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.

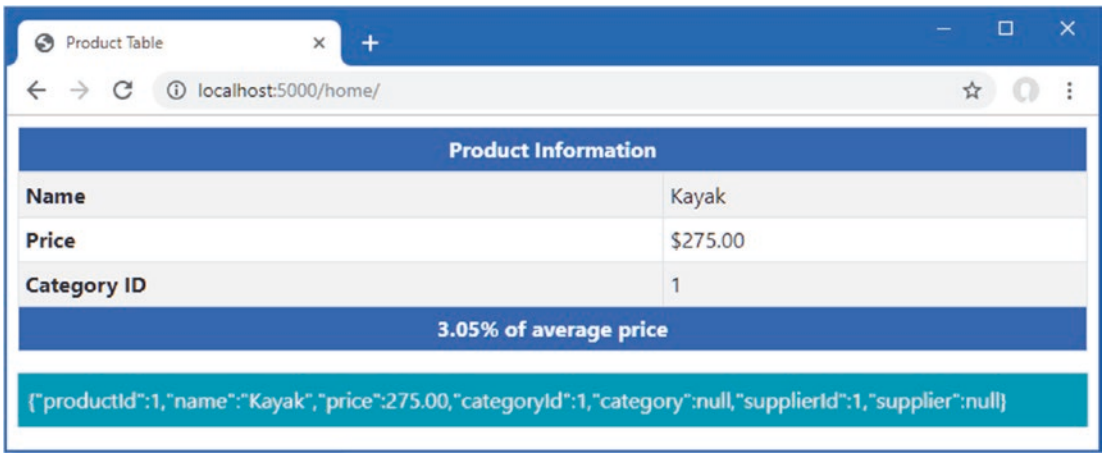

Figure 23-1. *Running the example application*

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.

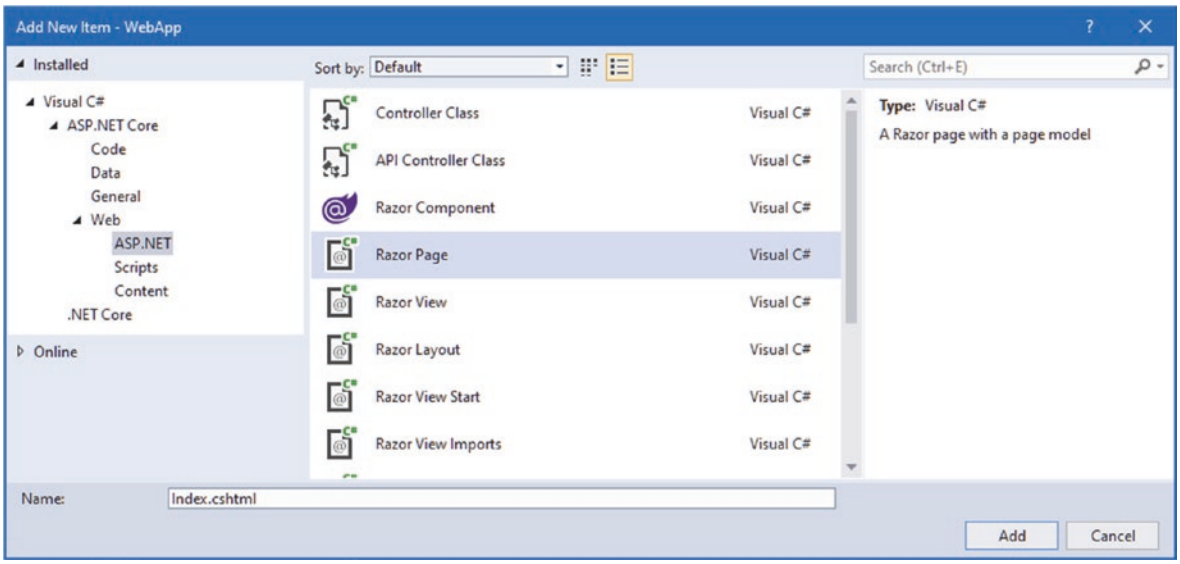

Figure 23-2. *Creating a Razor Page*

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.

