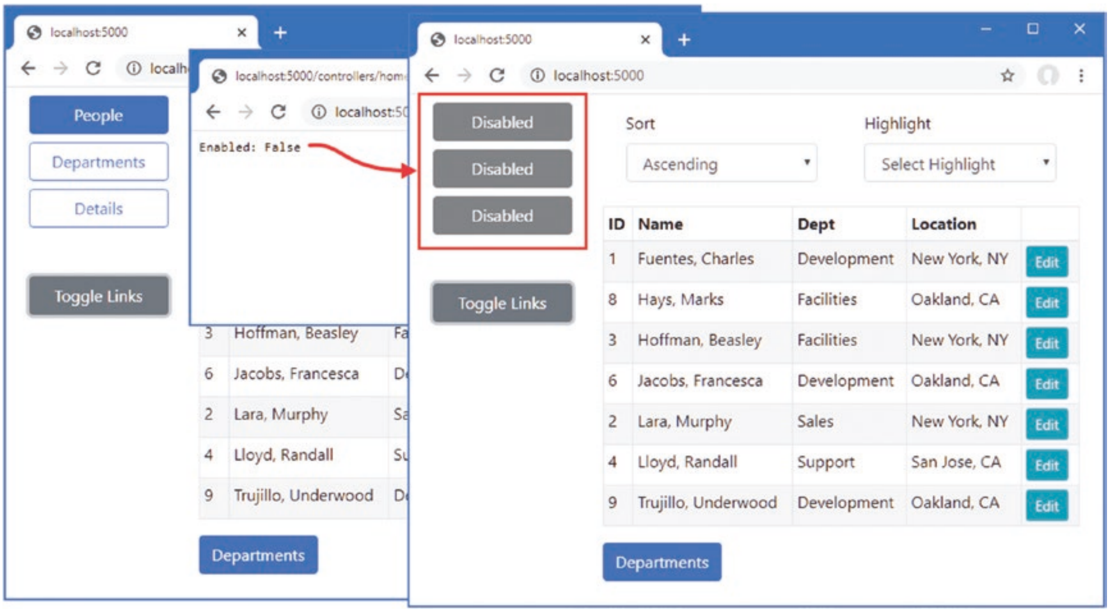

Figure 35-10. *Invoking component methods*

Interacting with Components Using JavaScript

Blazor provides a range of tools for interaction between JavaScript and server-side C# code, as described in the following sections.

Invoking a JavaScript Function from a Component

To prepare for these examples, add a JavaScript file named `interop.js` to the `wwwroot` folder and add the code shown in Listing 35-25.

Listing 35-25. The Contents of the interop.js File in the wwwroot Folder

```
function addTableRows(colCount) {
    let elem = document.querySelector("tbody");
    let row = document.createElement("tr");
    elem.append(row);
    for (let i = 0; i < colCount; i++) {
        let cell = document.createElement("td");
        cell.innerText = "New Elements"
        row.append(cell);
    }
}
```

The JavaScript code uses the API provided by the browser to locate a `tbody` element, which denotes the body of a table and adds a new row containing the number of cells specified by the function parameter.

To incorporate the JavaScript file into the application, add the element shown in Listing 35-26 to the `_Host` Razor Page, which was configured as the fallback page that delivers the Blazor application to the browser.

Listing 35-26. Adding an Element in the _Host.cshtml File in the Pages Folder

```
@page "/"
@{ Layout = null; }

<!DOCTYPE html>
<html>
```

```
<head>
    <title>@ViewBag.Title</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
    <base href="~/" />
</head>
<body>
    <div class="m-2">
        <component type="typeof(Advanced.Blazor.Routed)" render-mode="Server" />
    </div>
    <script src="_framework/blazor.server.js"></script>
    <script src="~/interop.js"></script>
</body>
</html>
```

Listing 35-27 revises the PersonDisplay component so that it renders a button that invokes the JavaScript function when the onclick event is triggered. I have also removed the delay that I added earlier to demonstrate the use of the component lifecycle methods.

Listing 35-27. Invoking a JavaScript Function in the PersonDisplay.razor File in the Blazor Folder

```
@page "/person"
@page "/person/{id:long}"

@if (Person == null) {
    <h5 class="bg-info text-white text-center p-2">Loading...</h5>
} else {
    <table class="table table-striped table-bordered">
        <tbody>
            <tr><th>Id</th><td>@Person.PersonId</td></tr>
            <tr><th>Surname</th><td>@Person.Surname</td></tr>
            <tr><th>Firstname</th><td>@Person.Firstname</td></tr>
        </tbody>
    </table>
}

<button class="btn btn-outline-primary" @onclick="@HandleClick">
    Invoke JS Function
</button>

@code {

    [Inject]
    public DataContext Context { get; set; }

    [Inject]
    public NavigationManager NavManager { get; set; }

    [Inject]
    public IJSRuntime JSRuntime { get; set; }

    [Parameter]
    public long Id { get; set; } = 0;

    public Person Person { get; set; }

    protected async override Task OnParametersSetAsync() {
        //await Task.Delay(1000);
        Person = await Context.People
            .FirstOrDefaultAsync(p => p.PersonId == Id) ?? new Person();
    }
```

```
    public async Task HandleClick() {
        await JSRuntime.InvokeVoidAsync("addTableRows", 2);
    }
}
```

Invoking a JavaScript function is done through the `IJSRuntime` interface, which components receive through dependency injection. The service is created automatically as part of the Blazor configuration and provides the methods described in Table 35-7.

Table 35-7. The IJSRuntime Methods

Name	Description
`InvokeAsync<T>(name, args)`	This method invokes the specified function with the arguments provided. The result type is specified by the generic type parameter.
`InvokeVoidAsync(name, args)`	This method invokes a function that doesn't produce a result.

In Listing 35-27, I use the `InvokeVoidAsync` method to invoke the `addTableRows` JavaScript function, providing a value for the function parameter. Restart ASP.NET Core, navigate to `http://localhost:5000/person/1`, and click the Invoke JS Function button. Blazor will invoke the JavaScript function, which adds a row to the end of the table, as shown in Figure 35-11.

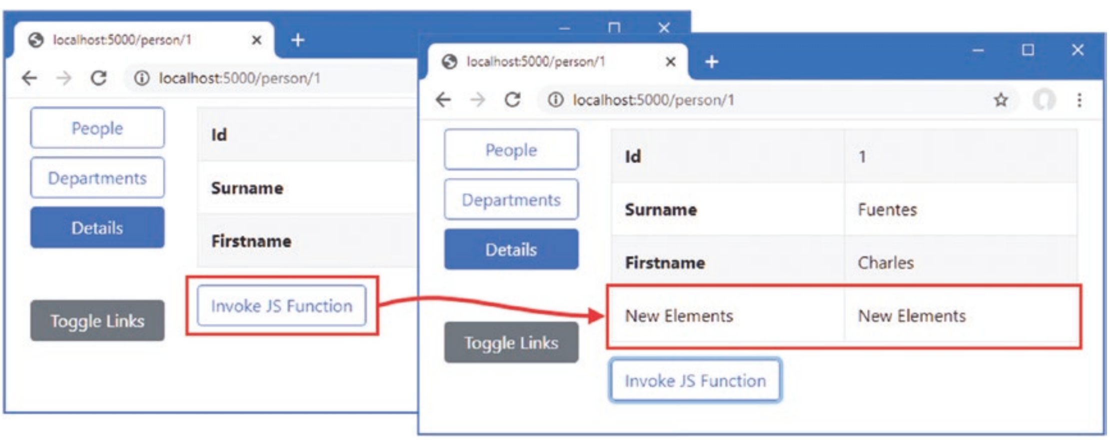

Figure 35-11. Invoking a JavaScript function

Retaining References to HTML Elements

Razor Components can retain references to the HTML elements they create and pass those references to JavaScript code. Listing 35-28 changes the JavaScript function from the previous example so that it operates on an HTML element it receives through a parameter.

Listing 35-28. Defining a Parameter in the interop.js File in the wwwroot Folder

```
function addTableRows(colCount, elem) {
    //let elem = document.querySelector("tbody");
    let row = document.createElement("tr");
    elem.parentNode.insertBefore(row, elem);
    for (let i = 0; i < colCount; i++) {
        let cell = document.createElement("td");
        cell.innerText = "New Elements"
        row.append(cell);
    }
}
```

In Listing 35-29, the `PersonDisplay` component retains a reference to one of the HTML elements it creates and passes it as an argument to the JavaScript function.

Listing 35-29. Retaining a Reference in the PersonDisplay.razor File in the Blazor Folder

```
@page "/person"
@page "/person/{id:long}"

@if (Person == null) {
    <h5 class="bg-info text-white text-center p-2">Loading...</h5>
} else {
    <table class="table table-striped table-bordered">
        <tbody>
            <tr><th>Id</th><td>@Person.PersonId</td></tr>
            <tr @ref="RowReference"><th>Surname</th><td>@Person.Surname</td></tr>
            <tr><th>Firstname</th><td>@Person.Firstname</td></tr>
        </tbody>
    </table>
}

<button class="btn btn-outline-primary" @onclick="@HandleClick">
    Invoke JS Function
</button>

@code {

    [Inject]
    public DataContext Context { get; set; }

    [Inject]
    public NavigationManager NavManager { get; set; }

    [Inject]
    public IJSRuntime JSRuntime { get; set; }

    [Parameter]
    public long Id { get; set; } = 0;

    public Person Person { get; set; }

    protected async override Task OnParametersSetAsync() {
        //await Task.Delay(1000);
        Person = await Context.People
            .FirstOrDefaultAsync(p => p.PersonId == Id) ?? new Person();
    }

    public ElementReference RowReference { get; set; }

    public async Task HandleClick() {
        await JSRuntime.InvokeVoidAsync("addTableRows", 2, RowReference);
    }
}
```

The `@ref` attribute assigns the HTML element to a property, whose type must be `ElementReference`. Restart ASP.NET Core, request `http://localhost:5000/person/1`, and click the Invoke JS Function button. The value of the ElementReference property is passed as an argument to the JavaScript function through the `InvokeVoidAsync` method, producing the result shown in Figure 35-12.

■ **Note** The only use for a reference to a regular HTML element is to pass it to a JavaScript function. Use the binding and event features described in earlier chapters to interact with the elements rendered by a component.

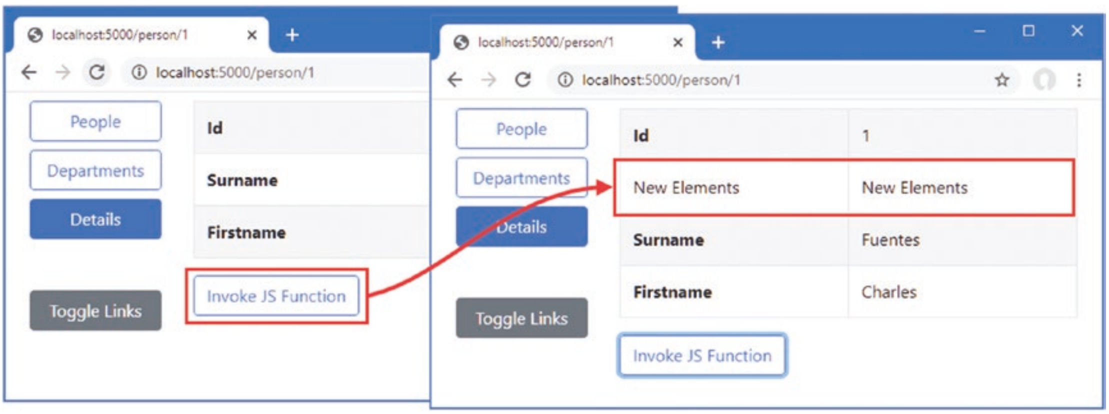

Figure 35-12. *Retaining a reference to an HTML element*

Invoking a Component Method from JavaScript

The basic approach for invoking a C# method from JavaScript is to use a `static` method. Listing 35-30 adds a static method to the `MultiNavLink` component that changes the enabled state.

