Lesson 05

HtmlHelper, Built-in Tag Helpers

ASP.NET Core

tongsreng TAL

ASP.NET Core

Contents

In this lesson we will discuss about using Views in View-Model-Controller Architecture in ASP.NET Core Web Application.

1 Html Helpers

(https://stephenwalther.com/archive/2009/03/03/chapter-6-understanding-html-helpers)

(https://www.c-sharpcorner.com/article/html-helpers-in-asp-net-mvc-5/)

You use HTML helpers in a view to render HTML content. An HTML helper, in most cases, is just a method that returns a string.

You can build an entire ASP.NET MVC application without using a single HTML helper. However, HTML helpers make your life as a developer easier. By taking advantage of helpers, you can build your views with far less work.

Helpers	**Description**
Html.ActionLink	Create <a> link to an action of a controller
Url.Action	Create full url for an action
Html helper for rendering form elements	
.BeginForm()	Start <form> tag
.CheckBox()	Render <input type="checkbox"> tag
·DropDownList()	Render <select> tag
.EndForm()	End </form> tag
.Hidden()	Render <input type="hidden"> tag
.ListBox()	Render <select multiple> tag
.Password()	Render <input type="password"> tag
.RadioButton()	Render <input type="radio"> tag
.TextArea()	Render <texterea> tag
.TextBox()	Render <input type="text"> tag

1.1 Html.ActionLink

Create <a> tag link to an action of a controller with some additional parameters. Examples:

```
…
    @Html.ActionLink("Go to Privacy page", "Privacy")
    @Html.ActionLink("Edit Record", "Edit", new {Id = 3})
    @Html.ActionLink("Edit Record", "Edit", "Customer", new {Id = 3})
…
```

Output:

```
…
<a href="/Home/Privacy">Go to Privacy page</a>
<a href="/Home/Edit/3">Edit Record</a>
<a href="/Customer/Edit/3">Edit Record</a>
```

```
...
```

1.2 Url.Action

Generate action relative URL. Example:

```
@Url.Action("Delete")
@Url.Action("Delete","Customer")
@Url.Action("Delete","Customer",new { Id = 3 })
```

Output:

```
/Home/Delete
/Customer/Delete
/Customer/Delete/3
```

1.3 Html Form rendering

There are many tags for form rendering but it all start from Html.BeginFrom() and end with Html.EndForm(). For example:

```
@{Html.BeginForm();}
...
@{Html.EndForm();}
```

Example:

```
@{Html.BeginForm();}

<fieldset>
    <legend>Register</legend>
    <p>
        <label for="FirstName">First Name:</label>
        @Html.TextBox("FirstName")
        @Html.ValidationMessage("FirstName", "*")
    </p>
    <p>
        <label for="LastName">Last Name:</label>
        @Html.TextBox("LastName")
        @Html.ValidationMessage("LastName", "*")
    </p>
    <p>
        <label for="Password">Password:</label>
        @Html.Password("Password")
        @Html.ValidationMessage("Password", "*")
    </p>
    <p>
        <label for="Password">Confirm Password:</label>
        @Html.Password("ConfirmPassword")
        @Html.ValidationMessage("ConfirmPassword", "*")
    </p>
    <p>
        <label for="Profile">Profile:</label>
        @Html.TextArea("Profile", new { cols = 60, rows = 10 })
    </p>
    <p>
        @Html.CheckBox("ReceiveNewsletter")
        <label for="ReceiveNewsletter" style="display:inline">Receive Newsletter?</label>
    </p>
    <p>
        <input type="submit" value="Register" />
    </p>
</fieldset>

@{Html.EndForm();}
```

