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.

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
```

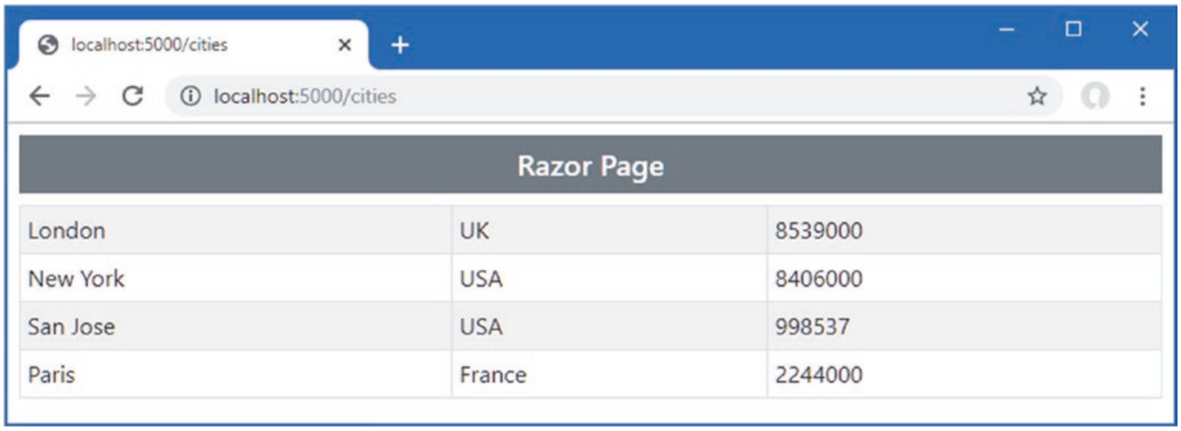
Running the Example Application

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.



Razor Page

London	UK	8539000
New York	USA	8406000
San Jose	USA	998537
Paris	France	2244000

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>
}
```

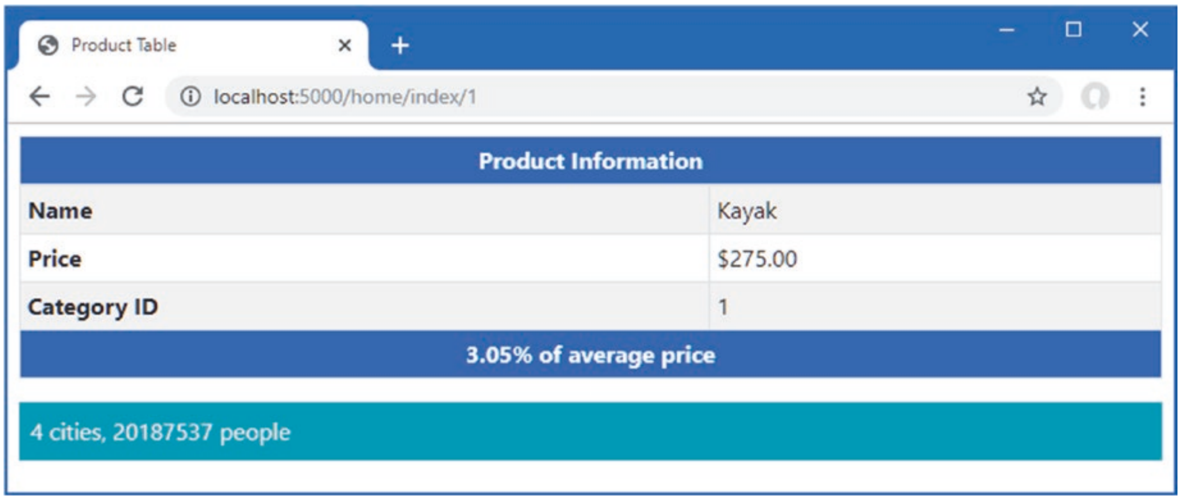
View components are applied using the `Component.InvokeAsync` method, using the name of the view component class as the argument. The syntax for this technique can be confusing. View component classes define either an `Invoke` or `InvokeAsync` method, depending on whether their work is performed synchronously 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.



Product Information	
Name	Kayak
Price	$275.00
Category ID	1

3.05% of average price

4 cities, 20187537 people

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.