Listing 35-30. Introducing Static Members in the MultiNavLink.razor File in the Blazor Folder

```
<a class="@ComputedClass" @onclick="HandleClick"  href="">
    @if (Enabled) {
        @ChildContent
    } else {
        @("Disabled")
    }
</a>

@code {

    // ...other methods and properties omitted for brevity...

    [JSInvokable]
    public static void ToggleEnabled() => ToggleEvent.Invoke(null, new EventArgs());

    private static event EventHandler ToggleEvent;

    protected override void OnInitialized() {
        ToggleEvent += (sender, args) => SetEnabled(!Enabled);
    }
}
```

Static methods must be decorated with the `JSInvokable` attribute before they can be invoked from JavaScript code. The main limitation of using `static` methods is that it makes it difficult to update individual components, so I have defined a `static` event that each instance of the component will handle. The event is named `ToggleEvent`, and it is triggered by the static method that will be called from JavaScript. To listen for the event, I have used the `OnInitialized` lifecycle event. When the event is received, the enabled state of the component is toggled through the instance method `SetEnabled`, which uses the `InvokeAsync` and `StateHasChanged` methods required when a change is made outside of Blazor.

Listing 35-31 adds a function to the JavaScript file that creates a button element that invokes the static C# method when it is clicked.

Listing 35-31. Adding a Function in the interop.js File in the wwwroot Folder

```
function addTableRows(colCount, elem) {
    //let elem = document.querySelector("tbody");
    let row = document.createElement("tr");
    elem.parentNode.insertBefore(row, elem);
    for (let i = 0; i < colCount; i++) {
        let cell = document.createElement("td");
        cell.innerText = "New Elements"
        row.append(cell);
    }
}

function createToggleButton() {
    let sibling = document.querySelector("button:last-of-type");
    let button = document.createElement("button");
    button.classList.add("btn", "btn-secondary", "btn-block");
    button.innerText = "JS Toggle";
    sibling.parentNode.insertBefore(button, sibling.nextSibling);
    button.onclick = () => DotNet.invokeMethodAsync("Advanced", "ToggleEnabled");
}
```

The new function locates one of the existing `button` elements and adds a new button after it. When the button is clicked, the component method is invoked, like this:

```
...
button.onclick = () => DotNet.invokeMethodAsync("Advanced", "ToggleEnabled");
...
```

It is important to pay close attention to the capitalization of the JavaScript function used to C# methods: it is `DotNet`, followed by a period, followed by `invokeMethodAsync`, with a lowercase `i`. The arguments are the name of the assembly and the name of the static method. (The name of the component is not required.)

The `button` element that the function in Listing 35-31 looks for isn't available until after Blazor has rendered content for the user. For this reason, Listing 35-32 adds a statement to the `OnAfterRenderAsync` method defined by the `NavLayout` component to invoke the JavaScript function only when the content has been rendered. (The `NavLayout` component is the parent to the `MultiNavLink` components that will be affected when the `static` method is invoked and allows me to ensure the JavaScript function is invoked only once.)

Listing 35-32. Invoking a JavaScript Function in the NavLayout.razor File in the Blazor Folder

```
...
@code {

    [Inject]
    public IJSRuntime JSRuntime { get; set; }

    [Inject]
    public ToggleService Toggler { get; set; }

    public Dictionary<string, string[]> NavLinks
        = new Dictionary<string, string[]> {
            {"People", new string[] {"/people", "/" } },
            {"Departments", new string[] {"/depts", "/departments" } },
            {"Details", new string[] { "/person" } }
        };
```

```
    public Dictionary<string, MultiNavLink> Refs
        = new Dictionary<string, MultiNavLink>();

    protected async override Task OnAfterRenderAsync(bool firstRender) {
        if (firstRender) {
            Toggler.EnrolComponents(Refs.Values);
            await JSRuntime.InvokeVoidAsync("createToggleButton");
        }
    }

    public void ToggleLinks() {
        Toggler.ToggleComponents();
    }
}
...
```

Restart ASP.NET Core and request `http://localhost:5000`. Once Blazor has rendered its content, the JavaScript function will be called and creates a new button. Clicking the button invokes the `static` method, which triggers the event that toggles the state of the navigation buttons and causes a Blazor update, as shown in Figure 35-13.

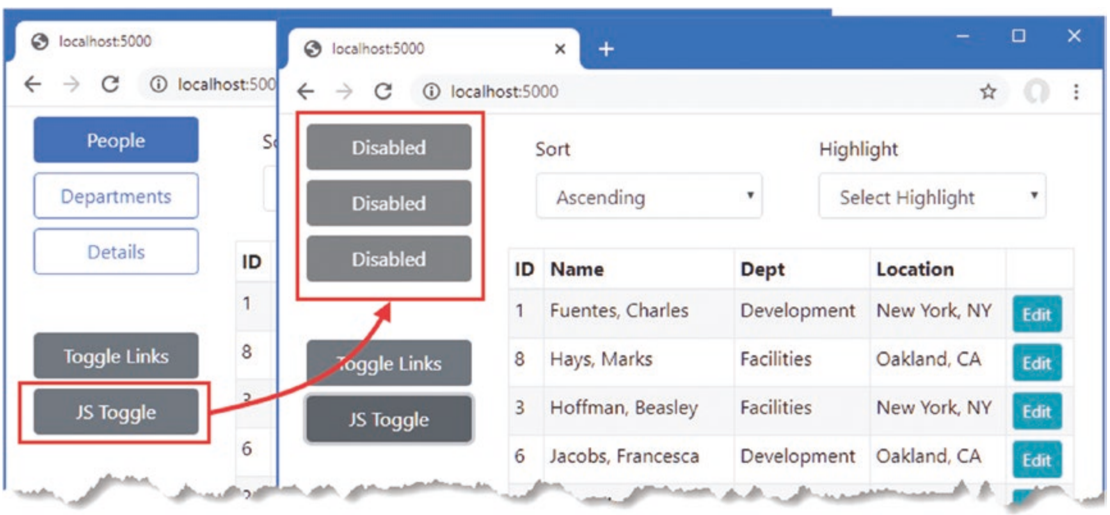

Figure 35-13. *Invoking a component method from JavaScript*

Invoking an Instance Method from a JavaScript Function

Part of the complexity in the previous example comes from responding to a `static` method to update the Razor Component objects. An alternative approach is to provide the JavaScript code with a reference to an instance method, which it can then invoke directly.

The first step is to add the `JSInvokable` attribute to the method that the JavaScript code will invoke. I am going to invoke the `ToggleComponents` methods defined by the `ToggleService` class, as shown in Listing 35-33.

Listing 35-33. Applying an Attribute in the ToggleService.cs File in the Services Folder

```
using Advanced.Blazor;
using System.Collections.Generic;
using Microsoft.JSInterop;

namespace Advanced.Services {
    public class ToggleService {
        private List<MultiNavLink> components = new List<MultiNavLink>();
        private bool enabled = true;
```

```
        public void EnrolComponents(IEnumerable<MultiNavLink> comps) {
            components.AddRange(comps);
        }

        [JSInvokable]
        public bool ToggleComponents() {
            enabled = !enabled;
            components.ForEach(c => c.SetEnabled(enabled));
            return enabled;
        }
    }
}
```

The next step is to provide the JavaScript function with a reference to the object whose method will be invoked, as shown in Listing 35-34.

Listing 35-34. Providing an Instance in the NavLayout.razor File in the Blazor Folder

```
...
protected async override Task OnAfterRenderAsync(bool firstRender) {
    if (firstRender) {
        Toggler.EnrolComponents(Refs.Values);
        await JSRuntime.InvokeVoidAsync("createToggleButton",
            DotNetObjectReference.Create(Toggler));
    }
}
...
```

The `DotNetObjectReference.Create` method creates a reference to an object, which is passed to the JavaScript function as an argument using the `JSRuntime.InvokeVoidAsync` method. The final step is to receive the object reference in JavaScript and invoke its method when the button element is clicked, as shown in Listing 35-35.

Listing 35-35. Invoking a C# Method in the interop.js File in the wwwroot Folder

```
function addTableRows(colCount, elem) {
    //let elem = document.querySelector("tbody");
    let row = document.createElement("tr");
    elem.parentNode.insertBefore(row, elem);
    for (let i = 0; i < colCount; i++) {
        let cell = document.createElement("td");
        cell.innerText = "New Elements"
        row.append(cell);
    }
}

function createToggleButton(toggleServiceRef) {
    let sibling = document.querySelector("button:last-of-type");
    let button = document.createElement("button");
    button.classList.add("btn", "btn-secondary", "btn-block");
    button.innerText = "JS Toggle";
    sibling.parentNode.insertBefore(button, sibling.nextSibling);
    button.onclick = () => toggleServiceRef.invokeMethodAsync("ToggleComponents");
}
```

The JavaScript function receives the reference to the C# object as a parameter and invokes its methods using `invokeMethodAsync`, specifying the name of the method as the argument. (Arguments to the method can also be provided but are not required in this example.)

Restart ASP.NET Core, request `http://localhost:5000`, and click the JS Toggle button. The result is the same as shown in Figure 35-13, but the change in the components is managed through the `ToggleService` object.

Summary

In this chapter, I explained how components can be combined with routing to alter the content displayed to the user based on the current URL. I described the component lifecycle and the methods it can implement for each stage in the process, and I finished this chapter by explaining the different ways that component methods can be invoked from outside of Blazor, including interoperability with JavaScript. In the next chapter, I describe the features that Blazor provides for HTML forms.

CHAPTER 36

Blazor Forms and Data

In this chapter, I describe the features that Blazor provides for dealing with HTML forms, including support for data validation. I describe the built-in components that Blazor provides and show you how they are used. In this chapter, I also explain how the Blazor model can cause unexpected results with Entity Framework Core and show you how to address these issues. I finish the chapter by creating a simple form application for creating, reading, updating, and deleting data (the CRUD operations) and explain how to extend the Blazor form features to improve the user's experience. Table 36-1 puts the Blazor form features in context.

Table 36-1. *Putting Blazor Form Features in Context*

Question	Answer
What are they?	Blazor provides a set of built-in components that present the user with a form that can be easily validated.
Why are they useful?	Forms remain one of the core building blocks of web applications, and these components provide functionality that will be required in most projects.
How are they used?	The `EditForm` component is used as a parent for individual form field components.
Are there any pitfalls or limitations?	There can be issues with the way that Entity Framework Core and Blazor work together, and these become especially apparent when using forms.
Are there any alternatives?	You could create your own form components and validation features, although the features described in this chapter are suitable for most projects and, as I demonstrate, can be easily extended.