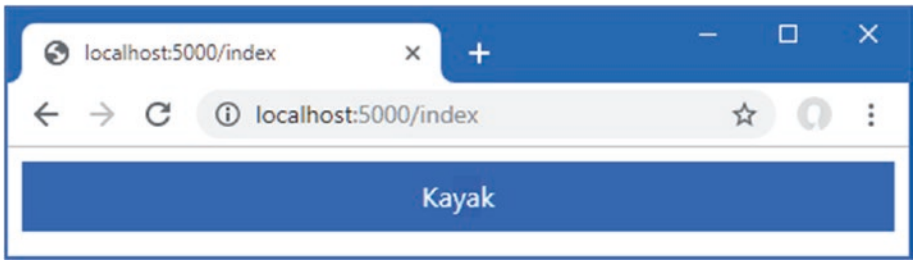

Figure 23-3. *Using a Razor Page*

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.

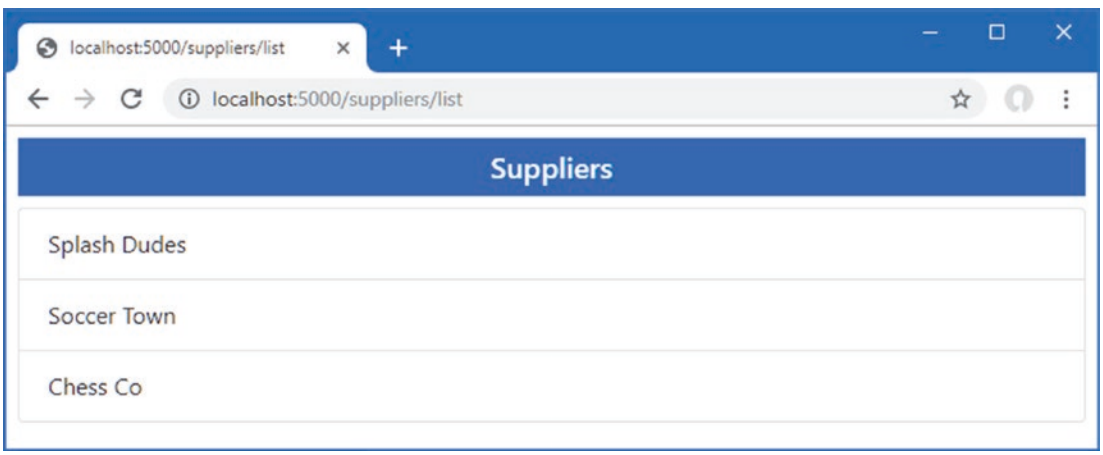

Figure 23-4. *Using a folder structure to route requests*

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.

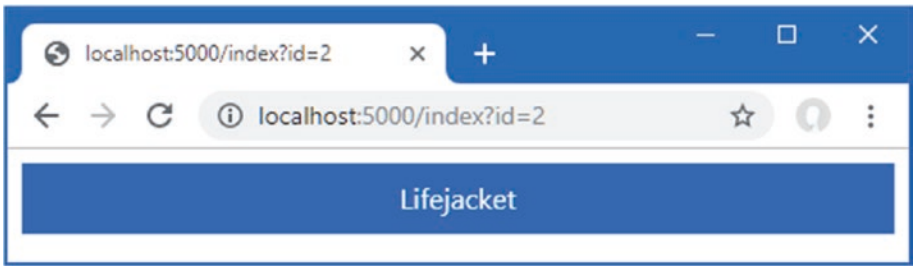

Figure 23-5. *Using a query string parameter*

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.

