



Using the Built-in Tag Helpers

ASP.NET Core provides a set of built-in tag helpers that apply the most commonly required element transformations. In this chapter, I explain those tag helpers that deal with anchor, script, link, and image elements, as well as features for caching content and selecting content based on the environment. In Chapter 27, I describe the tag helpers that support HTML forms. Table 26-1 puts the built-in tag helpers in context.

Table 26-1. *Putting the Built-in Tag Helpers in Context*

Question	Answer
What are they?	The built-in tag helpers perform commonly required transformations on HTML elements.
Why are they useful?	Using the built-in tag helpers means you don't have to create custom helpers using the techniques in Chapter 25.
How are they used?	The tag helpers are applied using attributes on standard HTML elements or through custom HTML elements.
Are there any pitfalls or limitations?	No, these tag helpers are well-tested and easy to use. Unless you have unusual needs, using these tag helpers is preferable to custom implementation.
Are there any alternatives?	These tag helpers are optional, and their use is not required.

Table 26-2 summarizes the chapter.

Table 26-2. *Chapter Summary*

Problem	Solution	Listing
Creating elements that target endpoints	Use the anchor element tag helper attributes	7, 8
Including JavaScript files in a response	Use the JavaScript tag helper attributes	9–13
Including CSS files in a response	Use the CSS tag helper attributes	14, 15
Managing image caching	Use the image tag helper attributes	16
Caching sections of a view	Use the caching tag helper	17–21
Varying content based on the application environment	Use the environment tag helper	22

Preparing for This Chapter

This chapter uses the WebApp project from Chapter 25. To prepare for this chapter, comment out the statements that register the tag component helpers in the Startup class, as shown in Listing 26-1.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/apress/pro-asp.net-core-3>. See Chapter 1 for how to get help if you have problems running the examples.

Listing 26-1. The Contents of the Startup.cs File in the WebApp Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using WebApp.Models;
//using Microsoft.AspNetCore.Razor.TagHelpers;
//using WebApp.TagHelpers;

namespace WebApp {
    public class Startup {

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

        public IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<DataContext>(opts => {
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:ProductConnection"]);
                opts.EnableSensitiveDataLogging(true);
            });
            services.AddControllersWithViews().AddRazorRuntimeCompilation();
            services.AddRazorPages().AddRazorRuntimeCompilation();
            services.AddSingleton<CitiesData>();
            //services.AddTransient<ITagHelperComponent, TimeTagHelperComponent>();
            //services.AddTransient<ITagHelperComponent,
            //    TableFooterTagHelperComponent>();
        }

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

Next, update the `_RowPartial.cshtml` partial view in the Views/Home folder, making the changes shown in Listing 26-2.

Listing 26-2. Making Changes in the `_RowPartial.cshtml` File in the Views/Home Folder

```
@model Product

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

Add the elements shown in Listing 26-3 to define additional columns in the table rendered in the Home controller's List view.

Listing 26-3. Adding Elements in the `List.cshtml` File in the Views/Home Folder

```
@model IEnumerable<Product>
@{ Layout = "_SimpleLayout"; }

<h6 class="bg-secondary text-white text-center m-2 p-2">Products</h6>
<div class="m-2">
    <table class="table table-sm table-striped table-bordered">
        <thead>
            <tr>
                <th>Name</th><th>Price</th>
                <th>Category</th><th>Supplier</th><th></th>
            </tr>
        </thead>
        <tbody>
            @foreach (Product p in Model) {
                <partial name="_RowPartial" model="p" />
            }
        </tbody>
    </table>
</div>
```

Adding an Image File

One of the tag helpers described in this chapter provides services for images. I created the `wwwroot/images` folder and added an image file called `city.png`. This is a public domain panorama of the New York City skyline, as shown in Figure 26-1.

**Figure 26-1.** Adding an image to the project

This image file is included in the source code for this chapter, which is available in the GitHub repository for this book. You can substitute your own image if you don't want to download the example project.

Installing a Client-Side Package

Some of the examples in this chapter demonstrate the tag helper support for working with JavaScript files, for which I use the jQuery package. Use a PowerShell command prompt to run the command shown in Listing 26-4 in the project folder, which contains the WebApp.csproj file. If you are using Visual Studio, you can select Project ► Manage Client-Side Libraries to select the jQuery package.

Listing 26-4. Installing a Package

```
libman install jquery@3.4.1 -d wwwroot/lib/jquery
```

Dropping the Database

Open a new PowerShell command prompt, navigate to the folder that contains the WebApp.csproj file, and run the command shown in Listing 26-5 to drop the database.

Listing 26-5. Dropping the Database

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

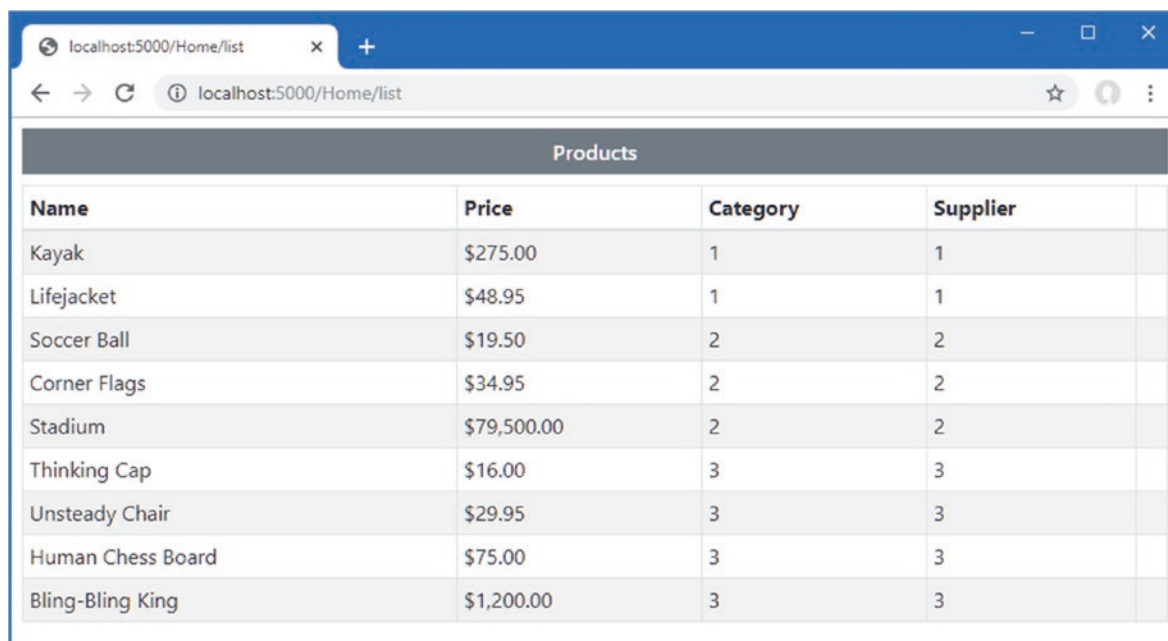
Running the Example Application

Select Start Without Debugging or Run Without Debugging from the Debug menu or use the PowerShell command prompt to run the command shown in Listing 26-6.

Listing 26-6. Running the Example Application