Output:

```
<form action="/" method="post">
  <fieldset>
    <legend>Register</legend>
    <p>
      <label for="FirstName">First Name:</label>
      <input id="FirstName" name="FirstName" type="text" value="" />
      <span class="field-validation-valid" data-valmsg-for="FirstName" data-valmsg-replace="false">*</span>
    </p>
    <p>
      <label for="LastName">Last Name:</label>
      <input id="LastName" name="LastName" type="text" value="" />
      <span class="field-validation-valid" data-valmsg-for="LastName" data-valmsg-replace="false">*</span>
    </p>
    <p>
      <label for="Password">Password:</label>
      <input id="Password" name="Password" type="password" />
      <span class="field-validation-valid" data-valmsg-for="Password" data-valmsg-replace="false">*</span>
    </p>
    <p>
      <label for="Password">Confirm Password:</label>
      <input id="ConfirmPassword" name="ConfirmPassword" type="password" />
      <span class="field-validation-valid" data-valmsg-for="ConfirmPassword" data-valmsg-replace="false">*
      </span>
    </p>
    <p>
       <label for="Profile">Profile:</label>
       <textarea cols="60" id="Profile" name="Profile" rows="10">
</textarea>
    </p>
    <p>
    <input id="ReceiveNewsletter" name="ReceiveNewsletter" type="checkbox" value="true" />
    <label for="ReceiveNewsletter" style="display:inline">Receive Newsletter?</label>
   </p>
   <p>
    <input type="submit" value="Register" />
   </p>
  </fieldset>

  <input name="__RequestVerificationToken" type="hidden" value="..." />
  <input name="ReceiveNewsletter" type="hidden" value="false" />
</form>
```

1.3.1 Rendering DropDownList

You can use the Html.DropDownList() helper to render a set of database records in an HTML <select> tag. You represent the set of database records with the SelectList class.

For example, the Index() action in Listing 3 creates an instance of the SelectList class that represents all of the customers from the Customers database table. You can pass the following parameters to the constructor for a SelectList when creating a new SelectList:

- items – The items represented by the SelectList.
- dataValueField – The name of the property to associate with each item in the SelectList as the value of the item.
- dataTextField – The name of the property to display for each item in the SelectList as the label of the item.
- selectedValue – The item to select in the SelectList.

Class customer as below:

```
namespace WebAppHelper.Models
{
    public class Customer
    {
        public long Id { get; set; }
        public string Username { get; set; }
        public string Pwd { get; set; }
    }
}
```

We have list of customers in HomeController below:

```
...
        public IActionResult Index()
        {
            var customers = new Customer[] {
                new Customer{Id=1, Username="Sodarith", Pwd="qe9238es8f928e"},
                new Customer{Id=2, Username="Kakna", Pwd="se9238es8fyh80"},
                new Customer{Id=3, Username="Pheara", Pwd="ye9s38esbfn28M"}
            };
            ViewData["CustomerId"] = new SelectList(customers, "Id", "Username");
            return View();
        }
...
```
We want to display name of customers but when user select a name in the list, the value selected behind is Id of customer. In this case, the dropdown code is:

```
@Html.DropDownList("CustomerId")
```

Output:

```
  <select id="CustomerId" name="CustomerId">
    <option value="1">Sodarith</option>
    <option value="2">Kakna</option>
    <option value="3">Pheara</option>
  </select>
```
You may want to display an option label such as “Select a customer”:

```
@Html.DropDownList("CustomerId", "Select a customer")
```

Output:

```
  <select id="CustomerId" name="CustomerId">
    <option value="">Select a customer</option>
    <option value="1">Sodarith</option>
    <option value="1">Kakna</option>
    <option value="1">Pheara</option>
  </select>
```
1.3.2 Encoding HTML content

We can encode HTML into plain text and display on browser. Example:

```
@Html.Encode("<h1>Hello</h1>")
```
Output:

```
&amp;lt;h1&amp;gt;Hello&amp;lt;/h1&amp;gt;
```

On server, we may want to decode back into HTML content:

```
HttpUtility.HtmlDecode("&amp;lt;h1&amp;gt;Hello&amp;lt;/h1&amp;gt;");
```
1.3.3 Using Anti-Forgery Tokens

There is a particular type of JavaScript injection attack that is called a Cross-Site Request Forgery (CSRF) attack. In a CSRF attack, a hacker takes advantage of the fact that you are logged into one website to steal or modify your information at another website.

To learn more about CSRF attacks, see the Wikipedia entry at http://en.wikipedia.org/wiki/Csrf. The example discussed in this section is based on the example described in the Wikipedia entry.

For example, imagine that you have an online bank account. The bank website identifies and authenticates you with a cookie. Now, imagine that you visit a forums website. This forums website enables users to post messages that contain images. An evil hacker has posted an image to the forums that looks like this:

```
<img src="http://www.BigBank.com/withdraw?amount =9999" />
```

Notice that the src attribute of this image tag points to a URL at the bank website.