Table 36-2 summarizes the chapter.

Table 36-2. *Chapter Summary*

Problem	Solution	Listing
Creating an HTML form	Use the `EditForm` and `Input*` components	7–9, 13
Validating data	Use the standard validation attributes and the events emitted by the `EditForm` component	10–12
Discarding unsaved data	Explicitly release the data or create new scopes for components	14–16
Avoiding repeatedly querying the database	Manage query execution explicitly	17–19

Preparing for This Chapter

This chapter uses the Advanced project from Chapter 35. To prepare for this chapter, create the `Blazor/Forms` folder and add to it a Razor Component named `EmptyLayout.razor` with the content shown in Listing 36-1. I will use this component as the main layout for this chapter.

A. Freeman, *Pro ASP.NET Core 3*, https://doi.org/10.1007/978-1-4842-5440-0_36

■ **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-3`. See Chapter 1 for how to get help if you have problems running the examples.

Listing 36-1. The Contents of the EmptyLayout.razor File in the Blazor/Forms Folder

```
@inherits LayoutComponentBase
<div class="m-2">
    @Body
</div>
```

Add a `RazorComponent` named `FormSpy.razor` to the `Blazor/Forms` folder with the content shown in Listing 36-2. This is a component I will use to display form elements alongside the values that are being edited.

Listing 36-2. The Contents of the FormSpy.razor File in the Blazor/Forms Folder

```
<div class="container-fluid no-gutters">
    <div class="row">
        <div class="col">
            @ChildContent
        </div>
        <div class="col">
            <table class="table table-sm table-striped table-bordered">
                <thead>
                    <tr><th colspan="2" class="text-center">Data Summary</th></tr>
                </thead>
                <tbody>
                    <tr><th>ID</th><td>@PersonData?.PersonId</td></tr>
                    <tr><th>Firstname</th><td>@PersonData?.Firstname</td></tr>
                    <tr><th>Surname</th><td>@PersonData?.Surname</td></tr>
                    <tr><th>Dept ID</th><td>@PersonData?.DepartmentId</td></tr>
                    <tr><th>Location ID</th><td>@PersonData?.LocationId</td></tr>
                </tbody>
            </table>
        </div>
    </div>
</div>

@code {

    [Parameter]
    public RenderFragment ChildContent { get; set; }

    [Parameter]
    public Person PersonData { get; set; }
}
```

Next, add a component named `Editor.razor` to the `Blazor/Forms` folder and add the content shown in Listing 36-3. This component will be used to edit existing `Person` objects and to create new ones.

■ **Caution** Do not use the `Editor` and `List` components in real projects until you have read the rest of the chapter. I have included common pitfalls that I explain later in the chapter.

Listing 36-3. The Contents of the Editor.razor File in the Blazor/Forms Folder

```
@page "/forms/edit/{id:long}"
@layout EmptyLayout

<h4 class="bg-primary text-center text-white p-2">Edit</h4>

<FormSpy PersonData="PersonData">
    <h4 class="text-center">Form Placeholder</h4>
    <div class="text-center">
        <NavLink class="btn btn-secondary" href="/forms">Back</NavLink>
    </div>
</FormSpy>

@code {

    [Inject]
    public NavigationManager NavManager { get; set; }

    [Inject]
    DataContext Context { get; set; }

    [Parameter]
    public long Id { get; set; }

    public Person PersonData { get; set; } = new Person();

    protected async override Task OnParametersSetAsync() {
        PersonData = await Context.People.FindAsync(Id);
    }
}
```

The component in Listing 36-3 uses an `@layout` expression to override the default layout and select `EmptyLayout`. The side-by-side layout is used to present the `PersonTable` component alongside a placeholder, which is where I will add a form.

Finally, create a component named `List.razor` in the `Blazor/Forms` folder and add the content shown in Listing 36-4 to define a component that will present the user with a list of `Person` objects, presented as a table.

Listing 36-4. The Contents of the List.razor File in the Blazor/Forms Folder

```
@page "/forms"
@page "/forms/list"
@layout EmptyLayout

<h5 class="bg-primary text-white text-center p-2">People</h5>

<table class="table table-sm table-striped table-bordered">
    <thead>
        <tr>
            <th>ID</th><th>Name</th><th>Dept</th><th>Location</th><th></th>
        </tr>
    </thead>
    <tbody>
        @if  (People.Count() == 0) {
            <tr><th colspan="5" class="p-4 text-center">Loading Data...</th></tr>
        } else {
            @foreach (Person p in People) {
                <tr>
```

```
                    <td>@p.PersonId</td>
                    <td>@p.Surname, @p.Firstname</td>
                    <td>@p.Department.Name</td>
                    <td>@p.Location.City</td>
                    <td>
                        <NavLink class="btn btn-sm btn-warning"
                             href="@GetEditUrl(p.PersonId)">
                            Edit
                        </NavLink>
                    </td>
                </tr>
            }
        }
    </tbody>
</table>

@code {

    [Inject]
    public DataContext Context { get; set; }

    public IEnumerable<Person> People { get; set; } = Enumerable.Empty<Person>();

    protected override void OnInitialized() {
        People = Context.People.Include(p => p.Department).Include(p => p.Location);
    }

    string GetEditUrl(long id) => $"/forms/edit/{id}";
}
```

Dropping the Database and Running the Application

Open a new PowerShell command prompt, navigate to the folder that contains the `Advanced.csproj` file, and run the command shown in Listing 36-5 to drop the database.

Listing 36-5. Dropping the Database

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

Select Start Without Debugging or Run Without Debugging from the Debug menu or use the PowerShell command prompt to run the command shown in Listing 36-6.

Listing 36-6. Running the Example Application

```
dotnet run
```

Use a browser to request `http://localhost:5000/forms`, which will produce a data table. Click one of the Edit buttons, and you will see a placeholder for the form and a summary showing the current property values of the selected `Person` object, as shown in Figure 36-1.

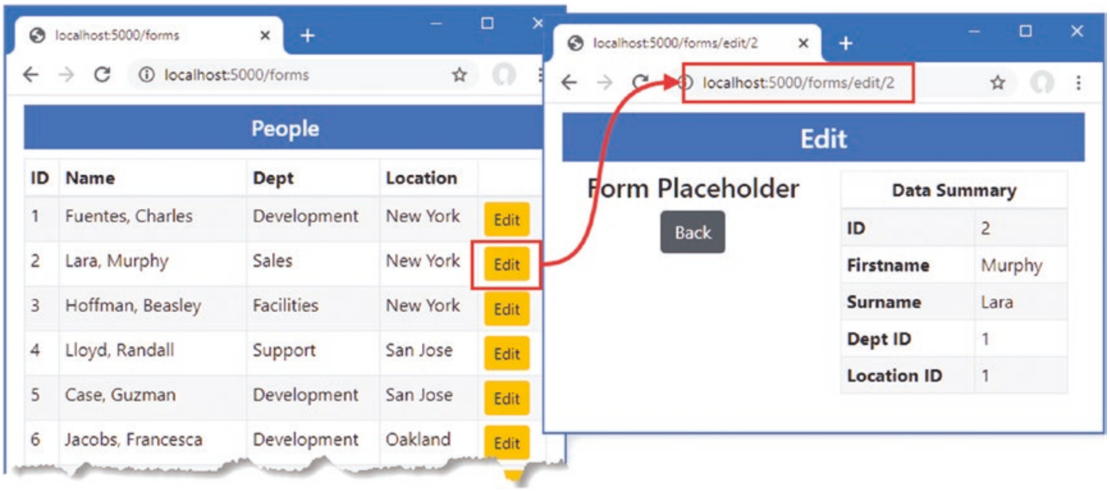

Figure 36-1. *Running the example application*

Using the Blazor Form Components

Blazor provides a set of built-in components that are used to render form elements, ensuring that the server-side component properties are updated after user interaction and integrating validation. Table 36-3 describes the components that Blazor provides.

Table 36-3. *The Bazor Form Components*

Name	Description
`EditForm`	This component renders a `form` element that is wired up for data validation.
`InputText`	This component renders an `input` element that is bound to a C# `string` property.
`InputCheckbox`	This component renders an input element whose `type` attribute is `checkbox` and that is bound to a C# `bool` property.
`InputDate`	This component renders an input element those `type` attribute is `date` and that is bound to a C# `DateTime` or `DateTimeOffset` property.
`InputNumber`	This component renders an input element those `type` attribute is `number` and that is bound to a C# `int`, `long`, `float`, `double`, or `decimal` value.
`InputTextArea`	This component renders a `textarea` component that is bound to a C# string property.

The `EditForm` component must be used for any of the other components to work. In Listing 36-7, I have added an `EditForm`, along with `InputText` components that represent two of the properties defined by the `Person` class.

Listing 36-7. Using Form Components in the Editor.razor File in the Blazor/Forms Folder

```
@page "/forms/edit/{id:long}"
@layout EmptyLayout

<h4 class="bg-primary text-center text-white p-2">Edit</h4>

<FormSpy PersonData="PersonData">
    <EditForm Model="PersonData">
        <div class="form-group">
            <label>Person ID</label>
            <InputNumber class="form-control"
                @bind-Value="PersonData.PersonId" disabled />
        </div>
```

```
        <div class="form-group">
            <label>Firstname</label>
            <InputText class="form-control" @bind-Value="PersonData.Firstname" />
        </div>
        <div class="form-group">
            <label>Surname</label>
            <InputText class="form-control" @bind-Value="PersonData.Surname" />
        </div>
        <div class="form-group">
            <label>Dept ID</label>
            <InputNumber class="form-control"
                @bind-Value="PersonData.DepartmentId" />
        </div>
        <div class="text-center">
            <NavLink class="btn btn-secondary" href="/forms">Back</NavLink>
       </div>
    </EditForm>
</FormSpy>

@code {

    // ...statements omitted for brevity...
}
```

The `EditForm` component renders a `form` element and provides the foundation for the validation features described in the "Validating Form Data" section. The `Model` attribute is used to provide the `EditForm` with the object that the form is used to edit and validate.

The components in Table 36-3 whose names begin with `Input` are used to display an `input` or `textarea` element for a single model property. These components define a custom binding named `Value` that is associated with the model property using the `@bind-Value` attribute. The property-level components must be matched to the type of the property they present to the user. It is for this reason that I have used the `InputText` component for the `Firstname` and `Surname` properties of the `Person` class, while the `InputNumber` component is used for the `PersonId` and `DepartmentId` properties. If you use a property-level component with a model property of the wrong type, you will receive an error when the component attempts to parse a value entered into the HTML element.

Restart ASP.NET Core and request `http://localhost:5000/forms/edit/2`, and you will see the three `input` elements displayed. Edit the values and move the focus by pressing the Tab key, and you will see the summary data on the right of the window update, as shown in Figure 36-2. The built-in form components support attribute splatting, which is why the `disabled` attribute applied to the `InputNumber` component for the `PersonId` property has been applied to the `input` element.