```
dotnet run
```

Use a browser to request <http://localhost:5000/Home/list>, which will display a list of products, as shown in Figure 26-2.



Products			
Name	Price	Category	Supplier
Kayak	\$275.00	1	1
Lifejacket	\$48.95	1	1
Soccer Ball	\$19.50	2	2
Corner Flags	\$34.95	2	2
Stadium	\$79,500.00	2	2
Thinking Cap	\$16.00	3	3
Unsteady Chair	\$29.95	3	3
Human Chess Board	\$75.00	3	3
Bling-Bling King	\$1,200.00	3	3

Figure 26-2. Running the example application

Enabling the Built-in Tag Helpers

The built-in tag helpers are all defined in the `Microsoft.AspNetCore.Mvc.TagHelpers` namespace and are enabled by adding an `@addTagHelpers` directive to individual views or pages or, as in the case of the example project, to the view imports file. Here is the required directive from the `_ViewImports.cshtml` file in the Views folder, which enables the built-in tag helpers for controller views:

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

Here is the corresponding directive in the `_ViewImports.cshtml` file in the Pages folder, which enables the built-in tag helpers for Razor Pages:

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

These directives were added to the example project in Chapter 24 to enable the view components feature.

Transforming Anchor Elements

The `a` element is the basic tool for navigating around an application and sending GET requests to the application. The `AnchorTagHelper` class is used to transform the `href` attribute of `a` elements so they target URLs generated using the routing system, which means that hard-coded URLs are not required and a change in the routing configuration will be automatically reflected in the application's anchor elements. Table 26-3 describes the attributes the `AnchorTagHelper` class supports.

Table 26-3. *The Built-in Tag Helper Attributes for Anchor Elements*

Name	Description
<code>asp-action</code>	This attribute specifies the action method that the URL will target.
<code>asp-controller</code>	This attribute specifies the controller that the URL will target. If this attribute is omitted, then the URL will target the controller or page that rendered the current view.
<code>asp-page</code>	This attribute specifies the Razor Page that the URL will target.
<code>asp-page-handler</code>	This attribute specifies the Razor Page handler function that will process the request, as described in Chapter 23.
<code>asp-fragment</code>	This attribute is used to specify the URL fragment (which appears after the <code>#</code> character).
<code>asp-host</code>	This attribute specifies the name of the host that the URL will target.
<code>asp-protocol</code>	This attribute specifies the protocol that the URL will use.
<code>asp-route</code>	This attribute specifies the name of the route that will be used to generate the URL.
<code>asp-route-*</code>	Attributes whose name begins with <code>asp-route-</code> are used to specify additional values for the URL so that the <code>asp-route-id</code> attribute is used to provide a value for the <code>id</code> segment to the routing system.
<code>asp-all-route-data</code>	This attribute provides values used for routing as a single value, rather than using individual attributes.

The `AnchorTagHelper` is simple and predictable and makes it easy to generate URLs in `a` elements that use the application's routing configuration. Listing 26-7 adds an anchor element that uses attributes from the table to create a URL that targets another action defined by the `Home` controller.

Listing 26-7. Transforming an Element in the `_RowPartial.cshtml` File in the Views/Home Folder

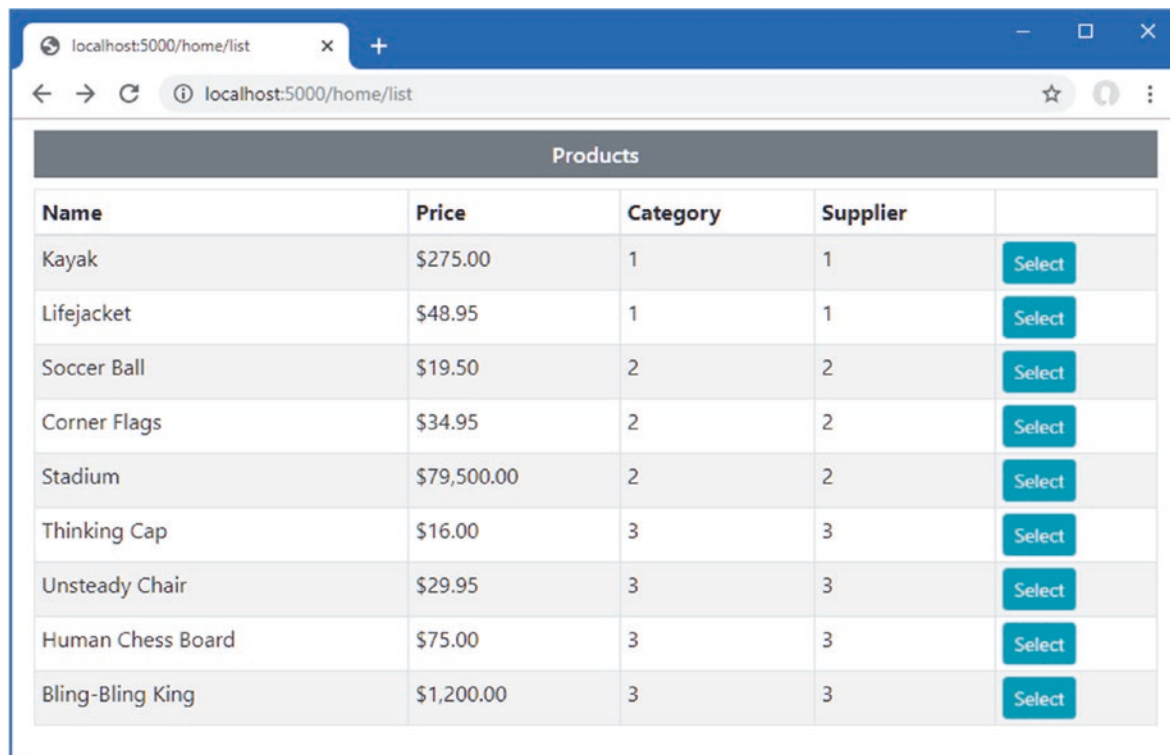
```
@model Product
```

```
<tr>
    <td>@Model.Name</td>
    <td>@Model.Price.ToString("c")</td>
    <td>@Model.CategoryId</td>
    <td>@Model.SupplierId</td>
    <td>
        <a asp-action="index" asp-controller="home" asp-route-id="@Model.ProductId"
            class="btn btn-sm btn-info">
            Select
        </a>
    </td>
</tr>
```

The `asp-action` and `asp-controller` attributes specify the name of the action method and the controller that defines it. Values for segment variables are defined using `asp-route-[name]` attributes, such that the `asp-route-id` attribute provides a value for the `id` segment variable that is used to provide an argument for the action method selected by the `asp-action` attribute.

■ **Tip** The `class` attributes added to the anchor elements in Listing 26-7 apply Bootstrap CSS Framework styles that give the elements the appearance of buttons. This is not a requirement for using the tag helper.

To see the anchor element transformations, use a browser to request `http://localhost:5000/home/list`, which will produce the response shown in Figure 26-3.



The screenshot shows a web browser window with the address bar set to `localhost:5000/home/list`. The page displays a table titled "Products" with the following data:

Name	Price	Category	Supplier	
Kayak	\$275.00	1	1	Select
Lifejacket	\$48.95	1	1	Select
Soccer Ball	\$19.50	2	2	Select
Corner Flags	\$34.95	2	2	Select
Stadium	\$79,500.00	2	2	Select
Thinking Cap	\$16.00	3	3	Select
Unsteady Chair	\$29.95	3	3	Select
Human Chess Board	\$75.00	3	3	Select
Bling-Bling King	\$1,200.00	3	3	Select

Figure 26-3. Transforming anchor elements

If you examine the Select anchor elements, you will see that each href attribute includes the ProductId value of the Product object it relates to, like this:

```
...
<a class="btn btn-sm btn-info" href="/Home/index/3">Select</a>
...
```

In this case, the value provided by the asp-route-id attribute means the default URL cannot be used, so the routing system has generated a URL that includes segments for the controller and action name, as well as a segment that will be used to provide a parameter to the action method. In both cases, since only an action method was specified, the URLs created by the tag helper target the controller that rendered the view. Clicking the anchor elements will send an HTTP GET request that targets the Home controller's Index method.

Using Anchor Elements for Razor Pages

The asp-page attribute is used to specify a Razor Page as the target for an anchor element's href attribute. The path to the page is prefixed with the / character, and values for route segments defined by the @page directive are defined using asp-route-[name] attributes. Listing 26-8 adds an anchor element that targets the List page defined in the Pages/Suppliers folder.

■ **Note** The asp-page-handler attribute can be used to specify the name of the page model handler method that will process the request.

Listing 26-8. Targeting a Razor Page in the List.cshtml File in the Views/Home Folder

```
@model IEnumerable<Product>
@{
    Layout = "_SimpleLayout";
}

<h6 class="bg-secondary text-white text-center m-2 p-2">Products</h6>
<div class="m-2">
    <table class="table table-sm table-striped table-bordered">
        <thead>
            <tr>
                <th>Name</th><th>Price</th>
                <th>Category</th><th>Supplier</th><th></th>
            </tr>
        </thead>
        <tbody>
            @foreach (Product p in Model) {
                <partial name="_RowPartial" model="p" />
            }
        </tbody>
    </table>
    <a asp-page="/suppliers/list" class="btn btn-secondary">Suppliers</a>
</div>
```

Use a browser to request `http://localhost:5000/home/list`, and you will see the anchor element, which is styled to appear as a button. If you examine the HTML sent to the client, you will see the anchor element has been transformed like this:

```
...
<a class="btn btn-secondary" href="/lists/suppliers">Suppliers</a>
...
```

This URL used in the href attribute reflects the @page directive, which has been used to override the default routing convention in this page. Click the element, and the browser will display the Razor Page, as shown in Figure 26-4.

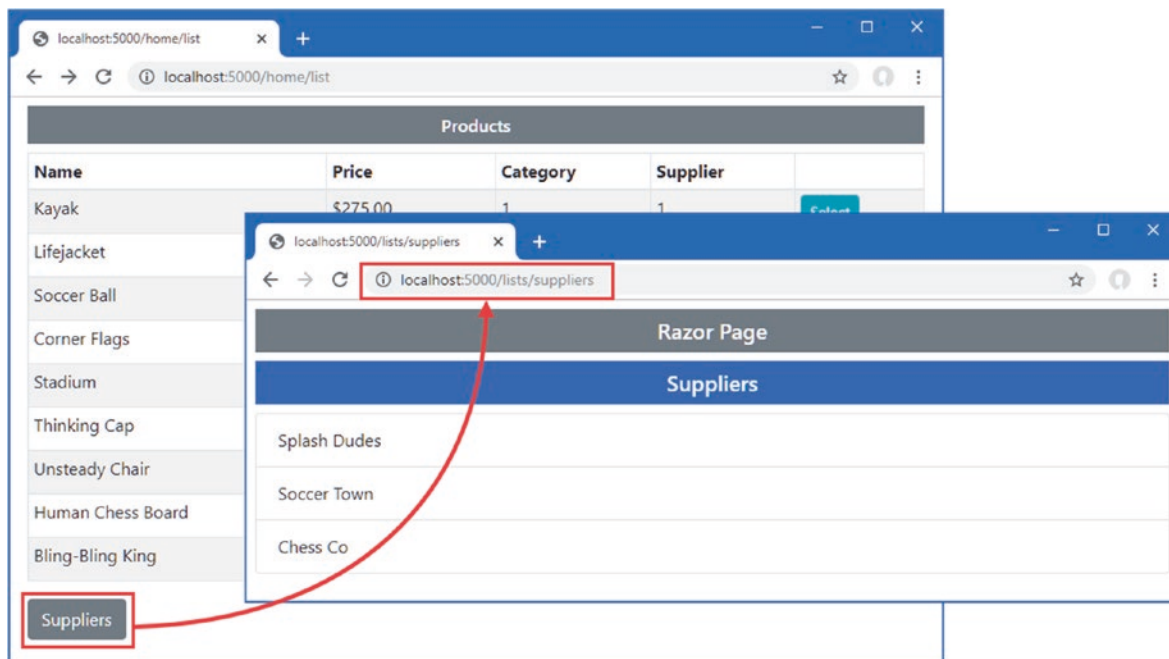


Figure 26-4. Targeting a Razor Page with an anchor element

GENERATING URLs (AND NOT LINKS)

The tag helper generates URLs only in anchor elements. If you need to generate a URL, rather than a link, then you can use the `Url` property, which is available in controllers, page models, and views. This property returns an object that implements the `IUrlHelper` interface, which provides a set of methods and extension methods that generate URLs. Here is a Razor fragment that generates a URL in a view:

```
...
<div>@Url.Page("/suppliers/list")</div>
...
```

This fragment produces a `div` element whose content is the URL that targets the `/Suppliers/List` Razor Page. The same interface is used in controllers or page model classes, such as with this statement:

```
...
string url = Url.Action("List", "Home");
...
```

The statement generates a URL that targets the `List` action on the `Home` controller and assigns it to the `string` variable named `url`.

Using the JavaScript and CSS Tag Helpers

ASP.NET Core provides tag helpers that are used to manage JavaScript files and CSS stylesheets through the `script` and `link` elements. As you will see in the sections that follow, these tag helpers are powerful and flexible but require close attention to avoid creating unexpected results.

Managing JavaScript Files

The `ScriptTagHelper` class is the built-in tag helper for `script` elements and is used to manage the inclusion of JavaScript files in views using the attributes described in Table 26-4, which I describe in the sections that follow.

Table 26-4. The Built-in Tag Helper Attributes for script Elements

Name	Description
asp-src-include	This attribute is used to specify JavaScript files that will be included in the view.
asp-src-exclude	This attribute is used to specify JavaScript files that will be excluded from the view.
asp-append-version	This attribute is used for cache busting, as described in the “Understanding Cache Busting” sidebar.
asp-fallback-src	This attribute is used to specify a fallback JavaScript file to use if there is a problem with a content delivery network.
asp-fallback-src-include	This attribute is used to select JavaScript files that will be used if there is a content delivery network problem.
asp-fallback-src-exclude	This attribute is used to exclude JavaScript files to prevent their use when there is a content delivery network problem.
asp-fallback-test	This attribute is used to specify a fragment of JavaScript that will be used to determine whether JavaScript code has been correctly loaded from a content delivery network.

Selecting JavaScript Files

The `asp-src-include` attribute is used to include JavaScript files in a view using globbing patterns. Globbing patterns support a set of wildcards that are used to match files, and Table 26-5 describes the most common globbing patterns.

Table 26-5. Common Globbing Patterns

Pattern	Example	Description
?	js/src?.js	This pattern matches any single character except /. The example matches any file contained in the js directory whose name is src, followed by any character, followed by .js, such as js/src1.js and js/srcX.js but not js/src123.js or js/mydir/src1.js.
*	js/*.js	This pattern matches any number of characters except /. The example matches any file contained in the js directory with the .js file extension, such as js/src1.js and js/src123.js but not js/mydir/src1.js.
**	js/**/*.js	This pattern matches any number of characters including /. The example matches any file with the .js extension that is contained within the js directory or any subdirectory, such as /js/src1.js and /js/mydir/src1.js.

Globbing is a useful way of ensuring that a view includes the JavaScript files that the application requires, even when the exact path to the file changes, which usually happens when the version number is included in the file name or when a package adds additional files.

Listing 26-9 uses the `asp-src-include` attribute to include all the JavaScript files in the `wwwroot/lib/jquery` folder, which is the location of the jQuery package installed with the command in Listing 26-4.

Listing 26-9. Selecting JS Files in the `_SimpleLayout.cshtml` File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
  <title>@ViewBag.Title</title>
  <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
  <script asp-src-include="lib/jquery/**/*.js"></script>
</head>
<body>
  <div class="m-2">
    @RenderBody()
  </div>
</body>
</html>
```

Patterns are evaluated within the `wwwroot` folder, and the pattern I used locates any file with the `js` file extension, regardless of its location within the `wwwroot` folder; this means that any JavaScript package added to the project will be included in the HTML sent to the client.