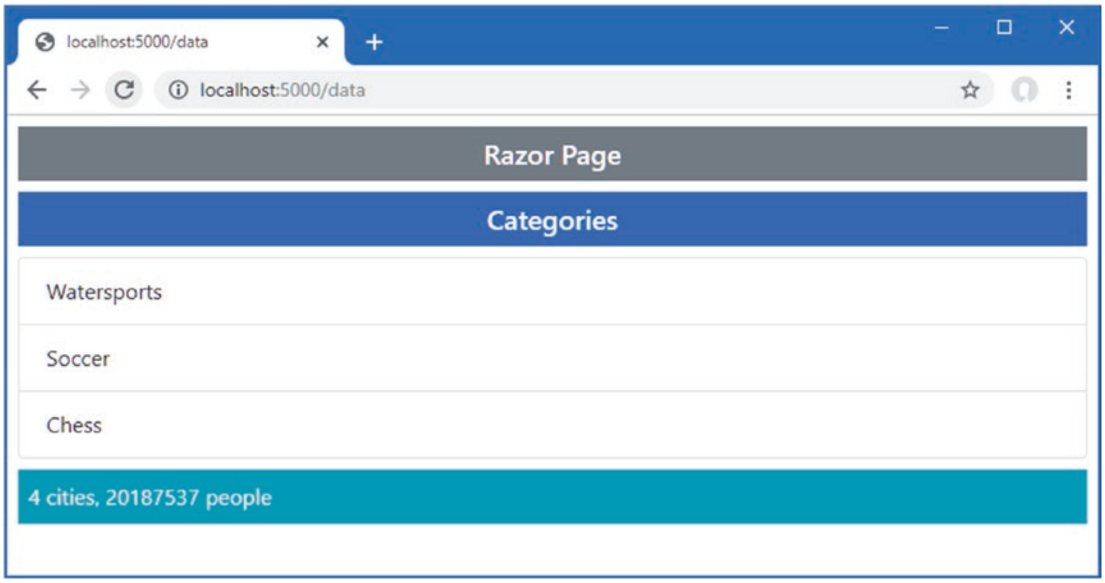


Figure 24-3. *Using a view component in a Razor Page*

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

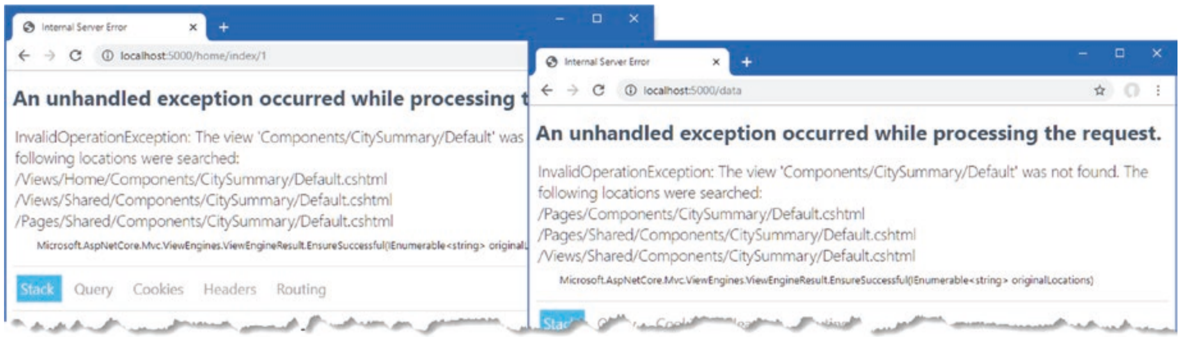
```
namespace WebApp.Models {

    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.



An unhandled exception occurred while processing t

InvalidOperationException: The view 'Components/CitySummary/Default' was
following locations were searched:
/Views/Home/Components/CitySummary/Default.cshtml
/Views/Shared/Components/CitySummary/Default.cshtml
/Pages/Shared/Components/CitySummary/Default.cshtml

```
Microsoft.AspNetCore.Mvc.ViewEngines.ViewEngineResult.EnsureSuccessful(IEnumerable<string> originalL
```

An unhandled exception occurred while processing the request.

InvalidOperationException: The view 'Components/CitySummary/Default' was not found. The following locations were searched:
/Pages/Components/CitySummary/Default.cshtml
/Pages/Shared/Components/CitySummary/Default.cshtml
/Views/Shared/Components/CitySummary/Default.cshtml