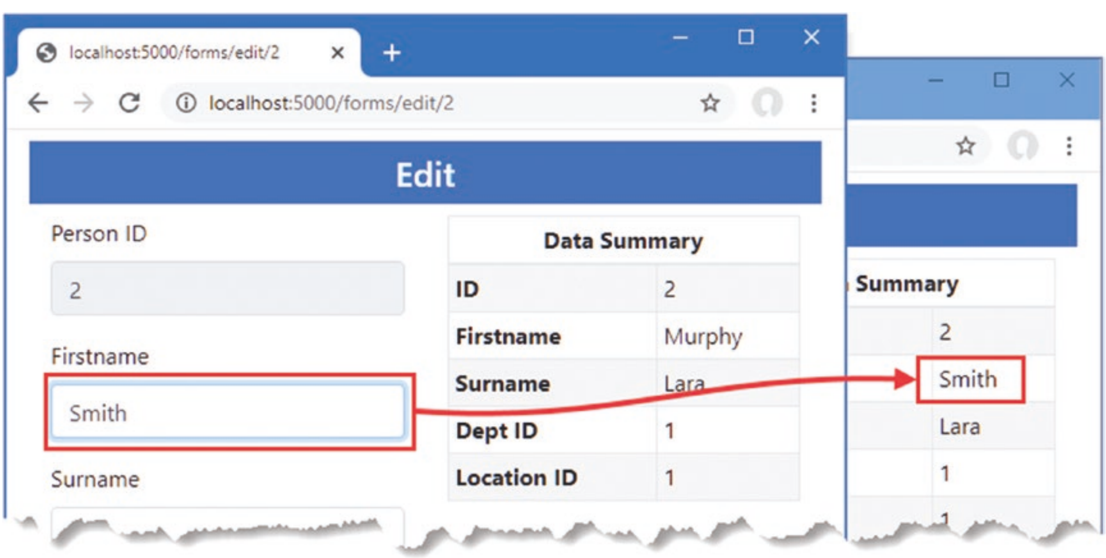

***Figure 36-2.** Using the Blazor form elements*

Creating Custom Form Components

Blazor provides built-in components for only `input` and `textarea` elements. Fortunately, creating a custom component that integrates into the Blazor form features is a simple process. Add a Razor Component named `CustomSelect.razor` to the `Blazor/Forms` folder and use it to define the component shown in Listing 36-8.

Listing 36-8. The Contents of the CustomSelect.razor File in the Blazor/Forms Folder

```
@typeparam TValue
@inherits InputBase<TValue>

<select class="form-control @CssClass" value="@CurrentValueAsString"
            @onchange="@(ev => CurrentValueAsString = ev.Value as string)">
        @ChildContent
        @foreach (KeyValuePair<string, TValue> kvp in Values) {
            <option value="@kvp.Value">@kvp.Key</option>
        }
</select>

@code {

    [Parameter]
    public RenderFragment ChildContent { get; set; }

    [Parameter]
    public IDictionary<string, TValue> Values { get; set; }

    [Parameter]
    public Func<string, TValue> Parser { get; set; }

    protected override bool TryParseValueFromString(string value, out TValue result,
            out string validationErrorMessage) {
        try {
            result = Parser(value);
            validationErrorMessage = null;
            return true;
        } catch {
            result = default(TValue);
            validationErrorMessage = "The value is not valid";
            return false;
        }
    }
}
```

The base class for form components is `InputBase<TValue>`, where the generic type argument is the model property type the component represents. The base class takes care of most of the work and provides the `CurrentValueAsString` property, which is used to provide the current value in event handlers when the user selects a new value, like this:

```
...
<select class="form-control @CssClass" value="@CurrentValueAsString"
            @onchange="@(ev => CurrentValueAsString = ev.Value as string)">
...
```

In preparation for data validation, which I describe in the next section, this component includes the value of the `CssClass` property in the `select` element's `class` attribute, like this:

```
...
<select class="form-control @CssClass" value="@CurrentValueAsString"
            @onchange="@(ev => CurrentValueAsString = ev.Value as string)">
...
```

The abstract TryParseValueFromString method has to be implemented so that the base class is able to map between string values used by HTML elements and the corresponding value for the C# model property. I don't want to implement my custom select element to any specific C# data type, so I have used an @typeparam expression to define a generic type parameter. The Values property is used to receive a dictionary mapping string values that will be displayed to the user and TValue values that will be used as C# values. The method receives two out parameters that are used to set the parsed value and a parser validation error message that will be displayed to the user if there is a problem. Since I am working with generic types, the Parser property receives a function that is invoked to parse a string value into a TValue value.

Listing 36-9 applies the new form component so the user can select values for the DepartmentId and LocationId properties defined by the Person class.

Listing 36-9. Using a Custom Form Element in the Editor.razor File in the Blazor/Forms Folder

```
@page "/forms/edit/{id:long}"
@layout EmptyLayout

<h4 class="bg-primary text-center text-white p-2">Edit</h4>

<FormSpy PersonData="PersonData">

    <EditForm Model="PersonData">
        <div class="form-group">
            <label>Firstname</label>
            <InputText class="form-control" @bind-Value="PersonData.Firstname" />
        </div>
        <div class="form-group">
            <label>Surname</label>
            <InputText class="form-control" @bind-Value="PersonData.Surname" />
        </div>
        <div class="form-group">
            <label>Dept ID</label>
            <CustomSelect TValue="long" Values="Departments"
                          Parser="@(str => long.Parse(str))"
                          @bind-Value="PersonData.DepartmentId">
                <option selected disabled value="0">Choose a Department</option>
            </CustomSelect>
        </div>
        <div class="form-group">
            <label>Location ID</label>
            <CustomSelect TValue="long" Values="Locations"
                          Parser="@(str => long.Parse(str))"
                          @bind-Value="PersonData.LocationId">
                <option selected disabled value="0">Choose a Location</option>
            </CustomSelect>
        </div>
        <div class="text-center">
            <NavLink class="btn btn-secondary" href="/forms">Back</NavLink>
        </div>
    </EditForm>
</FormSpy>

@code {

    [Inject]
    public NavigationManager NavManager { get; set; }

    [Inject]
    DataContext Context { get; set; }
```

```
    [Parameter]
    public long Id { get; set; }

    public Person PersonData { get; set; } = new Person();

    public IDictionary<string, long> Departments { get; set; }
        = new Dictionary<string, long>();
    public IDictionary<string, long> Locations { get; set; }
        = new Dictionary<string, long>();

    protected async override Task OnParametersSetAsync() {
        PersonData = await Context.People.FindAsync(Id);
        Departments = await Context.Departments
            .ToDictionaryAsync(d => d.Name, d => d.Departmentid);
        Locations = await Context.Locations
            .ToDictionaryAsync(l => $"{l.City}, {l.State}", l => l.LocationId);
    }
}
```

I use the Entity Framework Core `ToDictionaryAsync` method to create collections of values and labels from the `Department` and `Location` data and use them to configure the `CustomSelect` components. Restart ASP.NET Core and request `http://localhost:5000/forms/edit/2`; you will see the `select` elements shown in Figure 36-3. When you pick a new value, the `CustomSelect` component will update the `CurrentValueAsString` property, which will result in a call to the `TryParseValueFromString` method, with the result used to update the `Value` binding.

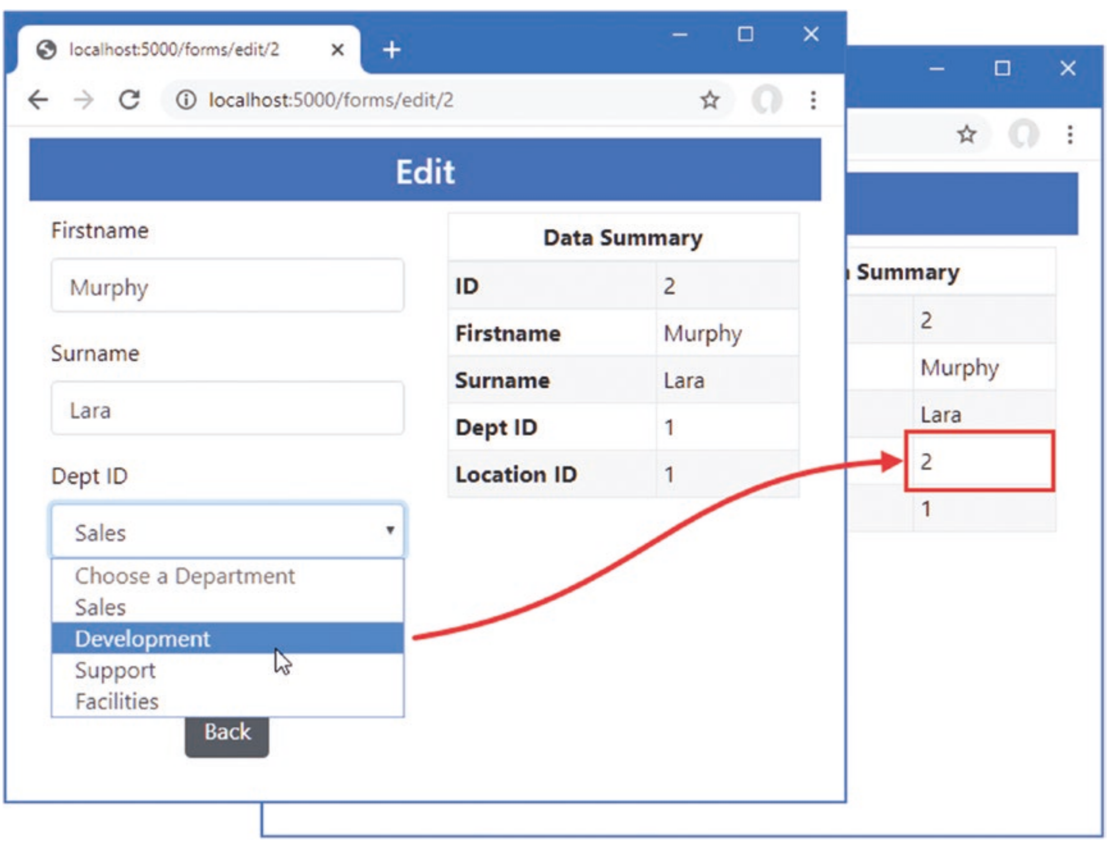

***Figure 36-3.** Using a custom form element*

Validating Form Data

Blazor provides components that perform validation using the standard attributes. Table 36-4 describes the validation components.

Table 36-4. *The Blazor Validation Components*

Name	Description
`DataAnnotationsValidator`	This component integrates the validation attributes applied to the model class into the Blazor form features.
`ValidationMessage`	This component displays validation error messages for a single property.
`ValidationSummary`	This component displays validation error messages for the entire model object.

The validation components generate elements assigned to classes, described in Table 36-5, which can be styled with CSS to draw the user's attention.

Table 36-5. *The Classes Used by the Blazor Validation Components*

Name	Description
`validation-errors`	The `ValidationSummary` component generates a `ul` element that is assigned to this class and is the top-level container for the summary of validation messages.
`validation-message`	The `ValidationSummary` component populates its `ul` element with `li` elements assigned to this class for each validation message. The `ValidationMessage` component renders a `div` element assigned to this class for its property-level messages.