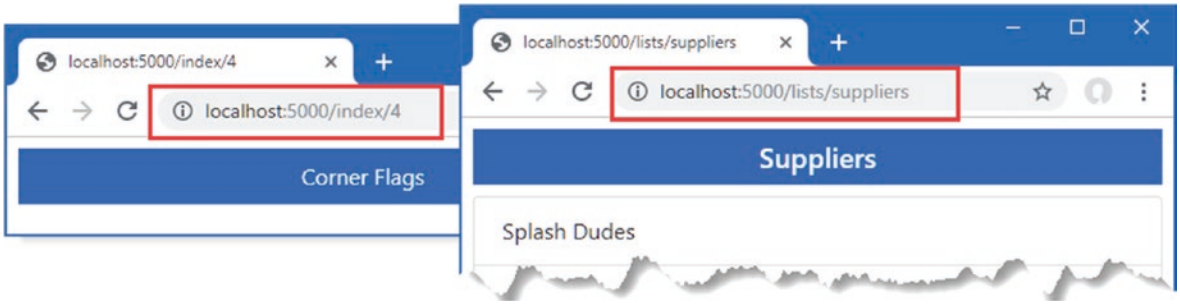

Figure 23-6. *Changing routes using the @page directive*

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.

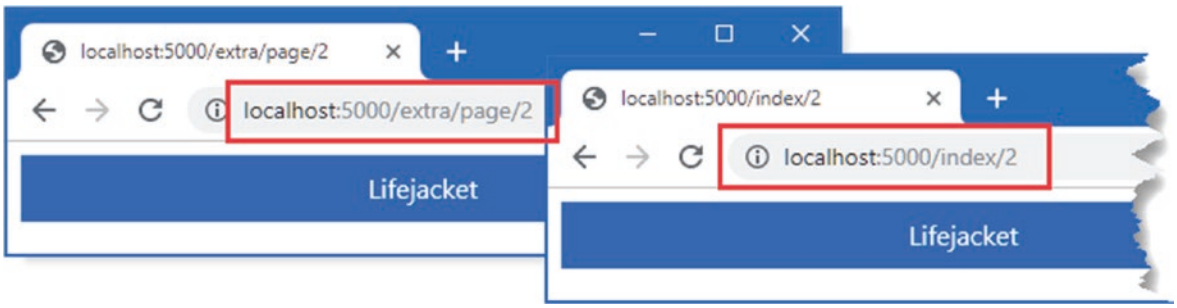

Figure 23-7. *Adding a route for a Razor Page*

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.

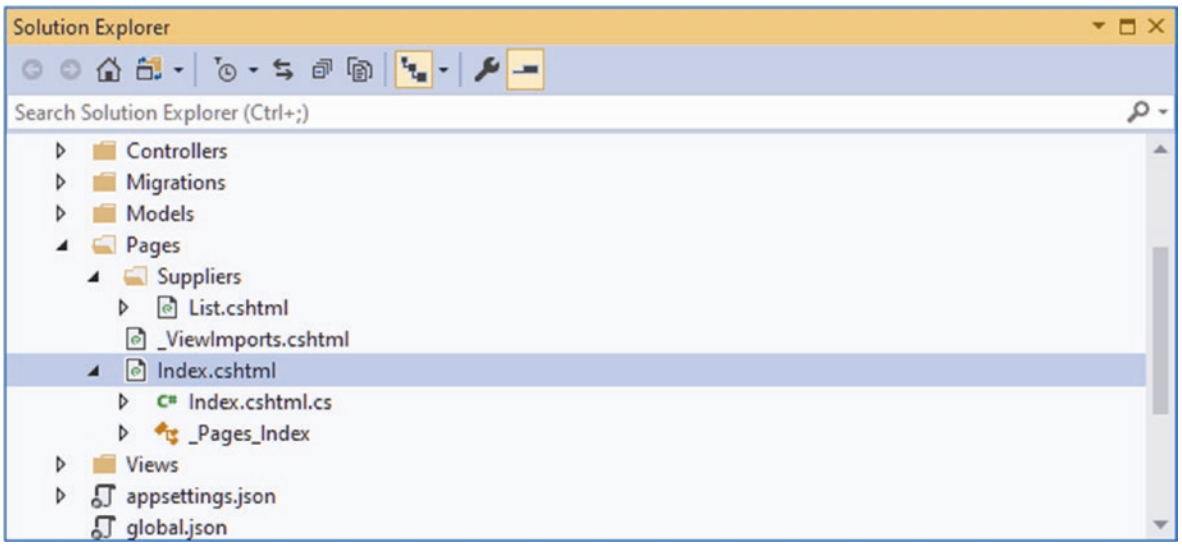

Figure 23-8. *Revealing the code-behind file in the Visual Studio Solution Explorer*

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.