When you view this message in your browser, $9,999 dollars is withdrawn from your bank account. The hacker is able to withdraw money from your bank account because the bank website uses a browser cookie to identify you. The hacker has hijacked your browser.

If you are creating the bank website, then you can prevent a CSRF attack by using the Html.AntiForgeryToken() helper.

For example, the view in Listing below uses the Html.AntiForgeryToken() helper.

```
@{Html.BeginForm(); }

@Html.AntiForgeryToken()

<fieldset>
   <legend>Fields</legend>
   <p>
      <label for="Amount">Amount:</label>
      @Html.TextBox("Amount")
   </p>
   <p>
      <input type="submit" value="Withdraw" />
   </p>
</fieldset>

@{Html.EndForm(); }
```

Output:

```
<form action="/" method="post">
   <fieldset>
      <legend>Fields</legend>
      <p>
         <label for="Amount">Amount:</label>
         <input id="Amount" name="Amount" type="text" value="" />
      </p>
      <p>
         <input type="submit" value="Withdraw" />
```

```
        </p>
      </fieldset>

      <input name="__RequestVerificationToken" type="hidden"
             value="CfDJ8Ci83L8wlslNjIiVAFV5-7xmusmT-mgYqXasVed-
u6Hf360ymSkNyXJYZO6i6k2uQBpJ33qCOYu2l2ZmipUXSp7wSZfo_mBE6x7Bt0GMpr6CmU69wV2KtIR1yzOp
AEYuJi72ca-M33xIS7X0Mk28v10" />
    </form>
```

The helper also creates a cookie that represents the random value. The value in the cookie is compared against the value in the hidden form field to determine whether a CSRF attack is being performed.

The Html.AntiForgeryToken() helper accepts the following optional parameters:

- **salt** – Enables you to add a cryptographic salt to the random value to increase the security of the anti-forgery token.
- **domain** – The domain associated with the anti-forgery cookie. The cookie is sent only when requests originate from this domain.
- **path** – The virtual path associated with the anti-forgery cookie. The cookie is sent only when requests originate from this path.

Generating the random value with the Html.AntiForgeryToken() helper is only half the story. To prevent CSRF attacks, you also must add a special attribute to the controller action that accepts the HTML form post. Example:

```
…
      [HttpPost]
      [ValidateAntiForgeryToken]
      public IActionResult Withdraw(decimal amount)
      {
         // Perform withdrawal
         return View();
      }
…
```

2 Built-in TagHelpers

This lesson creates a new WebApp project. To prepare for this lesson, create MVC project name WebApp with configure https and Enable RazorRuntimeCompilation in Startup.cs as shown in listing 26-1.

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

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace WebApp
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddControllersWithViews().AddRazorRuntimeCompilation();
            services.AddRazorPages().AddRazorRuntimeCompilation();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Home/Error");
                app.UseHsts();
            }
            app.UseHttpsRedirection();
            app.UseStaticFiles();

            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapRazorPages();
                endpoints.MapControllerRoute(
                    name: "default",
                    pattern: "{controller=Home}/{action=Index}/{id?}");
            });
        }
    }
}
```

Next, open Nuget Package manager to install packages: Microsoft.EntityFrameworkCore.Design, Microsoft.EntityFrameworkCore.Sqlite and Microsoft.EntityFrameworkCore.Proxies.

2.1 Prepare Data Model

The data model for this part of the book will consist of three related classes: Product, Supplier, and Category. Go to folder named Models and add to a class file named Category.cs, with the contents shown below.

```
using System.Collections.Generic;

namespace WebApp.Models
{
    public class Category
    {
        public long CategoryId { get; set; }
        public string Name { get; set; }
        public virtual IEnumerable<Product> Products { get; set; }
    }
}
```

Add a class called Supplier.cs to the Models folder and use it to define the class shown below.

```
using System.Collections.Generic;

namespace WebApp.Models
{
    public class Supplier
    {
        public long SupplierId { get; set; }
        public string Name { get; set; }
        public string City { get; set; }
        public virtual IEnumerable<Product> Products { get; set; }
    }
}
```

Next, add a class named Product.cs to the Models folder and use it to define the class shown below.