The Blazor `Input*` components add the HTML elements they generate to the classes described in Table 36-6 to indicate validation status. This includes the `InputBase<TValue>` class from which I derived the `CustomSelect` component and is the purpose of the `CssClass` property in Listing 36-8.

Table 36-6. *The Validation Classes Added to Form Elements*

Name	Description
`modified`	Elements are added to this class once the user has edited the value.
`valid`	Elements are added to this class if the value they contain passes validation.
`invalid`	Elements are added to this class if the value they contain fails validation.

This combination of components and classes can be confusing at first, but the key is to start by defining the CSS styles you require based on the classes in Tables 36-5 and 36-6. Add a CSS Stylesheet named `blazorValidation.css` to the `wwwroot` folder with the content shown in Listing 36-10.

Listing 36-10. The Contents of the blazorValidation.css File in the wwwroot Folder

```
.validation-errors {
    background-color: rgb(220, 53, 69); color: white; padding: 8px;
    text-align: center; font-size: 16px; font-weight: 500;
}
div.validation-message { color: rgb(220, 53, 69); font-weight: 500 }
.modified.valid { border: solid 3px rgb(40, 167, 69); }
.modified.invalid { border: solid 3px rgb(220, 53, 69); }
```

These styles format error messages in red and apply a red or green border to individual form elements. Listing 36-11 imports the CSS stylesheet and applies the Blazor validation components.

Listing 36-11. Applying Validation Components in the Editor.razor File in the Blazor/Forms Folder

```
@page "/forms/edit/{id:long}"
@layout EmptyLayout

<link href="/blazorValidation.css" rel="stylesheet" />
<h4 class="bg-primary text-center text-white p-2">Edit</h4>

<FormSpy PersonData="PersonData">
    <EditForm Model="PersonData">
        <DataAnnotationsValidator />
        <ValidationSummary />
        <div class="form-group">
            <label>Firstname</label>
            <ValidationMessage For="@(() => PersonData.Firstname)" />
            <InputText class="form-control" @bind-Value="PersonData.Firstname" />
        </div>
        <div class="form-group">
            <label>Surname</label>
            <ValidationMessage For="@(() => PersonData.Surname)" />
            <InputText class="form-control" @bind-Value="PersonData.Surname" />
        </div>
        <div class="form-group">
            <label>Dept ID</label>
            <ValidationMessage For="@(() => PersonData.DepartmentId)" />
            <CustomSelect TValue="long" Values="Departments"
                          Parser="@(str => long.Parse(str))"
                          @bind-Value="PersonData.DepartmentId">
                <option selected disabled value="0">Choose a Department</option>
            </CustomSelect>
        </div>
        <div class="form-group">
            <label>Location ID</label>
            <ValidationMessage For="@(() => PersonData.LocationId)" />
            <CustomSelect TValue="long" Values="Locations"
                          Parser="@(str => long.Parse(str))"
                          @bind-Value="PersonData.LocationId">
                <option selected disabled value="0">Choose a Location</option>
            </CustomSelect>
        </div>
        <div class="text-center">
            <NavLink class="btn btn-secondary" href="/forms">Back</NavLink>
        </div>
    </EditForm>
</FormSpy>

@code {

    // ...statements omitted for brevity...
}
```

The `DataAnnotationsValidator` and `ValidationSummary` components are applied without any configuration attributes. The `ValidationMessage` attribute is configured using the `For` attribute, which receives a function that returns the property the component represents. For example, here is the expression that selects the `Firstname` property:

```
..s.
<ValidationMessage For="@(() => PersonData.Firstname)" />
...
```

The expression defines no parameters and selects the property from the object used for the `Model` attribute of the `EditForm` component and not the model type. For this example, this means the expression operates on the `PersonData` object and not the `Person` class.

■ **Tip** Blazor isn't always able to determine the type of the property for the `ValidationMessage` component. If you receive an exception, then you can add a `TValue` attribute to set the type explicitly. For example, if the type of the property the `ValidationMessage` represents is `long`, then add a `TValue="long"` attribute.

The final step for enabling data validation is to apply attributes to the model class, as shown in Listing 36-12.

Listing 36-12. Applying Validation Attributes in the Person.cs File in the Models Folder

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace Advanced.Models {

    public class Person {

        public long PersonId { get; set; }

        [Required(ErrorMessage = "A firstname is required")]
        [MinLength(3, ErrorMessage = "Firstnames must be 3 or more characters")]
        public string Firstname { get; set; }

        [Required(ErrorMessage = "A surname is required")]
        [MinLength(3, ErrorMessage = "Surnames must be 3 or more characters")]
        public string Surname { get; set; }

        [Required]
        [Range(1, long.MaxValue,
            ErrorMessage = "A department must be selected")]
        public long DepartmentId { get; set; }

        [Required]
        [Range(1, long.MaxValue,
            ErrorMessage = "A location must be selected")]
        public long LocationId { get; set; }

        public Department Department { get; set; }
        public Location Location { get; set; }
    }
}
```

To see the effect of the validation components, restart ASP.NET Core and request `http://localhost:5000/forms/edit/2`. Clear the Firstname field and move the focus by pressing the Tab key or clicking on another field. As the focus changes, validation is performed, and error messages will be displayed. The `Editor` component shows both summary and per-property messages, so you will see the same error message shown twice. Delete all but the first two characters from the Surname field, and a second validation message will be displayed when you change the focus, as shown in Figure 36-4. (There is validation support for the other properties, too, but the `select` element doesn't allow the user to select an invalid valid. If you change a value, the `select` element will be decorated with a green border to indicate a valid selection, but you won't be able to see an invalid response until I demonstrate how the form components can be used to create new data objects.)

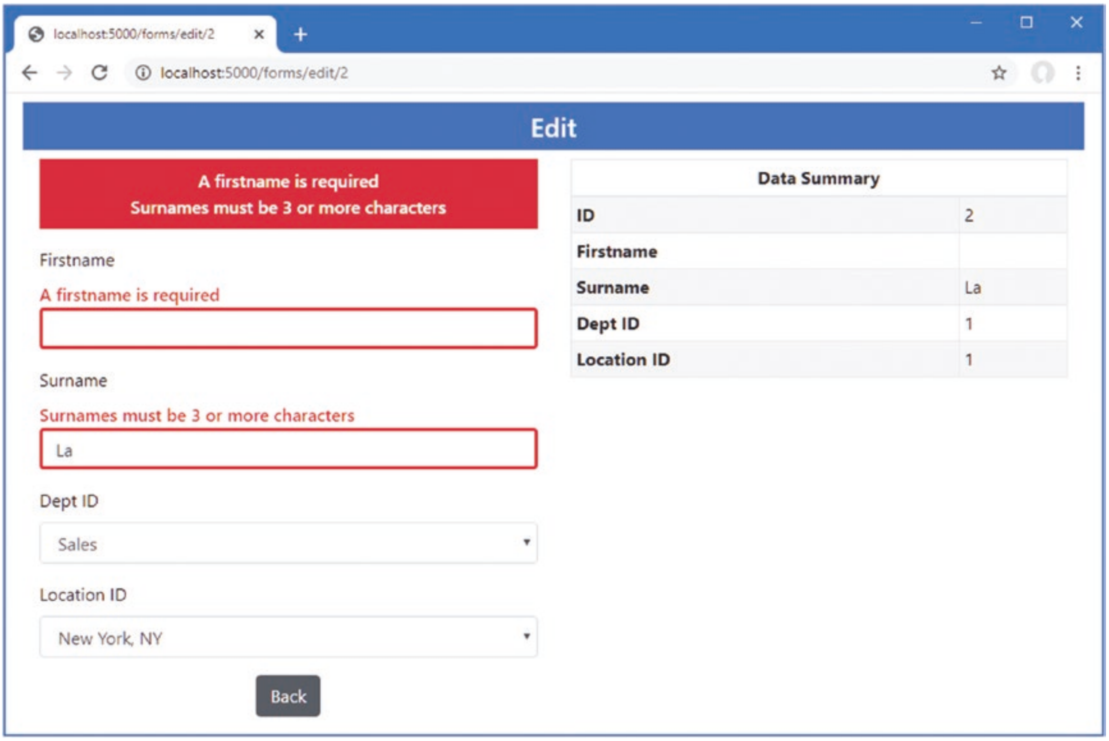

Figure 36-4. *Using the Blazor validation features*

Handling Form Events

The EditForm component defines events that allow an application to respond to user action, as described in Table 36-7.

Table 36-7. *The EditForm Events*

Name	Description
OnValidSubmit	This event is triggered when the form is submitted and the form data passes validation.
OnInvalidSubmit	This event is triggered when the form is submitted and the form data fails validation.
OnSubmit	This event is triggered when the form is submitted and before validation is performed.

These events are triggered by adding a conventional submit button within the content contained by the EditForm component. The EditForm component handles the onsubmit event sent by the form element it renders, applies validation, and triggers the events described in the table. Listing 36-13 adds a submit button to the Editor component and handles the EditForm events.

Listing 36-13. Handling EditForm Events in the Editor.razor File in the Blazor/Forms Folder

```
@page "/forms/edit/{id:long}"
@layout EmptyLayout

<link href="/blazorValidation.css" rel="stylesheet" />

<h4 class="bg-primary text-center text-white p-2">Edit</h4>
<h6 class="bg-info text-center text-white p-2">@FormSubmitMessage</h6>

<FormSpy PersonData="PersonData">
```

```
    <EditForm Model="PersonData" OnValidSubmit="HandleValidSubmit"
            OnInvalidSubmit="HandleInvalidSubmit">
        <DataAnnotationsValidator />
        <ValidationSummary />
        <div class="form-group">
            <label>Firstname</label>
            <ValidationMessage For="@(() => PersonData.Firstname)" />
            <InputText class="form-control" @bind-Value="PersonData.Firstname" />
        </div>
        <div class="form-group">
            <label>Surname</label>
            <ValidationMessage For="@(() => PersonData.Surname)" />
            <InputText class="form-control" @bind-Value="PersonData.Surname" />
        </div>
        <div class="form-group">
            <label>Dept ID</label>
            <ValidationMessage For="@(() => PersonData.DepartmentId)" />
            <CustomSelect TValue="long" Values="Departments"
                          Parser="@(str => long.Parse(str))"
                          @bind-Value="PersonData.DepartmentId">
                <option selected disabled value="0">Choose a Department</option>
            </CustomSelect>
        </div>
        <div class="form-group">
            <label>Location ID</label>
            <ValidationMessage For="@(() => PersonData.LocationId)" />
            <CustomSelect TValue="long" Values="Locations"
                          Parser="@(str => long.Parse(str))"
                          @bind-Value="PersonData.LocationId">
                <option selected disabled value="0">Choose a Location</option>
            </CustomSelect>
        </div>
        <div class="text-center">
            <button type="submit" class="btn btn-primary">Submit</button>
            <NavLink class="btn btn-secondary" href="/forms">Back</NavLink>
        </div>
    </EditForm>
</FormSpy>