```
Microsoft.AspNetCore.Mvc.ViewEngines.ViewEngineResult.EnsureSuccessful(IEnumerable<string> originalLocations)
```

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.

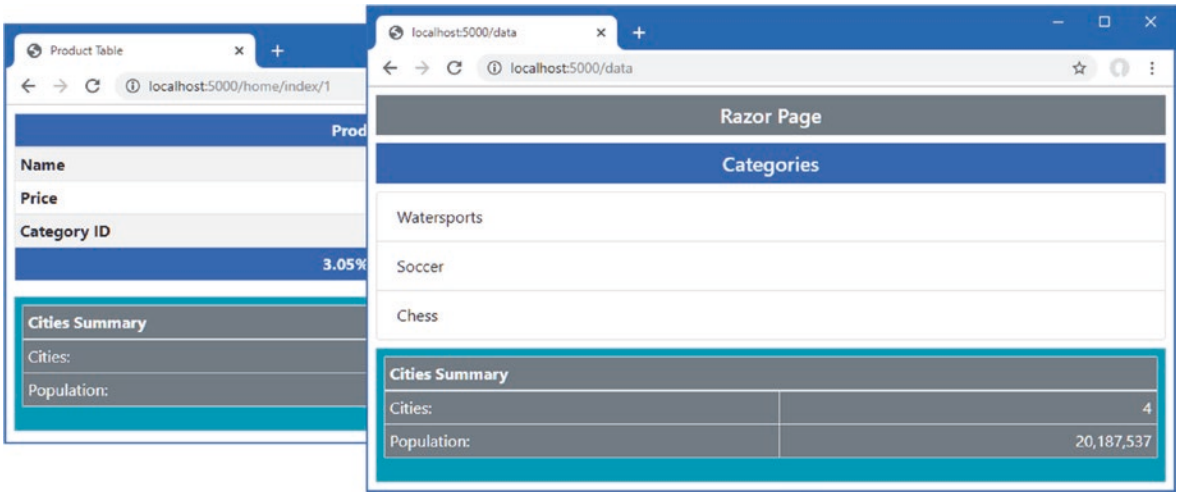


Figure 24-5. *Using a view with a view component*

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.

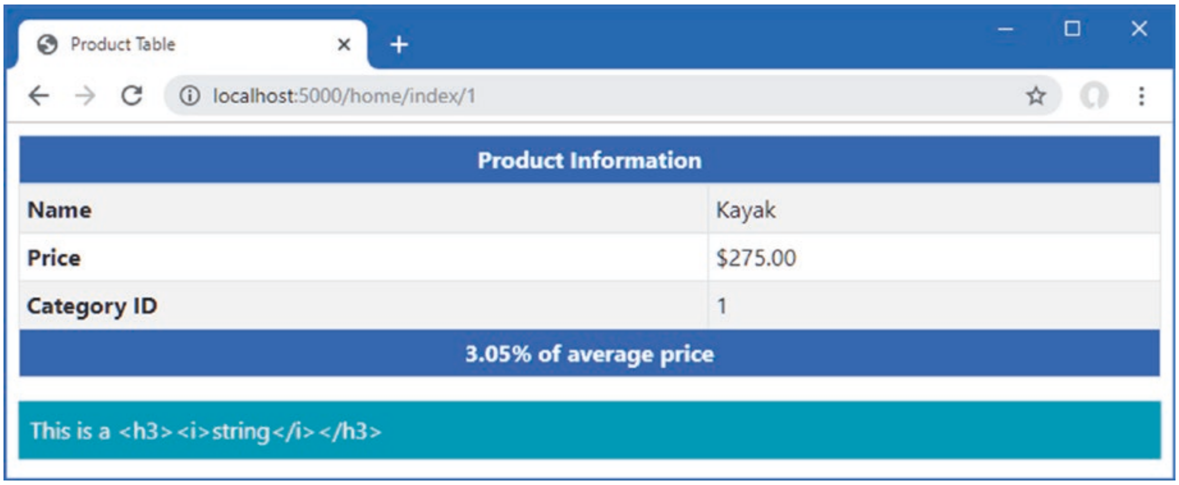
■ **Tip** In addition to the `Content` method, the `Invoke` method can return a `string`, which will be automatically converted to a `ContentViewComponentResult`. This is the approach I took in the view component when it was first defined.

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.



Product Information	
Name	Kayak
Price	$275.00
Category ID	1

3.05% of average price

This is a <h3><i>string</i></h3>

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>
...
```