Use a browser to request `http://localhost:5000/home/list` and examine the HTML sent to the browser. You will see the single `script` element in the layout has been transformed into a `script` element for each JavaScript file, like this:

```
...
<head>
  <title></title>
  <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet">
  <script src="/lib/jquery/core.js"></script>
  <script src="/lib/jquery/jquery.js"></script>
  <script src="/lib/jquery/jquery.min.js"></script>
  <script src="/lib/jquery/jquery.slim.js"></script>
  <script src="/lib/jquery/jquery.slim.min.js"></script>
</head>
...
```

If you are using Visual Studio, you may not have realized that the jQuery packages contain so many JavaScript files because Visual Studio hides them in the Solution Explorer. To reveal the full contents of the client-side package folders, you can either expand the individual nested entries in the Solution Explorer window or disable file nesting by clicking the button at the top of the Solution Explorer window, as shown in Figure 26-5. (Visual Studio Code does not nest files.)

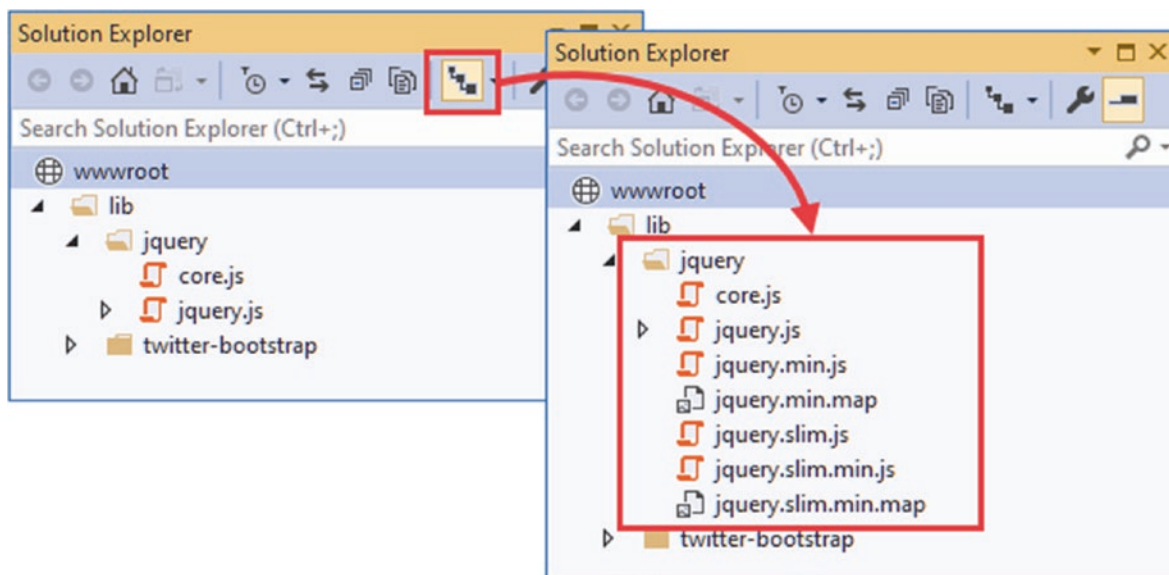


Figure 26-5. Disabling file nesting in the Visual Studio Solution Explorer

UNDERSTANDING SOURCE MAPS

JavaScript files are minified to make them smaller, which means they can be delivered to the client faster and using less bandwidth. The minification process removes all the whitespace from the file and renames functions and variables so that meaningful names such as `myHelpfullyNamedFunction` will be represented by a smaller number of characters, such as `x1`. When using the browser's JavaScript debugger to track down problems in your minified code, names like `x1` make it almost impossible to follow progress through the code.

The files that have the `map` file extension are *source maps*, which browsers use to help debug minified code by providing a map between the minified code and the developer-readable, unminified source file. When you open the browser's F12 developer tools, the browser will automatically request source maps and use them to help debug the application's client-side code.

Narrowing the Globbing Pattern

No application would require all the files selected by the pattern in Listing 26-9. Many packages include multiple JavaScript files that contain similar content, often removing less popular features to save bandwidth. The jQuery package includes the `jquery.slim.js` file, which contains the same code as the `jquery.js` file but without the features that handle asynchronous HTTP requests and animation effects. (There is also a `core.js` file, but this is included in the package by error and should be ignored.)

Each of these files has a counterpart with the `min.js` file extension, which denotes a minified file. Minification reduces the size of a JavaScript file by removing all whitespace and renaming functions and variables to use shorter names.

Only one JavaScript file is required for each package and if you only require the minified versions, which will be the case in most projects, then you can restrict the set of files that the globbing pattern matches, as shown in Listing 26-10.

Listing 26-10. Selecting Minified Files in the `_SimpleLayout.cshtml` File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
  <title>@ViewBag.Title</title>
  <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
  <script asp-src-include="lib/jquery**/*.min.js"></script>
</head>
<body>
  <div class="m-2">
    @RenderBody()
  </div>
</body>
</html>
```

Use a browser to request `http://localhost:5000/home/list` again and examine the HTML sent by the application. You will see that only the minified files have been selected.

```
...
<head>
  <title></title>
  <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet">
  <script src="/lib/jquery/jquery.min.js"></script>
  <script src="/lib/jquery/jquery.slim.min.js"></script>
</head>
...
```

Narrowing the pattern for the JavaScript files has helped, but the browser will still end up with the normal and slim versions of jQuery and the bundled and unbundled versions of the Bootstrap JavaScript files. To narrow the selection further, I can include `slim` in the pattern, as shown in Listing 26-11.

Listing 26-11. Narrowing the Focus in the `_SimpleLayout.cshtml` File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
  <title>@ViewBag.Title</title>
  <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
  <script asp-src-include="lib/jquery**/*.slim.min.js"></script>
</head>
<body>
  <div class="m-2">
    @RenderBody()
  </div>
</body>
</html>
```

Use the browser to request `http://localhost:5000/home/list` and examine the HTML the browser receives. The script element has been transformed like this:

```
...
<head>
  <title></title>
  <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet">
  <script src="/lib/jquery/jquery.slim.min.js"></script>
</head>
...
```

Only one version of the jQuery file will be sent to the browser while preserving the flexibility for the location of the file.

Excluding Files

Narrowing the pattern for the JavaScript files helps when you want to select a file whose name contains a specific term, such as `slim`. It isn't helpful when the file you want doesn't have that term, such as when you want the full version of the minified file. Fortunately, you can use the `asp-src-exclude` attribute to remove files from the list matched by the `asp-src-include` attribute, as shown in Listing 26-12.