@code {

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

    public string FormSubmitMessage { get; set; } = "Form Data Not Submitted";
    public void HandleValidSubmit() => FormSubmitMessage = "Valid Data Submitted";
    public void HandleInvalidSubmit() =>
        FormSubmitMessage = "Invalid Data Submitted";
}
```

Restart ASP.NET Core and request `http://localhost:5000/forms/edit/2`. Clear the Firstname field, and click the Submit button. In addition to the validation error, you will see a message indicating that the form was submitted with invalid data. Enter a name into the field and click Submit again, and the message will change, as shown in Figure 36-5.

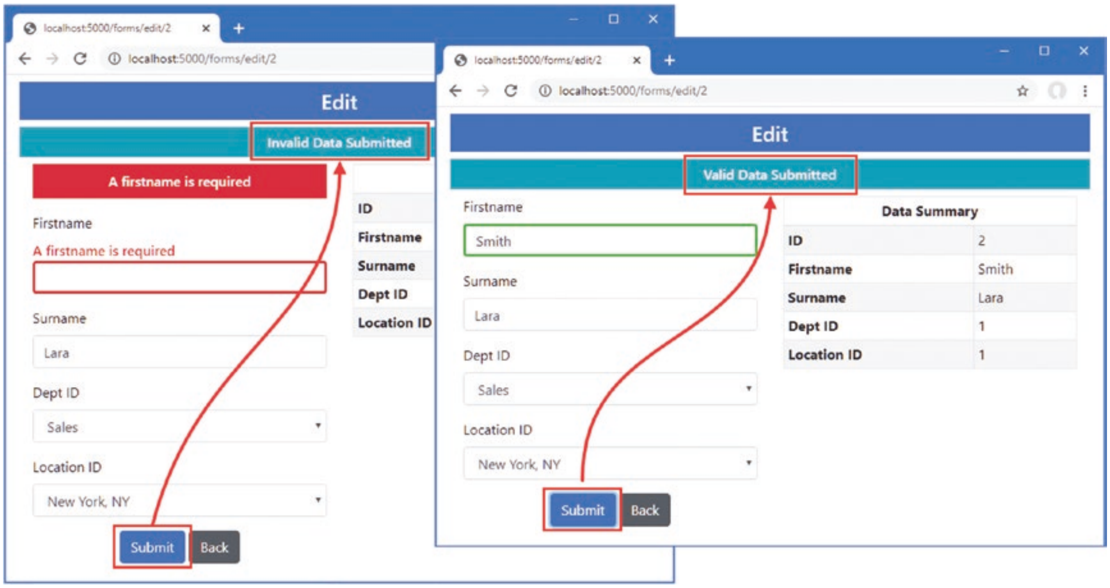

Figure 36-5. *Handling EditForm events*

Using Entity Framework Core with Blazor

The Blazor model changes the way that Entity Framework Core behaves, which can lead to unexpected results if you are used to writing conventional ASP.NET Core applications. In the sections that follow, I explain the issues and how to avoid the problems that can arise.

Understanding the Entity Framework Core Context Scope Issue

To see the first issue, request `http://localhost:5000/forms/edit/2`, clear the Firstname field, and change the contents of the Surname field to **La**. Neither of these values passes validation, and you will see error messages as you move between the form elements. Click the Back button, and you will see that the data table reflects the changes you made, as shown in Figure 36-6, even though they were not valid.

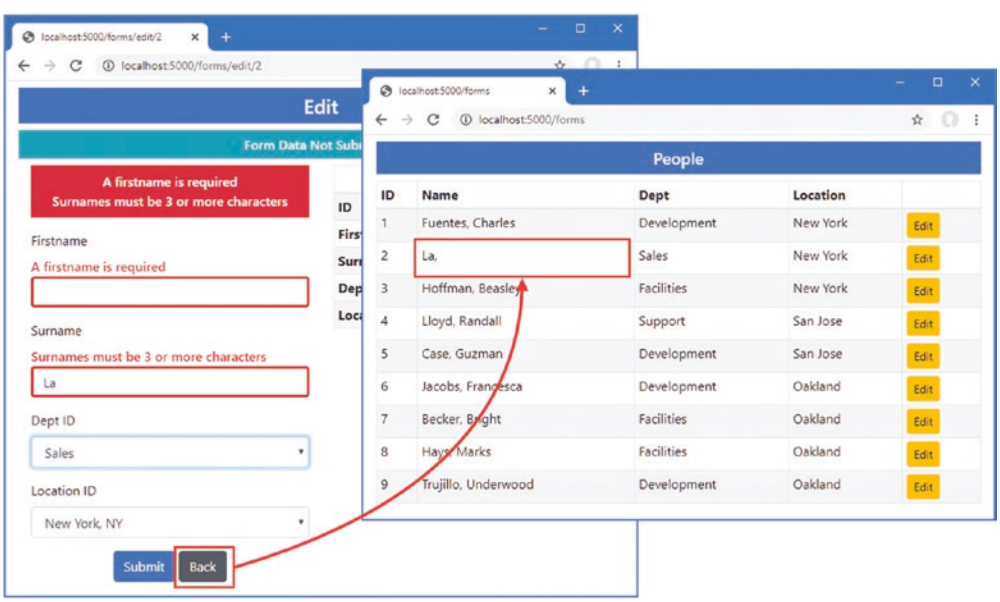

Figure 36-6. *The effect of editing data*

In a conventional ASP.NET Core application, written using controllers or Razor Pages, clicking a button triggers a new HTTP request. Each request is handled in isolation, and each request receives its own Entity Framework Core context object, which is configured as a scoped service. The result is that the data created when handling one request affects other requests only once it has been written to the database.

In a Blazor application, the routing system responds to URL changes without sending new HTTP requests, which means that multiple components are displayed using only the persistent HTTP connection that Blazor maintains to the server. This results in a single dependency injection scope being shared by multiple components, as shown in Figure 36-7, and the changes made by one component will affect other components even if the changes are not written to the database.

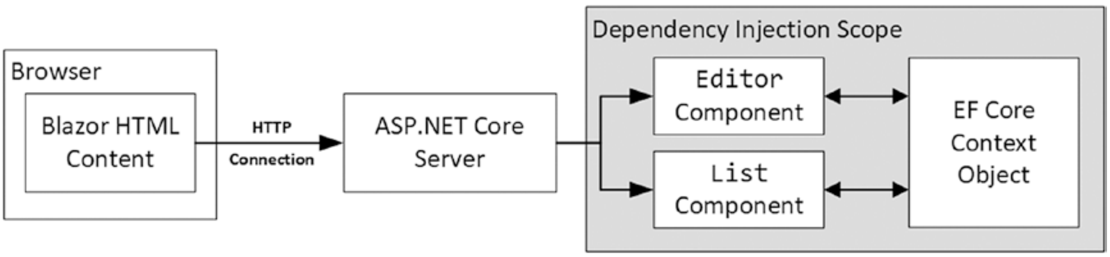

Figure 36-7. *The use of an Entity Framework Core context in a Blazor application*

Entity Framework Core is trying to be helpful, and this approach allows complex data operations to be performed over time before being stored (or discarded). Unfortunately, much like the helpful approach Entity Framework Core takes to dealing with related data, which I described in Chapter 35, it presents a pitfall for the unwary developer who expects components to handle data like the rest of ASP.NET Core.

Discarding Unsaved Data Changes

If sharing a context between components is appealing, which it will be for some applications, then you can embrace the approach and ensure that components discard any changes when they are destroyed, as shown in Listing 36-14.

Listing 36-14. Discarding Unsaved Data Changes in the Editor.razor File in the Blazor/Forms Folder

```
@page "/forms/edit/{id:long}"
@layout EmptyLayout
@implements IDisposable

<!-- ...elements omitted for brevity... -->

@code {

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

    public string FormSubmitMessage { get; set; } = "Form Data Not Submitted";
    public void HandleValidSubmit() => FormSubmitMessage = "Valid Data Submitted";
    public void HandleInvalidSubmit() =>
        FormSubmitMessage = "Invalid Data Submitted";

    public void Dispose() => Context.Entry(PersonData).State = EntityState.Detached;
}
```

As I noted in Chapter 35, components can implement the `System.IDisposable` interface, and the `Dispose` method will be invoked when the component is about to be destroyed, which happens when navigation to another component occurs. In Listing 36-14, the implementation of the `Dispose` method tells Entity Framework Core to disregard the `PersonData` object, which means it won't be used to satisfy future requests. To see the effect, restart ASP.NET Core, request `http://localhost:5000/forms/edit/2`, clear the Firstname field, and click the Back button. The modified `Person` object is disregarded when Entity Framework Core provides the `List` component with its data, as shown in Figure 36-8.

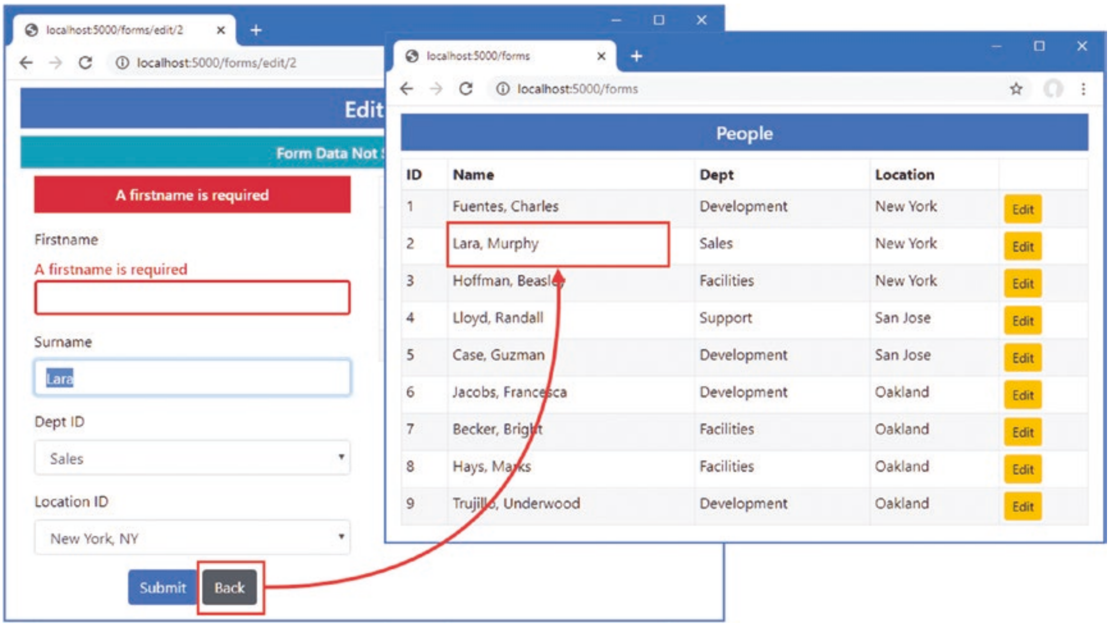

Figure 36-8. *Discarding data objects*

Creating New Dependency Injection Scopes

You must create new dependency injection scopes if you want to preserve the model used by the rest of ASP.NET Core and have each component receive its own Entity Framework Core context object. This is done by using the `@inherits` expression to set the base class for the component to `OwningComponentBase` or `OwningComponentBase<T>`.

The `OwningComponentCase` class defines a `ScopedServices` property that is inherited by the component and that provides an `IServiceProvider` object that can be used to obtain services that are created in a scope that is specific to the component's lifecycle and will not be shared with any other component, as shown in Listing 36-15.

Listing 36-15. Using a New Scope in the Editor.razor File in the Blazor/Forms Folder

```
@page "/forms/edit/{id:long}"
@layout EmptyLayout
@inherits OwningComponentBase
@using Microsoft.Extensions.DependencyInjection

<link href="/blazorValidation.css" rel="stylesheet" />

<h4 class="bg-primary text-center text-white p-2">Edit</h4>
<h6 class="bg-info text-center text-white p-2">@FormSubmitMessage</h6>

<!-- ...elements omitted for brevity... -->

@code {

    [Inject]
    public NavigationManager NavManager { get; set; }

    //[Inject]
    DataContext Context => ScopedServices.GetService<DataContext>();

    [Parameter]
    public long Id { get; set; }

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

```
    //public void Dispose() =>
    //      Context.Entry(PersonData).State = EntityState.Detached;
}
```

In the listing, I commented out the `Inject` attribute and set the value of the `Context` property by obtaining a `DataContext` service. The `Microsoft.Extensions.DependencyInjection` namespace contains extension methods that make it easier to obtain services from an `IServiceProvider` object, as described in Chapter 14.