```
using System.ComponentModel.DataAnnotations.Schema;

namespace WebApp.Models
{
    public class Product
    {
        public long ProductId { get; set; }
        public string Name { get; set; }
        [Column(TypeName = "decimal(8, 2)")]
        public decimal Price { get; set; }
        public long CategoryId { get; set; }
        public virtual Category Category { get; set; }
        public long SupplierId { get; set; }
        public virtual Supplier Supplier { get; set; }
    }
}
```

Each of the three data model classes defines a key property whose value will be allocated by the database when new objects are stored. There are also navigation properties that will be used to query for related data so that it will be possible to query for all the products in a specific category, for example.

The Price property has been decorated with the Column attribute, which specifies the precision of the values that will be stored in the database. There isn't a one-to-one mapping between C# and SQL numeric types, and the Column attribute tells Entity Framework Core which SQL type should be used in the database to store Price values. In this case, the decimal(8, 2) type will allow a total of eight digits, including two following the decimal point.

To create the Entity Framework Core context class that will provide access to the database, add a file called DataContext.cs to the Models folder and add the code shown below.

```
using Microsoft.EntityFrameworkCore;

namespace WebApp.Models
{
    public class DataContext:DbContext
    {
        public DataContext(DbContextOptions<DataContext> opts) : base(opts) { }
        public DbSet<Product> Products { get; set; }
        public DbSet<Category> Categories { get; set; }
        public DbSet<Supplier> Suppliers { get; set; }
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Product>().ToTable("Products");
            modelBuilder.Entity<Category>().ToTable("Categories");
            modelBuilder.Entity<Supplier>().ToTable("Suppliers");
        }
    }
}
```

2.2 Preparing seed data

Add a class called SeedData.cs to the Models folder and add the code shown below to define the seed data that will be used to populate the database.

```
using Microsoft.EntityFrameworkCore;
using System.Linq;

namespace WebApp.Models
{
    public class DataSeed
    {
        public static void SeedDatabase(DataContext context)
        {
            context.Database.EnsureCreated();
            if (context.Products.Count() == 0 && context.Suppliers.Count() == 0
            && context.Categories.Count() == 0)
            {
                Supplier s1 = new Supplier
                { Name = "Splash Dudes", City = "San Jose" };
                Supplier s2 = new Supplier
                { Name = "Soccer Town", City = "Chicago" };
                Supplier s3 = new Supplier
                { Name = "Chess Co", City = "New York" };
                Category c1 = new Category { Name = "Watersports" };
                Category c2 = new Category { Name = "Soccer" };
                Category c3 = new Category { Name = "Chess" };
                context.Products.AddRange(
                new Product
                {
                    Name = "Kayak",
                    Price = 275,
                    Category = c1,
                    Supplier = s1
                }, new Product
                {
                    Name = "Lifejacket",
                    Price = 48.95m,
                    Category = c1,
                    Supplier = s1
                }, new Product
                {
                    Name = "Soccer Ball",
                    Price = 19.50m,
                    Category = c2,
                    Supplier = s2
                }, new Product
                {
                    Name = "Corner Flags",
                    Price = 34.95m,
                    Category = c2,
                    Supplier = s2
                }, new Product
                {
```

```
                Name = "Stadium",
                Price = 79500,
                Category = c2,
                Supplier = s2
            }, new Product
            {
                Name = "Thinking Cap",
                Price = 16,
                Category = c3,
                Supplier = s3
            }, new Product
            {
                Name = "Unsteady Chair",
                Price = 29.95m,
                Category = c3,
                Supplier = s3
            }, new Product
            {
                Name = "Human Chess Board",
                Price = 75,
                Category = c3,
                Supplier = s3
            }, new Product
            {
                Name = "Bling-Bling King",
                Price = 1200,
                Category = c3,
                Supplier = s3
            });
            context.SaveChanges();
        }
    }
}
}
```

The static SeedDatabase method ensures that all pending migrations have been applied to the database. If the database is empty, it is seeded with categories, suppliers, and products. Entity Framework Core will take care of mapping the objects into the tables in the database, and the key properties will be assigned automatically when the data is stored.

2.3 Configuring Entity Framework Core Services and Middleware

Make the changes to the Startup class shown below, which configure Entity Framework Core and set up the DataContext services that will be used throughout this part of the book to access the database.

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

namespace WebApp
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

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

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

            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapRazorPages();
                endpoints.MapControllerRoute(
                    name: "default",
                    pattern: "{controller=Home}/{action=Index}/{id?}");
            });
            DataSeed.SeedDatabase(context);
        }
    }
}
```