Listing 26-12. Excluding Files in the `_SimpleLayout.cshtml` File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
  <title>@ViewBag.Title</title>
  <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
  <script asp-src-include="/lib/jquery/**/*.min.js"
        asp-src-exclude="**.slim.**">
  </script>
</head>
<body>
  <div class="m-2">
    @RenderBody()
  </div>
</body>
</html>
```

If you use the browser to request `http://localhost:5000/home/list` and examine the HTML response, you will see that the script element links only to the full minified version of the jQuery library, like this:

```
...
<head>
  <title></title>
  <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet">
  <script src="/lib/jquery/jquery.min.js"></script>
</head>
...
```

UNDERSTANDING CACHE BUSTING

Static content, such as images, CSS stylesheets, and JavaScript files, is often cached to stop requests for content that rarely changes from reaching the application servers. Caching can be done in different ways: the browser can be told to cache content by the server, the application can use cache servers to supplement the application servers, or the content can be distributed using a content delivery network. Not all caching will be under your control. Large corporations, for example, often install caches to reduce their bandwidth demands since a substantial percentage of requests tend to go to the same sites or applications.

One problem with caching is that clients don't immediately receive new versions of static files when you deploy them because their requests are still being serviced by previously cached content. Eventually, the cached content will expire, and the new content will be used, but that leaves a period where the dynamic content generated by the application's controllers is out of step with the static content being delivered by the caches. This can lead to layout problems or unexpected application behavior, depending on the content that has been updated.

Addressing this problem is called *cache busting*. The idea is to allow caches to handle static content but immediately reflect any changes that are made at the server. The tag helper classes support cache busting by adding a query string to the URLs for static content that includes a checksum that acts as a version number. For JavaScript files, for example, the `ScriptTagHelper` class supports cache busting through the `asp-append-version` attribute, like this:

```
...
<script asp-src-include="/lib/jquery/**/*.min.js"
        asp-src-exclude="**.slim.**" asp-append-version="true">
</script>
...
```

Enabling the cache busting feature produces an element like this in the HTML sent to the browser:

```
...
<script src="/lib/jquery/dist/jquery.min.js?v=3zRSQ1HF-ocUiVcdv9yKTXqM"></script>
...
```

The same version number will be used by the tag helper until you change the contents of the file, such as by updating a JavaScript library, at which point a different checksum will be calculated. The addition of the version number means that each time you change the file, the client will request a different URL, which caches treat as a request for new content that cannot be satisfied with the previously cached content and pass on to the application server. The content is then cached as normal until the next update, which produces another URL with a different version.

Working with Content Delivery Networks

Content delivery networks (CDNs) are used to offload requests for application content to servers that are closer to the user. Rather than requesting a JavaScript file from your servers, the browser requests it from a hostname that resolves to a geographically local server, which reduces the amount of time required to load files and reduces the amount of bandwidth you have to provision for your application. If you have a large, geographically disbursed set of users, then it can make commercial sense to sign up to a CDN, but even the smallest and simplest application can benefit from using the free CDNs operated by major technology companies to deliver common JavaScript packages, such as jQuery.

For this chapter, I am going to use CDNJS, which is the same CDN used by the Library Manager tool to install client-side packages in the ASP.NET Core project. You can search for packages at <https://cdnjs.com>; for jQuery 3.4.1, which is the package and version installed in Listing 26-4, there are six CDNJS URLs.

- <https://cdnjs.cloudflare.com/ajax/libs/jquery/3.4.1/jquery.js>
- <https://cdnjs.cloudflare.com/ajax/libs/jquery/3.4.1/jquery.min.js>
- <https://cdnjs.cloudflare.com/ajax/libs/jquery/3.4.1/jquery.min.map>
- <https://cdnjs.cloudflare.com/ajax/libs/jquery/3.4.1/jquery.slim.js>
- <https://cdnjs.cloudflare.com/ajax/libs/jquery/3.4.1/jquery.slim.min.js>
- <https://cdnjs.cloudflare.com/ajax/libs/jquery/3.4.1/jquery.slim.min.map>

These URLs provide the regular JavaScript file, the minified JavaScript file, and the source map for the minified file for both the full and slim versions of jQuery. (There is also a URL for the `core.js` file, but, as noted earlier, this file is not used and will be removed from future jQuery releases.)

The problem with CDNs is that they are not under your organization's control, and that means they can fail, leaving your application running but unable to work as expected because the CDN content isn't available. The `ScriptTagHelper` class provides the ability to fall back to local files when the CDN content cannot be loaded by the client, as shown in Listing 26-13.

Listing 26-13. Using CDN Fallback in the `_SimpleLayout.cshtml` File in the Views/Shared Folder

```

<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.4.1/jquery.min.js"
        asp-fallback-src="/lib/jquery/jquery.min.js"
        asp-fallback-test="window.jQuery">
    </script>
</head>
<body>
    <div class="m-2">
        @RenderBody()
    </div>
</body>
</html>

```

The `src` attribute is used to specify the CDN URL. The `asp-fallback-src` attribute is used to specify a local file that will be used if the CDN is unable to deliver the file specified by the regular `src` attribute. To figure out whether the CDN is working, the `asp-fallback-test` attribute is used to define a fragment of JavaScript that will be evaluated at the browser. If the fragment evaluates as `false`, then the fallback files will be requested.

■ **Tip** The `asp-fallback-src-include` and `asp-fallback-src-exclude` attributes can be used to select the local files with globbing patterns. However, given that CDN script elements select a single file, I recommend using the `asp-fallback-src` attribute to select the corresponding local file, as shown in the example.

Use a browser to request `http://localhost:5000/home/list`, and you will see that the HTML response contains two script elements, like this:

```

...
<head>
    <title></title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet">
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.4.1/jquery.min.js"></script>
    <script>
        (window.jQuery||document.write("\u003Cscript
            src=\u0022lib/jquery/jquery.min.js\u0022\u003E\u003C/script\u003E"));
    </script>
</head>
...

```

The first script element requests the JavaScript file from the CDN. The second script element evaluates the JavaScript fragment specified by the `asp-fallback-test` attribute, which checks to see whether the first script element has worked. If the fragment evaluates to `true`, then no action is taken because the CDN worked. If the fragment evaluates to `false`, a new script element is added to the HTML document that instructs the browser to load the JavaScript file from the fallback URL.

It is important to test your fallback settings because you won't find out if they fail until the CDN has stopped working and your users cannot access your application. The simplest way to check the fallback is to change the name of the file specified by the `src` attribute to something that you know doesn't exist (I append the word `FAIL` to the file name) and then look at the network requests that the browser makes using the F12 developer tools. You should see an error for the CDN file followed by a request for the fallback file.

■ **Caution** The CDN fallback feature relies on browsers loading and executing the contents of `script` elements synchronously and in the order in which they are defined. There are a number of techniques in use to speed up JavaScript loading and execution by making the process asynchronous, but these can lead to the fallback test being performed before the browser has retrieved a file from the CDN and executed its contents, resulting in requests for the fallback files even when the CDN is working perfectly and defeating the use of a CDN in the first place. Do not mix asynchronous script loading with the CDN fallback feature.

Managing CSS Stylesheets

The `LinkTagHelper` class is the built-in tag helper for `link` elements and is used to manage the inclusion of CSS style sheets in a view. This tag helper supports the attributes described in Table 26-6, which I demonstrate in the following sections.

Table 26-6. *The Built-in Tag Helper Attributes for link Elements*

Name	Description
<code>asp-href-include</code>	This attribute is used to select files for the <code>href</code> attribute of the output element.
<code>asp-href-exclude</code>	This attribute is used to exclude files from the <code>href</code> attribute of the output element.
<code>asp-append-version</code>	This attribute is used to enable cache busting, as described in the “Understanding Cache Busting” sidebar.
<code>asp-fallback-href</code>	This attribute is used to specify a fallback file if there is a problem with a CDN.
<code>asp-fallback-href-include</code>	This attribute is used to select files that will be used if there is a CDN problem.
<code>asp-fallback-href-exclude</code>	This attribute is used to exclude files from the set that will be used when there is a CDN problem.
<code>asp-fallback-href-test-class</code>	This attribute is used to specify the CSS class that will be used to test the CDN.
<code>asp-fallback-href-test-property</code>	This attribute is used to specify the CSS property that will be used to test the CDN.
<code>asp-fallback-href-test-value</code>	This attribute is used to specify the CSS value that will be used to test the CDN.

Selecting Stylesheets

The `LinkTagHelper` shares many features with the `ScriptTagHelper`, including support for globbing patterns to select or exclude CSS files so they do not have to be specified individually. Being able to accurately select CSS files is as important as it is for JavaScript files because stylesheets can come in regular and minified versions and support source maps. The popular Bootstrap package, which I have been using to style HTML elements throughout this book, includes its CSS stylesheets in the `wwwroot/lib/twitter-bootstrap/css` folder. These will be visible in Visual Studio Code, but you will have to expand each item in the Solution Explorer or disable nesting to see them in the Visual Studio Solution Explorer, as shown in Figure 26-6.

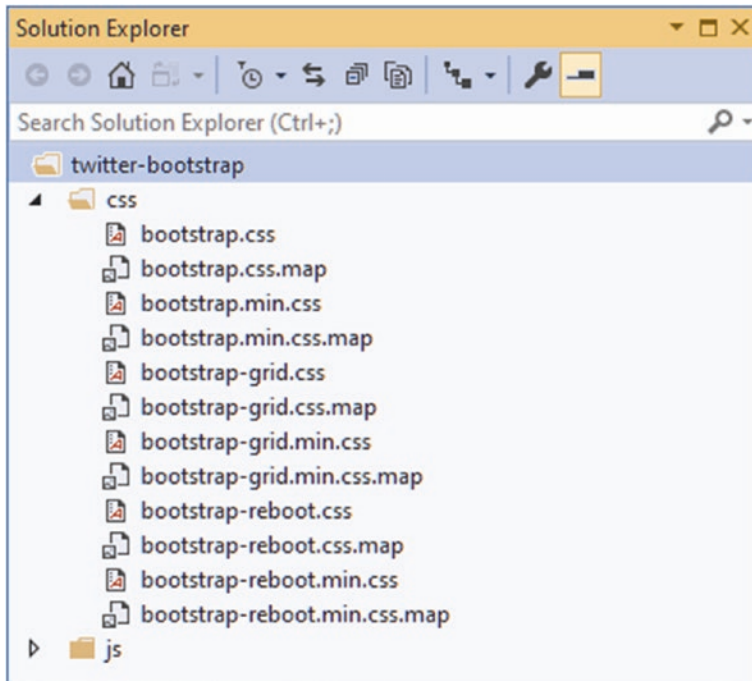


Figure 26-6. The Bootstrap CSS files

The `bootstrap.css` file is the regular stylesheet, the `bootstrap.min.css` file is the minified version, and the `bootstrap.css.map` file is a source map. The other files contain subsets of the CSS features to save bandwidth in applications that don't use them.

Listing 26-14 replaces the regular link element in the layout with one that uses the `asp-href-include` and `asp-href-exclude` attributes. (I removed the script element for jQuery, which is no longer required.)

Listing 26-14. Selecting a Stylesheet in the `_SimpleLayout.cshhtml` File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
  <title>@ViewBag.Title</title>
  <link asp-href-include="/lib/twitter-bootstrap/css/*.min.css"
        asp-href-exclude="**/*-reboot*,**/*-grid*" rel="stylesheet" />
</head>
<body>
  <div class="m-2">
    @RenderBody()
  </div>
</body>
</html>
```

The same attention to detail is required as when selecting JavaScript files because it is easy to generate link elements for multiple versions of the same file or files that you don't want.

Working with Content Delivery Networks

The `LinkTag` helper class provides a set of attributes for falling back to local content when a CDN isn't available, although the process for testing to see whether a stylesheet has loaded is more complex than testing for a JavaScript file. Listing 26-15 uses the CDNJS URL for the Bootstrap CSS stylesheet.

Listing 26-15. Using a CDN for CSS in the _SimpleLayout.cshhtml File in the Views/Home Folder

```

<!DOCTYPE html>
<html>
<head>
  <title>@ViewBag.Title</title>
  <link href="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/4.3.1/css/bootstrap.min.css"
    asp-fallback-href="/lib/twitter-bootstrap/css/bootstrap.min.css"
    asp-fallback-test-class="btn"
    asp-fallback-test-property="display"
    asp-fallback-test-value="inline-block"
    rel="stylesheet" />
</head>
<body>
  <div class="m-2">
    @RenderBody()
  </div>
</body>
</html>

```

The href attribute is used to specify the CDN URL, and I have used the asp-fallback-href attribute to select the file that will be used if the CDN is unavailable. Testing whether the CDN works, however, requires the use of three different attributes and an understanding of the CSS classes defined by the CSS stylesheet that is being used.

Use a browser to request <http://localhost:5000/home/list> and examine the HTML elements in the response. You will see that the link element from the layout has been transformed into three separate elements, like this:

```

...
<head>
  <title></title>
  <link href="https://cdnjs.cloudflare.com/.../bootstrap.min.css" rel="stylesheet">
  <meta name="x-stylesheet-fallback-test" content="" class="btn">
  <script>
    ! function(a, b, c, d) {
      var e, f = document,
          g = f.getElementsByTagName("SCRIPT"),
          h = g[g.length-1].previousElementSibling,
          i = f.defaultView && f.defaultView.getComputedStyle ?
            f.defaultView.getComputedStyle(h) : h.currentStyle;
      if (i && i[a] !== b)
        for (e = 0; e < c.length; e++)
          f.write('<link href="' + c[e] + '" ' + d + "/>");
    }("display", "inline-block", ["/lib/twitter-bootstrap/css/bootstrap.min.css"],
      "rel=\u0022stylesheet\u0022 ");
  </script>
</head>
...

```

To make the transformation easier to understand, I have formatted the JavaScript code and shortened the URL.

The first element is a regular link whose href attribute specifies the CDN file. The second element is a meta element, which specifies the class from the asp-fallback-test-class attribute in the view. I specified the btn class in the listing, which means that an element like this is added to the HTML sent to the browser:

```
<meta name="x-stylesheet-fallback-test" content="" class="btn">
```

The CSS class that you specify must be defined in the stylesheet that will be loaded from the CDN. The btn class that I specified provides the basic formatting for Bootstrap button elements.

The `asp-fallback-test-property` attribute is used to specify a CSS property that is set when the CSS class is applied to an element, and the `asp-fallback-test-value` attribute is used to specify the value that it will be set to.

The `script` element created by the tag helper contains JavaScript code that adds an element to the specified class and then tests the value of the CSS property to determine whether the CDN stylesheet has been loaded. If not, a `link` element is created for the fallback file. The Bootstrap `btn` class sets the `display` property to `inline-block`, and this provides the test to see whether the browser has been able to load the Bootstrap stylesheet from the CDN.

Tip The easiest way to figure out how to test for third-party packages like Bootstrap is to use the browser’s F12 developer tools. To determine the test in Listing 26-15, I assigned an element to the `btn` class and then inspected it in the browser, looking at the individual CSS properties that the class changes. I find this easier than trying to read through long and complex style sheets.

Working with Image Elements

The `ImageTagHelper` class is used to provide cache busting for images through the `src` attribute of `img` elements, allowing an application to take advantage of caching while ensuring that modifications to images are reflected immediately. The `ImageTagHelper` class operates in `img` elements that define the `asp-append-version` attribute, which is described in Table 26-7 for quick reference.

Table 26-7. The Built-in Tag Helper Attribute for Image Elements

Name	Description
<code>asp-append-version</code>	This attribute is used to enable cache busting, as described in the “Understanding Cache Busting” sidebar.

In Listing 26-16, I have added an `img` element to the shared layout for the city skyline image that I added to the project at the start of the chapter. I have also reset the `link` element to use a local file for brevity.