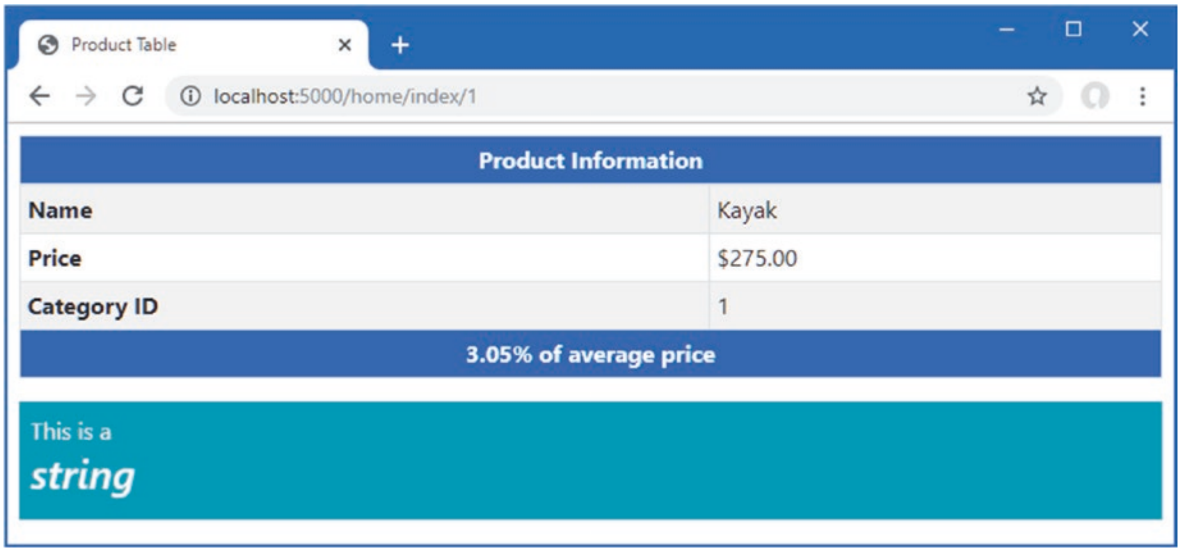
You don't need to encode content if you trust its source and want it to be interpreted as HTML. The `Content` method always encodes its argument, so you must create the `HtmlContentViewComponentResult` object directly and provide its constructor with an `HtmlString` object, which represents a string 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.



Product Information	
Name	Kayak
Price	$275.00
Category ID	1
3.05% of average price	

This is a
string

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.

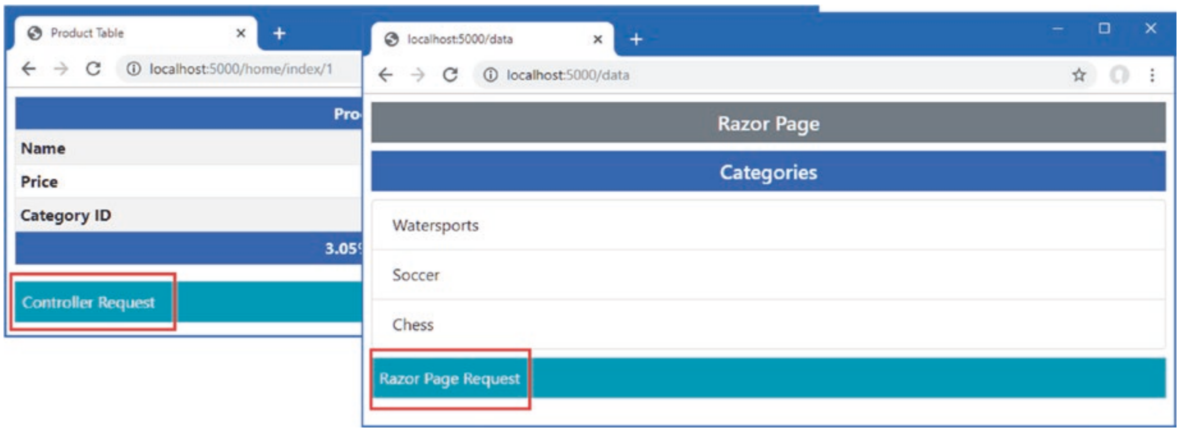


Figure 24-8. *Using context data in a view component*

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.