■ **Note** Changing the base class doesn't affect services that are received using the `Inject` attribute, which will still be obtained within the request scope. Each service that you require in the dedicated component's scope must be obtained through the `ScopedServices` property, and the `Inject` attribute should not be applied to that property.

The `OwningComponentBase<T>` class defines an additional convenience property that provides access to a scoped service of type `T` and that can be useful if a component requires only a single scoped service, as shown in Listing 36-16 (although further services can still be obtained through the `ScopedServices` property).

Listing 36-16. Using the Typed Base Class in the Editor.razor File in the Blazor/Forms Folder

```
@page "/forms/edit/{id:long}"
@layout EmptyLayout
@inherits OwningComponentBase<DataContext>

<link href="/blazorValidation.css" rel="stylesheet" />

<h4 class="bg-primary text-center text-white p-2">Edit</h4>
<h6 class="bg-info text-center text-white p-2">@FormSubmitMessage</h6>

<!-- ...elements omitted for brevity... -->

@code {

    [Inject]
    public NavigationManager NavManager { get; set; }

    //[Inject]
    DataContext Context => Service;

    // ...statements omitted for brevity...
}
```

The scoped service is available through a property named `Service`. In this example, I specified `DataContext` as the type argument for the base class.

Regardless of which base class is used, the result is that the `Editor` component has its own dependency injection scope and its own `DataContext` object. The `List` component has not been modified, so it will receive the request-scoped `DataContext` object, as shown in Figure 36-9.

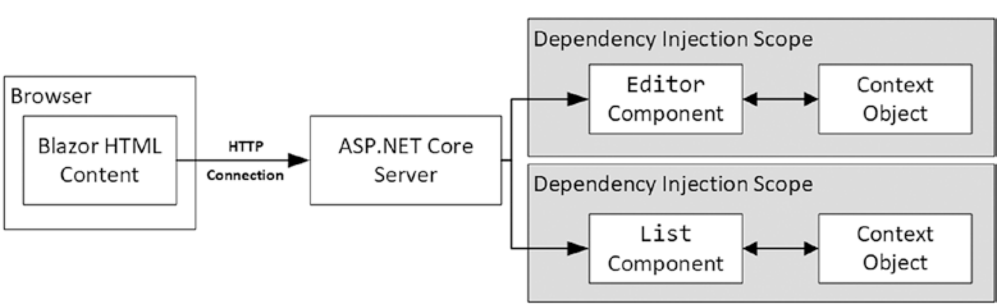

Figure 36-9. *Using scoped services for components*

Restart ASP.NET Core, navigate to `http://localhost:5000/forms/edit/2`, clear the Firstname field, and click the Back button. The changes made by the Editor component are not saved to the database, and since the `Editor` component's data context is separate from the one used by the `List` component, the edited data is discarded, producing the same response as shown in Figure 36-8.

Understanding the Repeated Query Issue

Blazor responds to changes in state as efficiently as possible but still has to render a component's content to determine the changes that should be sent to the browser.

One consequence of the way that Blazor works is that it can lead to a sharp increase in the number of queries sent to the database. To demonstrate the issue, Listing 36-17 adds a button that increments a counter to the `List` component.

Listing 36-17. Adding a Button in the List.razor File in the Blazor/Forms Folder

```
@page "/forms"
@page "/forms/list"
@layout EmptyLayout

<h5 class="bg-primary text-white text-center p-2">People</h5>

<table class="table table-sm table-striped table-bordered">
    <thead>
        <tr>
            <th>ID</th><th>Name</th><th>Dept</th><th>Location</th><th></th>
        </tr>
    </thead>
    <tbody>
        @if  (People.Count() == 0) {
            <tr><th colspan="5" class="p-4 text-center">Loading Data...</th></tr>
        } else {
            @foreach (Person p in People) {
                <tr>
                    <td>@p.PersonId</td>
                    <td>@p.Surname, @p.Firstname</td>
                    <td>@p.Department.Name</td>
                    <td>@p.Location.City</td>
                    <td>
                        <NavLink class="btn btn-sm btn-warning"
                               href="@GetEditUrl(p.PersonId)">
                            Edit
                        </NavLink>
                    </td>
                </tr>
            }
        }
    </tbody>
</table>

<button class="btn btn-primary" @onclick="@(() => Counter++)">Increment</button>
<span class="h5">Counter: @Counter</span>

@code {

    [Inject]
    public DataContext Context { get; set; }

    public IEnumerable<Person> People { get; set; } = Enumerable.Empty<Person>();
```

```
    protected override void OnInitialized() {
        People = Context.People.Include(p => p.Department).Include(p => p.Location);
    }

    string GetEditUrl(long id) => $"/forms/edit/{id}";

    public int Counter { get; set; } = 0;
}
```

Restart ASP.NET Core and request `http://localhost:5000/forms`. Click the button and watch the output from the ASP.NET Core server. Each time you click the button, the event handler is invoked, and a new database query is sent to the database, producing logging messages like these:

```
...
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (1ms) [Parameters=[], CommandType='Text',
      CommandTimeout='30']
  SELECT [p].[PersonId], [p].[DepartmentId], [p].[Firstname], [p].[LocationId],
       [p].[Surname], [d].[Departmentid], [d].[Name], [l].[LocationId], [l].[City],
       [l].[State]
    FROM [People] AS [p]
    INNER JOIN [Departments] AS [d] ON [p].[DepartmentId] = [d].[Departmentid]
    INNER JOIN [Locations] AS [l] ON [p].[LocationId] = [l].[LocationId]
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (0ms) [Parameters=[], CommandType='Text',
      CommandTimeout='30']
  SELECT [p].[PersonId], [p].[DepartmentId], [p].[Firstname], [p].[LocationId],
      [p].[Surname], [d].[Departmentid], [d].[Name], [l].[LocationId], [l].[City],
      [l].[State]
    FROM [People] AS [p]
    INNER JOIN [Departments] AS [d] ON [p].[DepartmentId] = [d].[Departmentid]
    INNER JOIN [Locations] AS [l] ON [p].[LocationId] = [l].[LocationId]
...
```

Each time the component is rendered, Entity Framework Core sends two identical requests to the database, even when the Increment button is clicked where no data operations are performed.

This issue can arise whenever Entity Framework Core is used and is exacerbated by Blazor. Although it is common practice to assign database queries to `IEnumerable<T>` properties, doing so masks an important aspect of Entity Framework Core, which is that its LINQ expressions are expressions of queries and not results, and each time the property is read, a new query is sent to the database. The value of the `People` property is read twice by the `List` component: once by the `Count` property to determine whether the data has loaded and once by the `@foreach` expression to generate the rows for the HTML table. When the user clicks the Increment button, Blazor renders the `List` component again to figure out what has changed, which causes the `People` property to be read twice more, producing two additional database queries.

Blazor and Entity Framework Core are both working the way they should. Blazor must rerender the component's output to figure out what HTML changes need to be sent to the browser. It has no way of knowing what effect clicking the button has until after it has rendered the elements and evaluated all the Razor expressions. Entity Framework Core is executing its query each time the property is read, ensuring that the application always has fresh data.

This combination of features presents two issues. The first is that needless queries are sent to the database, which can increase the capacity required by an application (although not always because database servers are adept at handling queries).

The second issue is that changes to the database will be reflected in the content presented to the user after they make an unrelated interaction. If another user adds a `Person` object to the database, for example, it will appear in the table the next time the user clicks the Increment button. Users expect applications to reflect only their actions, and unexpected changes are confusing and distracting.

Managing Queries in a Component

The interaction between Blazor and Entity Framework Core won't be a problem for all projects, but, if it is, then the best approach is to query the database once and requery only for operations where the user might expect an update to occur. Some applications may need to present the user with an explicit option to reload the data, especially for applications where updates are likely to occur that the user will want to see, as shown in Listing 36-18.

Listing 36-18. Controlling Queries in the List.razor File in the Blazor/Forms Folder

```
@page "/forms"
@layout EmptyLayout

<h5 class="bg-primary text-white text-center p-2">People</h5>

<table class="table table-sm table-striped table-bordered">
    <thead>
        <tr>
            <th>ID</th><th>Name</th><th>Dept</th><th>Location</th><th></th>
        </tr>
    </thead>
    <tbody>
        @if  (People.Count() == 0) {
            <tr><th colspan="5" class="p-4 text-center">Loading Data...</th></tr>
        } else {
            @foreach (Person p in People) {
                <tr>
                    <td>@p.PersonId</td>
                    <td>@p.Surname, @p.Firstname</td>
                    <td>@p.Department.Name</td>
                    <td>@p.Location.City</td>
                    <td></td>
                </tr>
            }
        }
    </tbody>
</table>

<button class="btn btn-danger" @onclick="UpdateData">Update</button>

<button class="btn btn-primary" @onclick="@(() => Counter++)">Increment</button>
<span class="h5">Counter: @Counter</span>

@code {

    [Inject]
    public DataContext Context { get; set; }

    public IEnumerable<Person> People { get; set; } = Enumerable.Empty<Person>();

    protected async override Task OnInitializedAsync() {
        await UpdateData();
    }

    private async Task UpdateData() =>
        People = await Context.People.Include(p => p.Department)
            .Include(p => p.Location).ToListAsync<Person>();

    public int Counter { get; set; } = 0;
}
```

The UpdateData method performs the same query but applies the ToListAsync method, which forces evaluation of the Entity Framework Core query. The results are assigned to the People property and can be read repeatedly without triggering additional queries. To give the user control over the data, I added a button that invokes the UpdateData method when it is clicked. Restart ASP.NET Core, request http://localhost:5000/forms, and click the Increment button. Monitor the output from the ASP.NET Core server, and you will see that there is a query made only when the component is initialized. To explicitly trigger a query, click the Update button.