Listing 26-16. Adding an Image in the `_SimpleLayout.cshtml` File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
  <title>@ViewBag.Title</title>
  <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
  <div class="m-2">
    
    @RenderBody()
  </div>
</body>
</html>
```

Use a browser to request `http://localhost:5000/home/list`, which will produce the response shown in Figure 26-7.

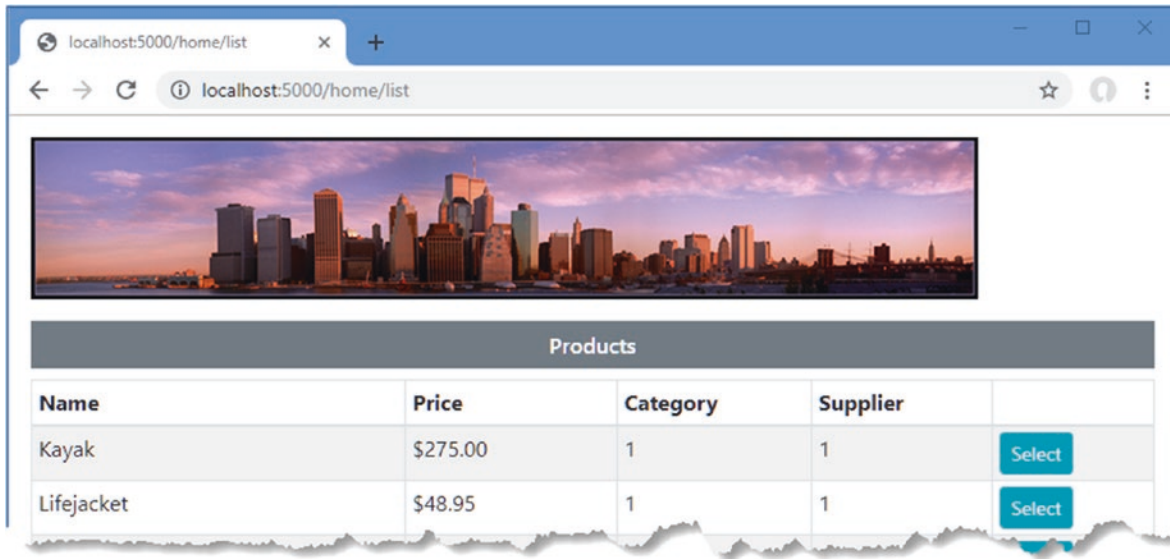


Figure 26-7. Using an image

Examine the HTML response, and you will see that the URL used to request the image file includes a version checksum, like this:

```
...

...
```

The addition of the checksum ensures that any changes to the file will pass through any caches, avoiding stale content.

Using the Data Cache

The `CacheTagHelper` class allows fragments of content to be cached to speed up rendering of views or pages. The content to be cached is denoted using the `cache` element, which is configured using the attributes shown in Table 26-8.

■ **Note** Caching is a useful tool for reusing sections of content so they don't have to be generated for every request. But using caching effectively requires careful thought and planning. While caching can improve the performance of an application, it can also create odd effects, such as users receiving stale content, multiple caches containing different versions of content, and update deployments that are broken because content cached from the previous version of the application is mixed with content from the new version. Don't enable caching unless you have a clearly defined performance problem to resolve, and make sure you understand the impact that caching will have.

Table 26-8. *The Built-in Tag Helper Attributes for cache Elements*

Name	Description
enabled	This bool attribute is used to control whether the contents of the cache element are cached. Omitting this attribute enables caching.
expires-on	This attribute is used to specify an absolute time at which the cached content will expire, expressed as a DateTime value.
expires-after	This attribute is used to specify a relative time at which the cached content will expire, expressed as a TimeSpan value.
expires-sliding	This attribute is used to specify the period since it was last used when the cached content will expire, expressed as a TimeSpan value.
vary-by-header	This attribute is used to specify the name of a request header that will be used to manage different versions of the cached content.
vary-by-query	This attribute is used to specify the name of a query string key that will be used to manage different versions of the cached content.
vary-by-route	This attribute is used to specify the name of a routing variable that will be used to manage different versions of the cached content.
vary-by-cookie	This attribute is used to specify the name of a cookie that will be used to manage different versions of the cached content.
vary-by-user	This bool attribute is used to specify whether the name of the authenticated user will be used to manage different versions of the cached content.
vary-by	This attribute is evaluated to provide a key used to manage different versions of the content.
priority	This attribute is used to specify a relative priority that will be taken into account when the memory cache runs out of space and purges unexpired cached content.

Listing 26-17 replaces the `img` element from the previous section with content that contains timestamps.

Listing 26-17. Caching Content in the `_SimpleLayout.cshtml` File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
  <title>@ViewBag.Title</title>
  <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
  <div class="m-2">
    <h6 class="bg-primary text-white m-2 p-2">
      Uncached timestamp: @DateTime.Now.ToLongTimeString()
    </h6>
    <cache>
      <h6 class="bg-primary text-white m-2 p-2">
        Cached timestamp: @DateTime.Now.ToLongTimeString()
      </h6>
    </cache>
    @RenderBody()
  </div>
</body>
</html>
```

The `cache` element is used to denote a region of content that should be cached and has been applied to one of the `h6` elements that contains a timestamp. Use a browser to request `http://localhost:5000/home/list`, and both timestamps will be the same. Reload the browser, and you will see that the cached content is used for one of the `h6` elements and the timestamp doesn't change, as shown in Figure 26-8.

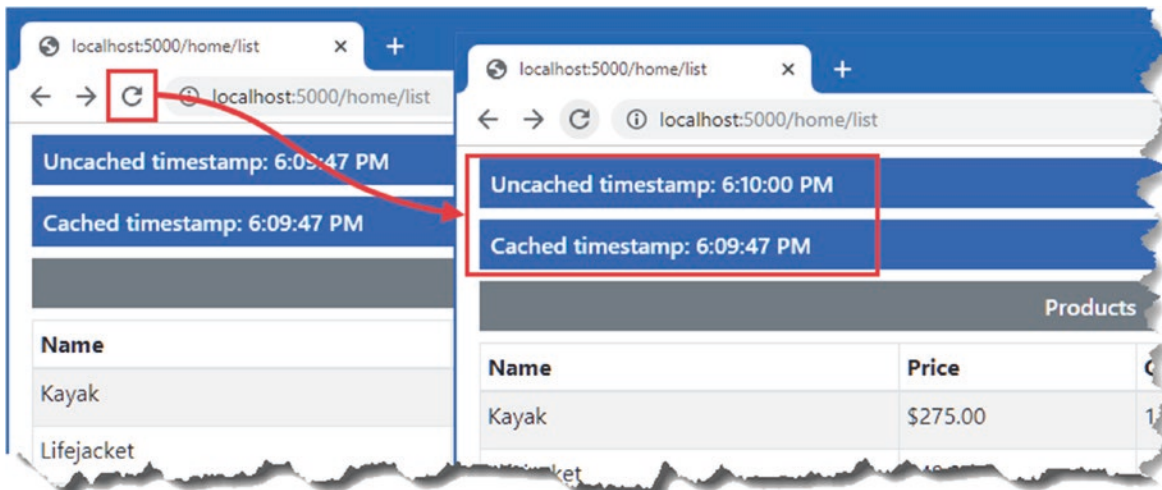


Figure 26-8. Using the caching tag helper

USING DISTRIBUTED CACHING FOR CONTENT

The cache used by the `CacheTagHelper` class is memory-based, which means that its capacity is limited by the available RAM and that each application server maintains a separate cache. Content will be ejected from the cache when there is a shortage of capacity available, and the entire contents are lost when the application is stopped or restarted.