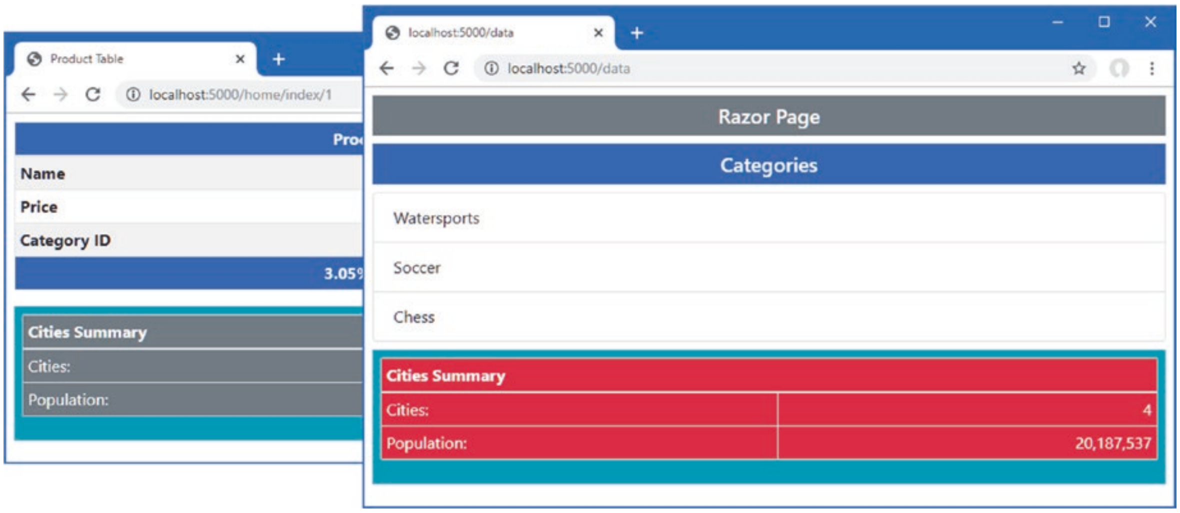


Figure 24-9. *Using context data in a view component*

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.

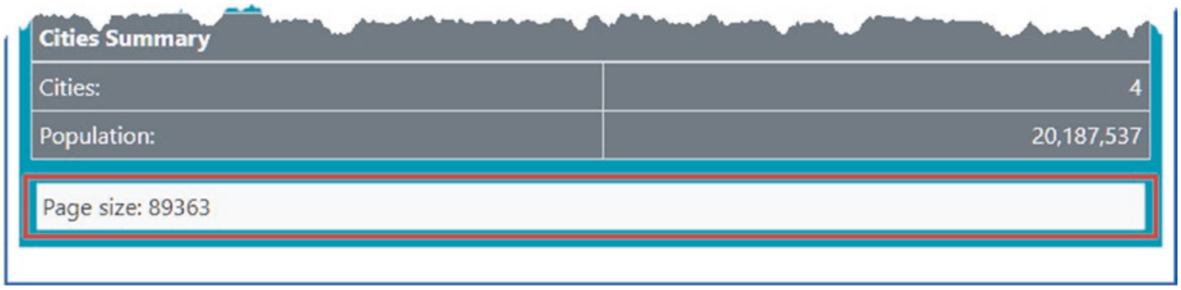
Listing 24-26. Using an Asynchronous Component in the Index.cshtml File in the Views/Home Folder

```
@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.



Cities Summary	
Cities:	4
Population:	20,187,537

Page size: 89363

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

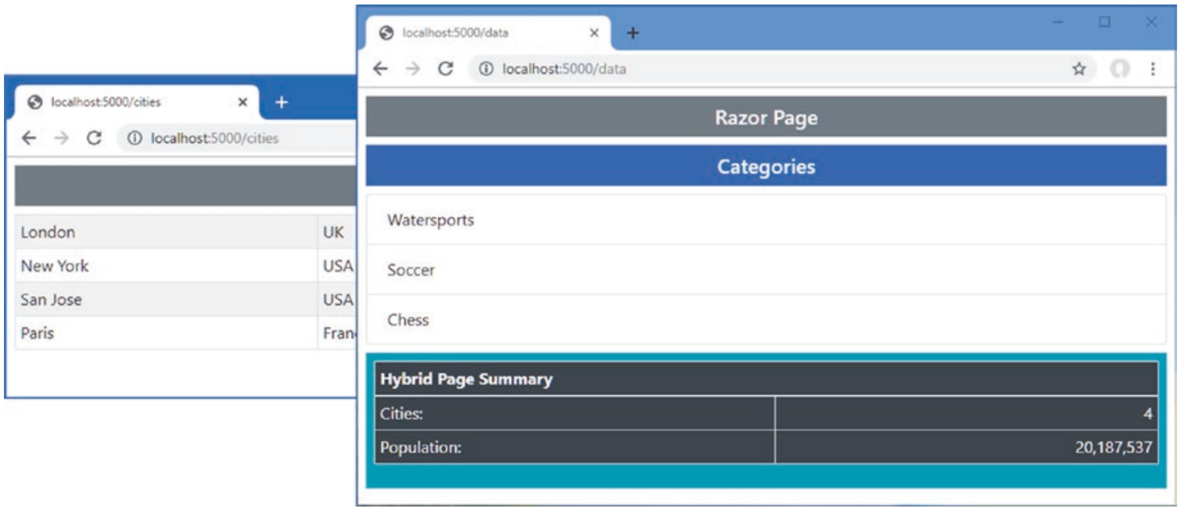
```
@model CityViewModel