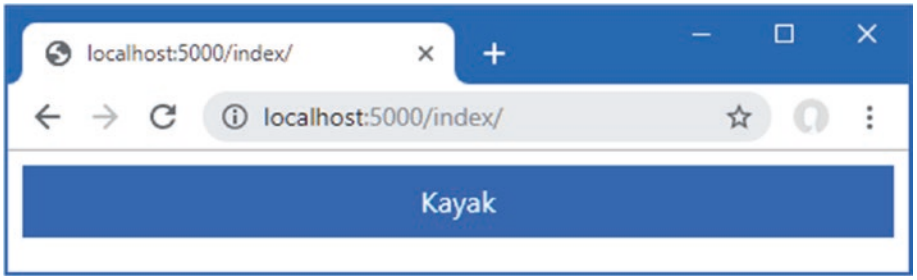

Figure 23-9. *Using a code-behind file*

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)RedirectTo ActionPermanent(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.

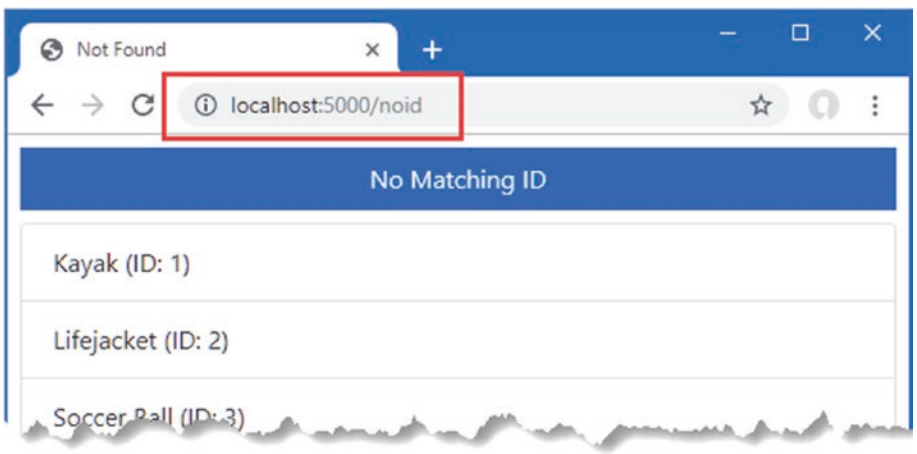

Figure 23-10. *Redirecting to a different Razor Page*

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.