The distributed-cache element can be used to store content in a shared cache, which ensures that all application servers use the same data and that the cache survives restarts. The distributed-cache element is configured with the same attributes as the cache element, as described in Table 26-8. See Chapter 17 for details of setting up a distributed cache.

Setting Cache Expiry

The `expires-*` attributes allow you to specify when cached content will expire, expressed either as an absolute time or a time relative to the current time, or to specify a duration during which the cached content isn't requested. In Listing 26-18, I have used the `expires-after` attribute to specify that the content should be cached for 15 seconds.

Listing 26-18. Setting Cache Expiry in the `_SimpleLayout.cshtml` File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
  <title>@ViewBag.Title</title>
  <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
  <div class="m-2">
    <h6 class="bg-primary text-white m-2 p-2">
      Uncached timestamp: @DateTime.Now.ToLongTimeString()
    </h6>
    <cache expires-after="@TimeSpan.FromSeconds(15)">
      <h6 class="bg-primary text-white m-2 p-2">
        Cached timestamp: @DateTime.Now.ToLongTimeString()
      </h6>
    </cache>
  </div>
  @RenderBody()
</body>
</html>
```

```

    </div>
</body>
</html>

```

Use a browser to request `http://localhost:5000/home/list` and then reload the page. After 15 seconds the cached content will expire, and a new section of content will be created.

Setting a Fixed Expiry Point

You can specify a fixed time at which cached content will expire using the `expires-on` attribute, which accepts a `DateTime` value, as shown in Listing 26-19.

Listing 26-19. Setting Cache Expiry in the `_SimpleLayout.cshtml` File in the Views/Shared Folder

```

<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="m-2">
        <h6 class="bg-primary text-white m-2 p-2">
            Uncached timestamp: @DateTime.Now.ToLongTimeString()
        </h6>
        <cache expires-on="@DateTime.Parse("2100-01-01")">
            <h6 class="bg-primary text-white m-2 p-2">
                Cached timestamp: @DateTime.Now.ToLongTimeString()
            </h6>
        </cache>
        @RenderBody()
    </div>
</body>
</html>

```

I have specified that that data should be cached until the year 2100. This isn't a useful caching strategy since the application is likely to be restarted before the next century starts, but it does illustrate how you can specify a fixed point in the future rather than expressing the expiry point relative to the moment when the content is cached.

Setting a Last-Used Expiry Period

The `expires-sliding` attribute is used to specify a period after which content is expired if it hasn't been retrieved from the cache. In Listing 26-20, I have specified a sliding expiry of 10 seconds.

Listing 26-20. Using a Sliding Expiry in the `_SimpleLayout.cshtml` File in the Views/Shared Folder

```

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

```

```

<body>
  <div class="m-2">
    <h6 class="bg-primary text-white m-2 p-2">
      Uncached timestamp: @DateTime.Now.ToLongTimeString()
    </h6>
    <cache expires-sliding="@TimeSpan.FromSeconds(10)">
      <h6 class="bg-primary text-white m-2 p-2">
        Cached timestamp: @DateTime.Now.ToLongTimeString()
      </h6>
    </cache>
    @RenderBody()
  </div>
</body>
</html>

```

You can see the effect of the `express-sliding` attribute by requesting `http://localhost:5000/home/list` and periodically reloading the page. If you reload the page within 10 seconds, the cached content will be used. If you wait longer than 10 seconds to reload the page, then the cached content will be discarded, the view component will be used to generate new content, and the process will begin anew.

Using Cache Variations

By default, all requests receive the same cached content. The `CacheTagHelper` class can maintain different versions of cached content and use them to satisfy different types of HTTP requests, specified using one of the attributes whose name begins with `vary-`. Listing 26-21 shows the use of the `vary-by-route` attribute to create cache variations based on the action value matched by the routing system.

Listing 26-21. Creating a Variation in the `_SimpleLayout.cshtml` File in the Views/Shared Folder

```

<!DOCTYPE html>
<html>
<head>
  <title>@ViewBag.Title</title>
  <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
  <div class="m-2">
    <h6 class="bg-primary text-white m-2 p-2">
      Uncached timestamp: @DateTime.Now.ToLongTimeString()
    </h6>
    <cache expires-sliding="@TimeSpan.FromSeconds(10)" vary-by-route="action">
      <h6 class="bg-primary text-white m-2 p-2">
        Cached timestamp: @DateTime.Now.ToLongTimeString()
      </h6>
    </cache>
    @RenderBody()
  </div>
</body>
</html>

```

If you use two browser tabs to request `http://localhost:5000/home/index` and `http://localhost:5000/home/list`, you will see that each window receives its own cached content with its own expiration, since each request produces a different action routing value.

■ **Tip** If you are using Razor Pages, then you can achieve the same effect using `page` as the value matched by the routing system.

Using the Hosting Environment Tag Helper

The `EnvironmentTagHelper` class is applied to the custom `environment` element and determines whether a region of content is included in the HTML sent to the browser based on the hosting environment, which I described in Chapters 15 and 16. The `environment` element relies on the `names` attribute, which I have described in Table 26-9.

Table 26-9. *The Built-in Tag Helper Attribute for environment Elements*

Name	Description
<code>names</code>	This attribute is used to specify a comma-separated list of hosting environment names for which the content contained within the <code>environment</code> element will be included in the HTML sent to the client.

In Listing 26-22, I have added `environment` elements to the shared layout including different content in the view for the development and production hosting environments.

Listing 26-22. Using `environment` in the `_SimpleLayout.cshtml` File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="m-2">
        <environment names="development">
            <h2 class="bg-info text-white m-2 p-2">This is Development</h2>
        </environment>
        <environment names="production">
            <h2 class="bg-danger text-white m-2 p-2">This is Production</h2>
        </environment>
        @RenderBody()
    </div>
</body>
</html>
```

The `environment` element checks the current hosting environment name and either includes the content it contains or omits it (the `environment` element itself is always omitted from the HTML sent to the client). Figure 26-9 shows the output for the development and production environments. (See Chapter 15 for details of how to set the environment.)

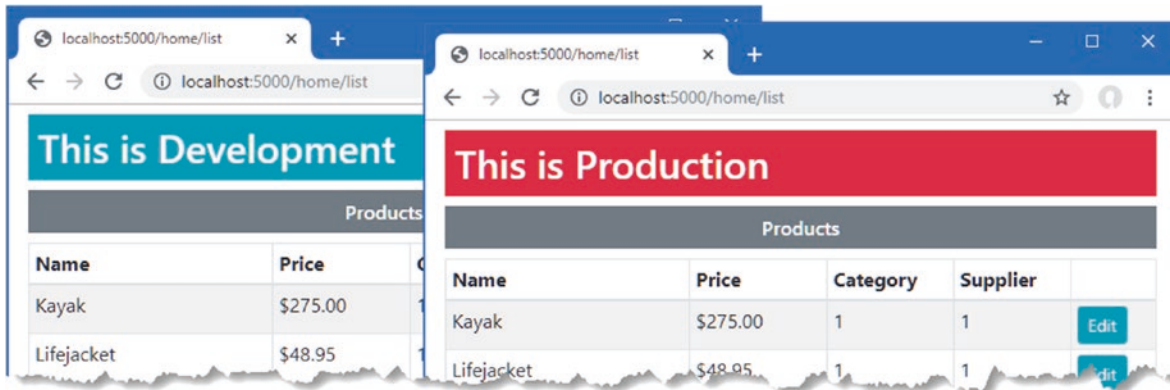


Figure 26-9. Managing content using the hosting environment

Summary

In this chapter, I described the basic built-in tag helpers and explained how they are used to transform anchor, link, script, and image elements. I also explained how to cache sections of content and how to render content based on the application's environment. In the next chapter, I describe the tag helpers that ASP.NET Core provides for working with HTML forms.