<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.



London	UK
New York	USA
San Jose	USA
Paris	Fran

Razor Page

Categories

Watersports

Soccer

Chess

Hybrid Page Summary	
Cities:	4
Population:	20,187,537

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>
```

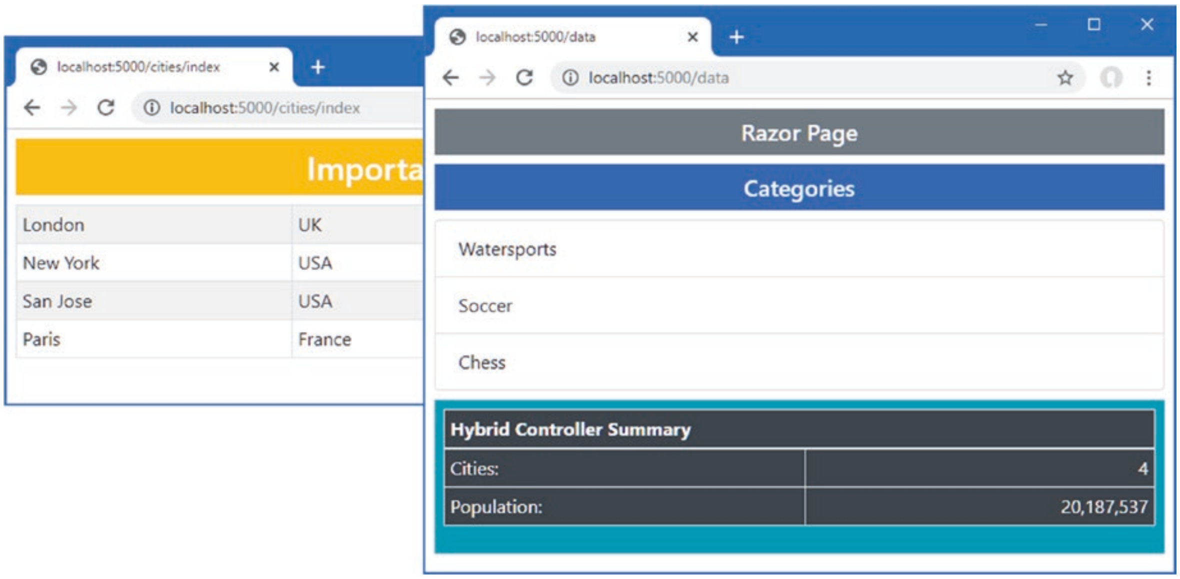
Listing 24-34 applies the hybrid view component in the `Data.cshtml` Razor Page, replacing the hybrid class created in the previous section.

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.



London	UK
New York	USA
San Jose	USA
Paris	France

Hybrid Controller Summary	
Cities:	4
Population:	20,187,537

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.