To define the connection string that will be used for the application's data, add the configuration settings shown below in the appsettings.json file. The connection string should be entered on a single line.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information",
      "Microsoft.EntityFrameworkCore": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "ProductConnection": "Data Source=MyData.db"
  }
}
```

In addition to the connection string, Listing above increases the logging detail for Entity Framework Core so that the SQL queries sent to the database are logged.

Update Controllers/HomeController.cs to list all products in Index action:

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

namespace WebApp.Controllers
{
    public class HomeController : Controller
    {
        private DataContext context;

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

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

        public IActionResult Privacy()
        {
            return View();
        }

        [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]
        public IActionResult Error()
        {
            return View(new ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });
        }
    }
}
```

Update Views/Home/Index.cshtml to display all products:

```
@model IEnumerable<Product>
@{
    ViewData["Title"] = "Home Page";
}

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

2.4 Enable Partial View

Create file _RowPartial.cshtml in Views/Home folder, with content as below.

```
@model Product
<tr>
    <td>@Model.Name</td>
    <td>@Model.Price.ToString("c")</td>
    <td>@Model.Category?.Name</td>
    <td>@Model.Supplier?.Name</td>
    <td></td>
</tr>
```

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

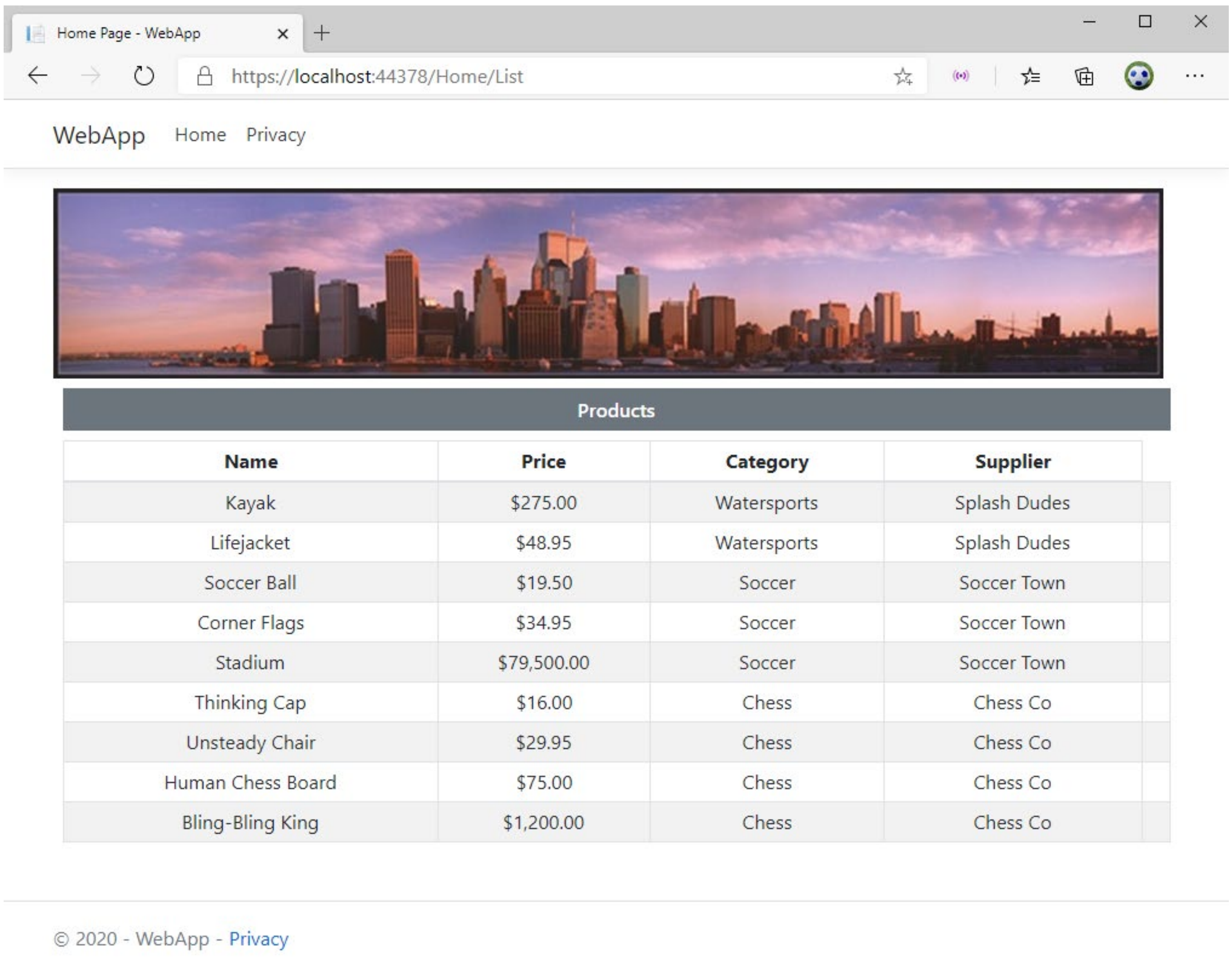
```
@model IEnumerable<Product>
@{
    ViewData["Title"] = "Home Page";
}