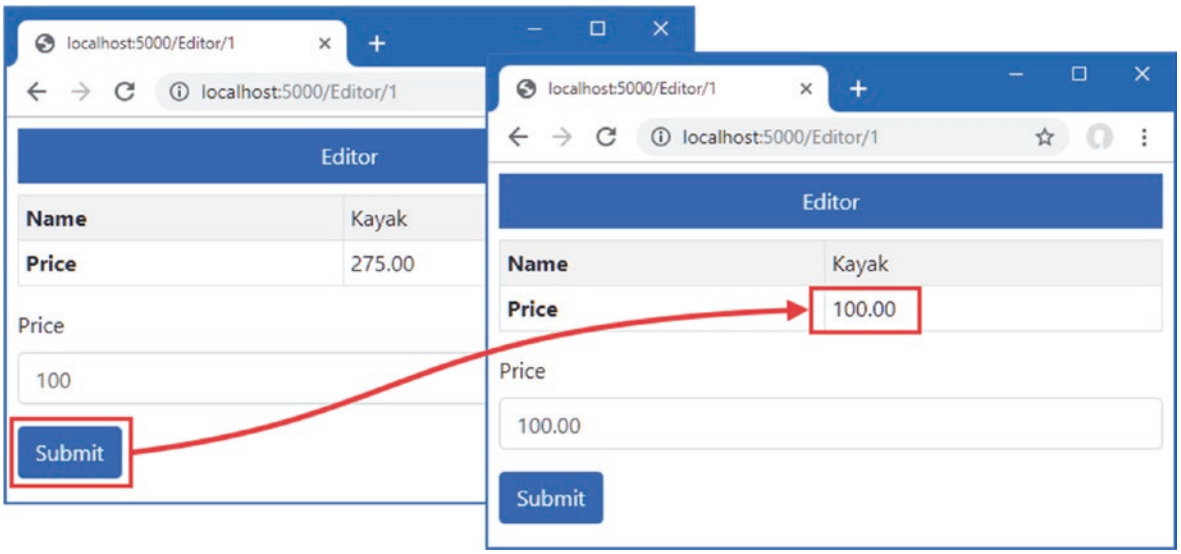

Figure 23-11. *Handling multiple HTTP methods*

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.

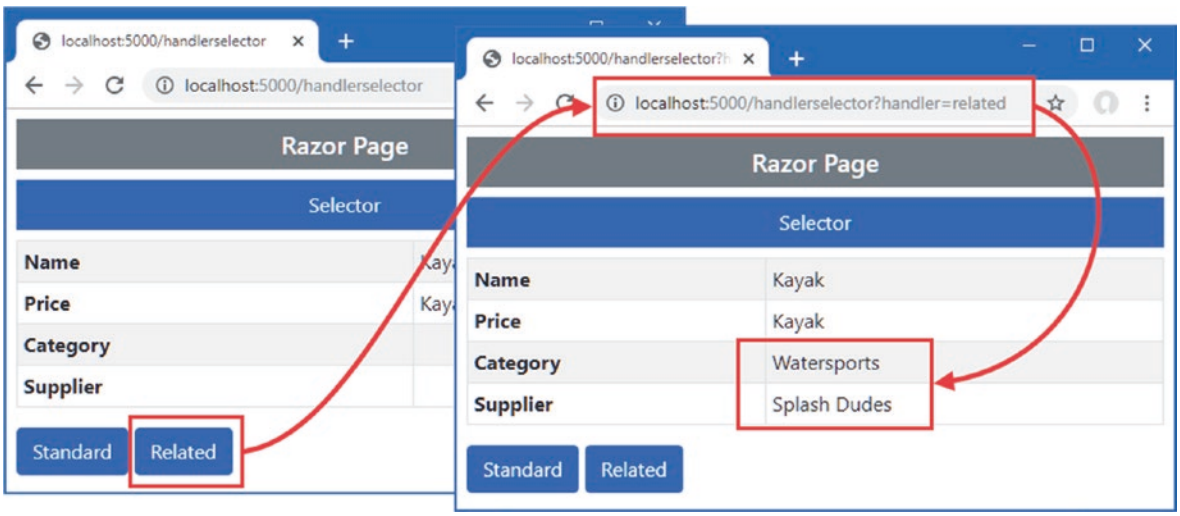

Figure 23-12. *Selecting handler methods*

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.