Some operations may require a new query, which is easy to perform. To demonstrate, Listing 36-19 adds a sort operation to the List component, which is implemented both with and without a new query.

Listing 36-19. Adding Operations to the List.razor File in the Blazor/Forms Folder

```
@page "/forms"
@page "/forms/list"
@layout EmptyLayout

<h5 class="bg-primary text-white text-center p-2">People</h5>

<table class="table table-sm table-striped table-bordered">
    <thead>
        <tr>
            <th>ID</th><th>Name</th><th>Dept</th><th>Location</th><th></th>
        </tr>
    </thead>
    <tbody>
        @if  (People.Count() == 0) {
            <tr><th colspan="5" class="p-4 text-center">Loading Data...</th></tr>
        } else {
            @foreach (Person p in People) {
                <tr>
                    <td>@p.PersonId</td>
                    <td>@p.Surname, @p.Firstname</td>
                    <td>@p.Department.Name</td>
                    <td>@p.Location.City</td>
                    <td>
                        <NavLink class="btn btn-sm btn-warning"
                                href="@GetEditUrl(p.PersonId)">
                            Edit
                        </NavLink>
                    </td>
                </tr>
            }
        }
    </tbody>
</table>

<button class="btn btn-danger" @onclick="@(() => UpdateData())">Update</button>
<button class="btn btn-info" @onclick="SortWithQuery">Sort (With Query)</button>
<button class="btn btn-info" @onclick="SortWithoutQuery">Sort (No Query)</button>
<button class="btn btn-primary" @onclick="@(() => Counter++)">Increment</button>
<span class="h5">Counter: @Counter</span>

@code {

    [Inject]
    public DataContext Context { get; set; }

    public IEnumerable<Person> People { get; set; } = Enumerable.Empty<Person>();

    protected async override Task OnInitializedAsync() {
        await UpdateData();
    }

    private IQueryable<Person> Query => Context.People.Include(p => p.Department)
            .Include(p => p.Location);
```

```
    private async Task UpdateData(IQueryable<Person> query = null) =>
        People = await (query ?? Query).ToListAsync<Person>();

    public async Task SortWithQuery() {
        await UpdateData(Query.OrderBy(p => p.Surname));
    }

    public void SortWithoutQuery() {
        People = People.OrderBy(p => p.Firstname).ToList<Person>();
    }

    string GetEditUrl(long id) => $"/forms/edit/{id}";

    public int Counter { get; set; } = 0;
}
```

Entity Framework Core queries are expressed as `IQueryable<T>` objects, allowing the query to be composed with additional LINQ methods before it is dispatched to the database server. The new operations in the example both use the LINQ `OrderBy` method, but one applies this to the `IQueryable<T>`, which is then evaluated to send the query with the `ToListAsync` method. The other operation applies the `OrderBy` method to the existing result data, sorting it without sending a new query. To see both operations, restart ASP.NET Core, request `http://localhost:5000/forms`, and click the Sort buttons, as shown in Figure 36-10. When the Sort (With Query) button is clicked, you will see a log message indicating that a query has been sent to the database.

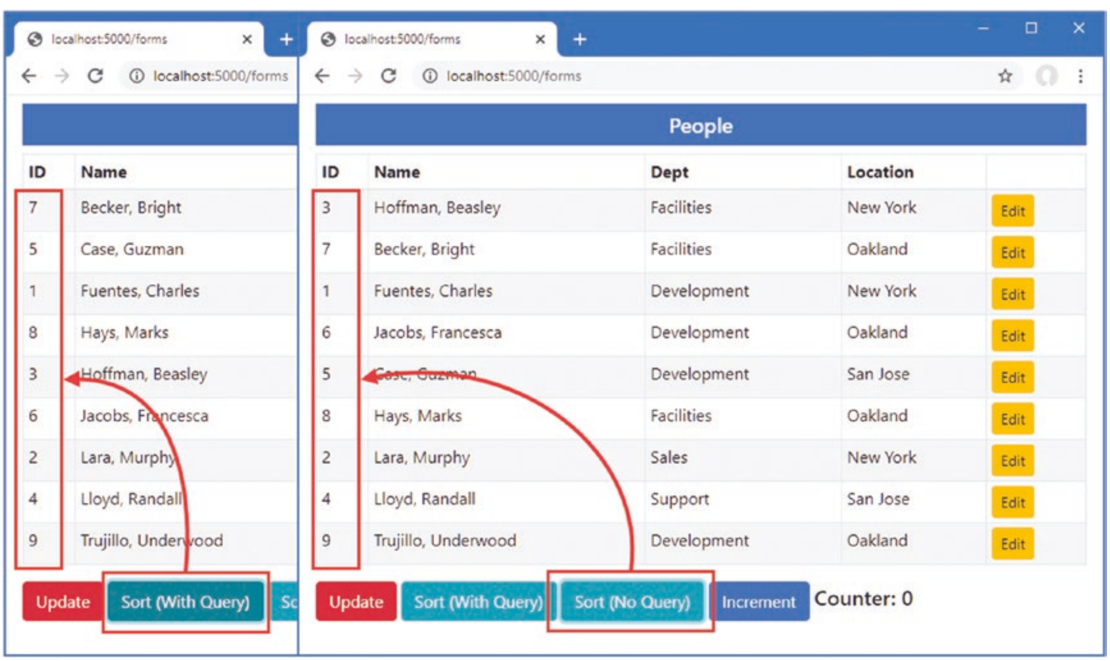

Figure 36-10. Managing component queries

AVOIDING THE OVERLAPPING QUERY PITFALL

You may encounter an exception telling you that “a second operation started on this context before a previous operation completed.” This happens when a child component uses the `OnParametersSetAsync` method to perform an asynchronous Entity Framework Core query and a change in the parent’s data triggers a second call to `OnParametersSetAsync` before the query is complete. The second method call starts a duplicate query that causes the exception. This problem can be resolved by performing the Entity Framework Core query synchronously. You can see an example in Listing 36-12, where I perform queries synchronously because the parent component will trigger an update when it receives its data.

Performing Create, Read, Update, and Delete Operations

To show how the features described in previous sections fit together, I am going to create a simple application that allows the user to perform CRUD operations on Person objects.

Creating the List Component

The List component contains the basic functionality I require. Listing 36-20 removes some of the features from earlier sections that are no longer required and adds buttons that allow the user to navigate to other functions.

Listing 36-20. Preparing the Component in the List.razor File in the Blazor/Forms Folder

```
@page "/forms"
@page "/forms/list"
@layout EmptyLayout
@inherits OwningComponentBase<DataContext>

<h5 class="bg-primary text-white text-center p-2">People</h5>

<table class="table table-sm table-striped table-bordered">
    <thead>
        <tr>
            <th>ID</th><th>Name</th><th>Dept</th><th>Location</th><th></th>
        </tr>
    </thead>
    <tbody>
        @if  (People.Count() == 0) {
            <tr><th colspan="5" class="p-4 text-center">Loading Data...</th></tr>
        } else {
            @foreach (Person p in People) {
                <tr>
                    <td>@p.PersonId</td>
                    <td>@p.Surname, @p.Firstname</td>
                    <td>@p.Department.Name</td>
                    <td>@p.Location.City</td>
                    <td class="text-center">
                        <NavLink class="btn btn-sm btn-info"
                                href="@GetDetailsUrl(p.PersonId)">
                            Details
                        </NavLink>
                        <NavLink class="btn btn-sm btn-warning"
                                href="@GetEditUrl(p.PersonId)">
                            Edit
                        </NavLink>
                        <button class="btn btn-sm btn-danger"
                                 @onclick="@(() => HandleDelete(p))">
                            Delete
                        </button>
                    </td>
                </tr>
            }
        }
    </tbody>
</table>

<NavLink class="btn btn-primary" href="/forms/create">Create</NavLink>
```

```
@code {

    public DataContext Context => Service;

    public IEnumerable<Person> People { get; set; } = Enumerable.Empty<Person>();

    protected async override Task OnInitializedAsync() {
        await UpdateData();
    }

    private IQueryable<Person> Query => Context.People.Include(p => p.Department)
            .Include(p => p.Location);

    private async Task UpdateData(IQueryable<Person> query = null) =>
        People = await (query ?? Query).ToListAsync<Person>();

    string GetEditUrl(long id) => $"/forms/edit/{id}";
    string GetDetailsUrl(long id) => $"/forms/details/{id}";

    public async Task HandleDelete(Person p) {
        Context.Remove(p);
        await Context.SaveChangesAsync();
        await UpdateData();
    }
}
```

The operations for creating, viewing, and editing objects navigate to other URLs, but the delete operations are performed by the `List` component, taking care to reload the data after the changes have been saved to reflect the change to the user.

Creating the Details Component

The details component displays a read-only view of the data, which doesn't require the Blazor form features or present any issues with Entity Framework Core. Add a Blazor Component named `Details.razor` to the `Blazor/Forms` folder with the content shown in Listing 36-21.

Listing 36-21. The Contents of the Details.razor File in the Blazor/Forms Folder

```
@page "/forms/details/{id:long}"
@layout EmptyLayout
@inherits OwningComponentBase<DataContext>

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

<div class="form-group">
    <label>ID</label>
    <input class="form-control" value="@PersonData.PersonId" disabled />
</div>
<div class="form-group">
    <label>Firstname</label>
    <input class="form-control" value="@PersonData.Firstname" disabled />
</div>
<div class="form-group">
    <label>Surname</label>
    <input class="form-control" value="@PersonData.Surname" disabled />
</div>
```

```
<div class="form-group">
    <label>Department</label>
    <input class="form-control" value="@PersonData.Department?.Name" disabled />
</div>
<div class="form-group">
    <label>Location</label>
    <input class="form-control"
           value="@($"{PersonData.Location?.City}, {PersonData.Location?.State}")"
           disabled />
</div>
<div class="text-center">
    <NavLink class="btn btn-info" href="@EditUrl">Edit</NavLink>
    <NavLink class="btn btn-secondary" href="/forms">Back</NavLink>
</div>

@code {

    [Inject]
    public NavigationManager NavManager { get; set; }

    DataContext Context => Service;

    [Parameter]
    public long Id { get; set; }

    public Person PersonData { get; set; } = new Person();

    protected async override Task OnParametersSetAsync() {
        PersonData = await Context.People.Include(p => p.Department)
            .Include(p => p.Location).FirstOrDefaultAsync(p => p.PersonId == Id);
    }

    public string EditUrl => $"/forms/edit/{Id}";
}
```

All the input elements displayed by this component are disabled, which means there is no need to handle events or process user input.

Creating the Editor Component

The remaining features will be handled by the Editor component. Listing 36-22 removes the features from earlier examples that are no longer required and adds support for creating and editing objects, including persisting the data.

Listing 36-22. Adding Application Features in the Editor.razor File in the Forms/Blazor Folder

```
@page "/forms/edit/{id:long}"
@page "/forms/create"
@layout EmptyLayout
@inherits OwningComponentBase<DataContext>

<link href="/blazorValidation.css" rel="stylesheet" />

<h4 class="bg-@Theme text-center text-white p-2">@Mode</h4>

<EditForm Model="PersonData" OnValidSubmit="HandleValidSubmit">
    <DataAnnotationsValidator />
```