<div class="text-center">
    <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>Actions</th></tr>
            </thead>
            <tbody>
                @foreach (Product p in Model)
                {
                    <partial name="_RowPartial" model="p"/>
                }
            </tbody>
        </table>
    </div>
</div>
```

2.5 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 new_york.jpg. 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*


Home Page - WebApp

https://localhost:44378/Home/List

WebApp Home Privacy

Products

Name	Price	Category	Supplier
Kayak	$275.00	Watersports	Splash Dudes
Lifejacket	$48.95	Watersports	Splash Dudes
Soccer Ball	$19.50	Soccer	Soccer Town
Corner Flags	$34.95	Soccer	Soccer Town
Stadium	$79,500.00	Soccer	Soccer Town
Thinking Cap	$16.00	Chess	Chess Co
Unsteady Chair	$29.95	Chess	Chess Co
Human Chess Board	$75.00	Chess	Chess Co
Bling-Bling King	$1,200.00	Chess	Chess Co

© 2020 - WebApp - Privacy

Figure 26-2. *Running the example application*

2.6 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
@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
```

The `@addTagHelper` directive makes Tag Helpers available to the view. In this case, the view file is *Pages/_ViewImports.cshtml*, which by default is inherited by all files in the *Pages* folder and subfolders; making Tag Helpers available. The code above uses the wildcard syntax ("*") to specify that all Tag Helpers in the specified assembly (*Microsoft.AspNetCore.Mvc.TagHelpers*) will be available to every view file in the *Views* directory or subdirectory. The first parameter after `@addTagHelper` specifies the Tag Helpers to load (we are using "*" for all Tag Helpers), and the second parameter "Microsoft.AspNetCore.Mvc.TagHelpers" specifies the assembly containing the Tag Helpers. *Microsoft.AspNetCore.Mvc.TagHelpers* is the assembly for the built-in ASP.NET Core Tag Helpers.

```
@removeTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

The `@removeTagHelper` has the same two parameters as `@addTagHelper`, and it removes a Tag Helper that was previously added. For example, `@removeTagHelper` applied to a specific view removes the specified Tag Helper from the view. Using `@removeTagHelper` in a *Views/Folder/_ViewImports.cshtml* file removes the specified Tag Helper from all of the views in *Folder*.

2.7 Transforming Anchor Elements

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

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

Name	Description
asp-action	This attribute specifies the action method that the URL will target.
asp-controller	This attribute specifies the controller that the URL will target. If this attribute is omitted, then the URL will target the controller or page that rendered the current view.
asp-page	This attribute specifies the Razor Page that the URL will target.

asp-page-handler	This attribute specifies the Razor Page handler function that will process the request, as described in the end of this lesson.
asp-fragment	This attribute is used to specify the URL fragment (which appears after the # character).
asp-host	This attribute specifies the name of the host that the URL will target.
asp-protocol	This attribute specifies the protocol that the URL will use.
asp-route	This attribute specifies the name of the route that will be used to generate the URL.
asp-route-*	Attributes whose name begins with asp-route- are used to specify additional values for the URL so that the asp-route-id attribute is used to provide a value for the id segment to the routing system.
asp-all-route-data	This attribute provides values used for routing as a single value, rather than using individual attributes.

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

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

```
@model Product
<tr>
    <td>@Model.Name</td>
    <td>@Model.Price.ToString("c")</td>
    <td>@Model.Category?.Name</td>
    <td>@Model.Supplier?.Name</td>
    <td>
        <a asp-action="detail" 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.