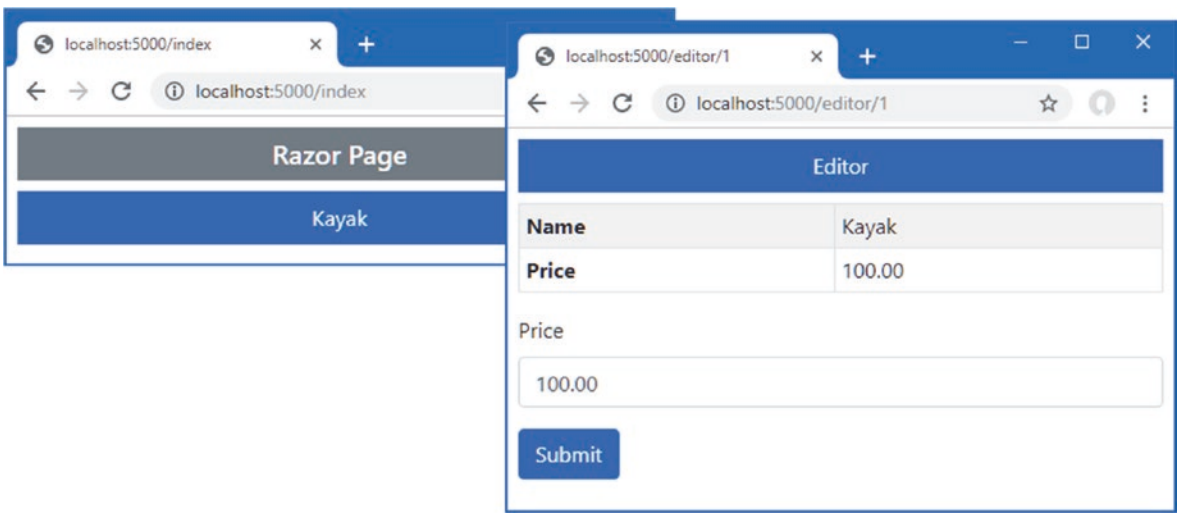

Figure 23-13. *Using a layout in Razor Pages*

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.

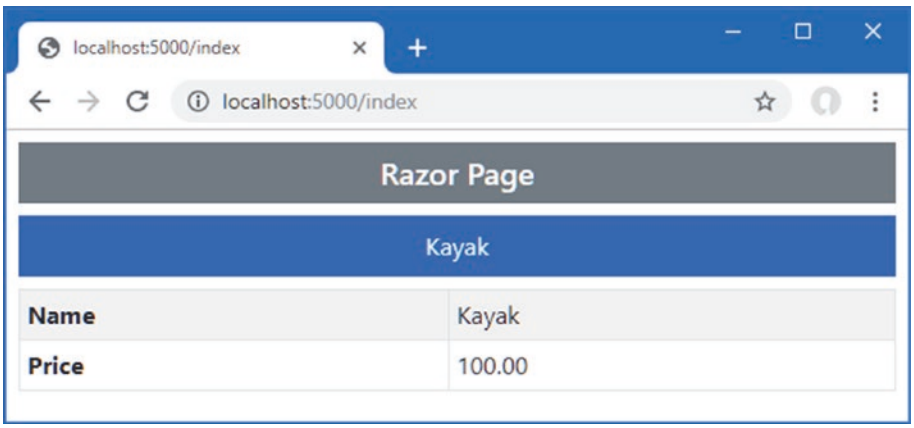

Figure 23-14. *Using a partial view*

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.

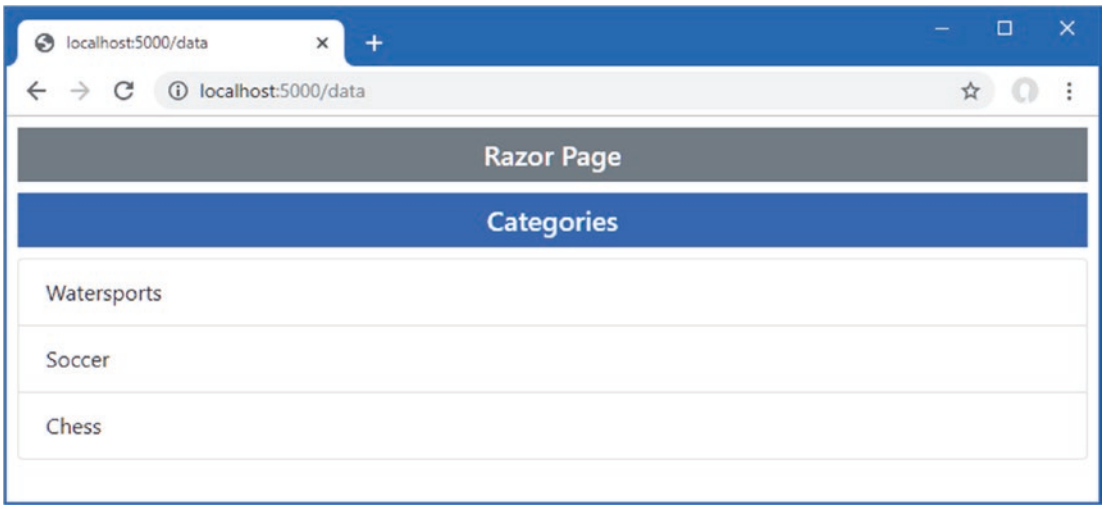

Figure 23-15. *Using a Razor Page without a page model*

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.