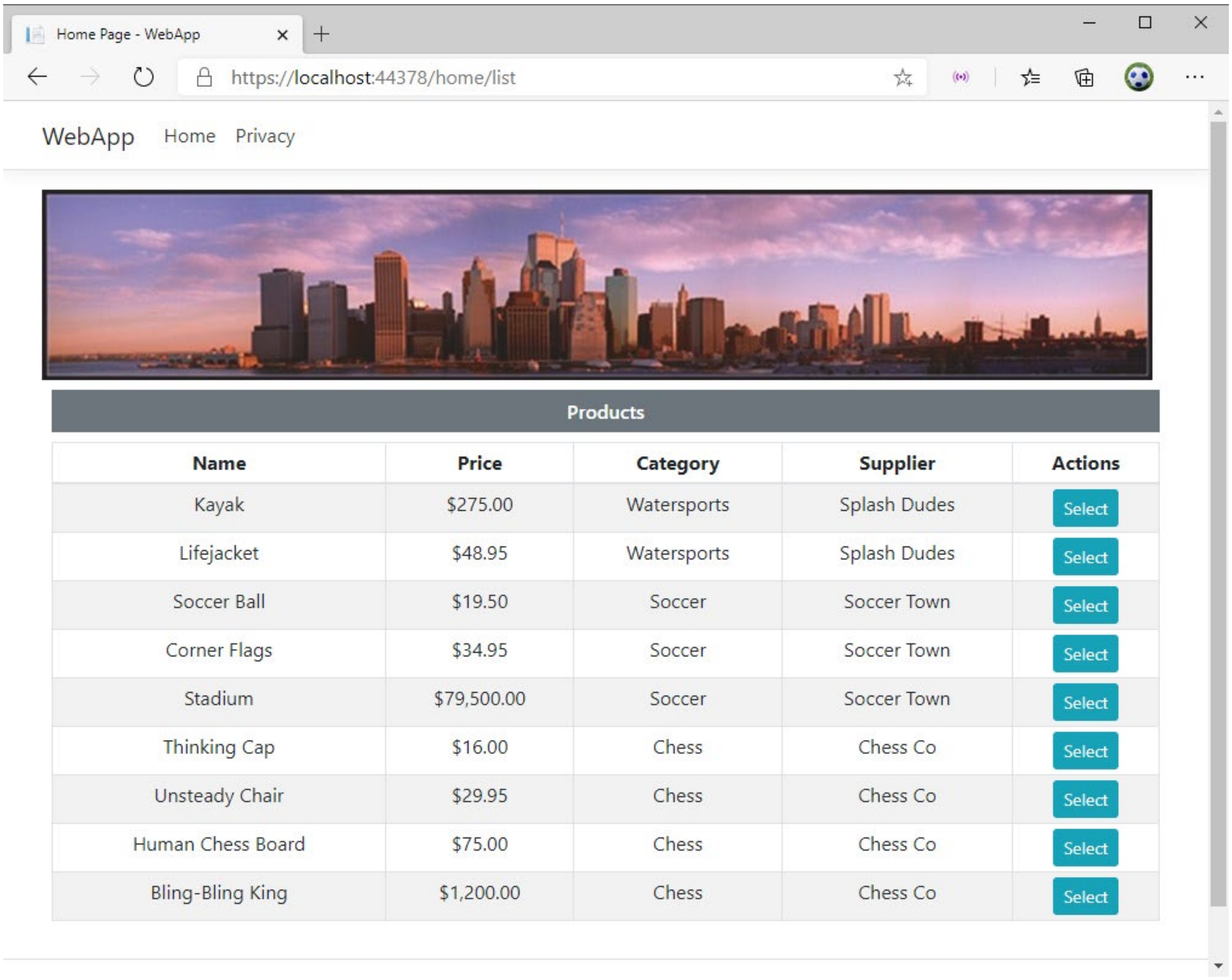
Add Detail Action in HomeController as below:

```
…
        public async Task<IActionResult> Detail(int Id)
        {
            return View(await context.Products.FindAsync(Id));
        }
…
```

Add view Detail.cshtml in Views/Home folder as below:

```
@model Product
@{
    ViewData["Title"] = "Product Detail";
}
<h6 class="bg-secondary text-white text-center m-2 p-2">Product @Model.Name</h6>
<div class="m-2">
    <table class="table table-sm table-striped table-bordered">
        <tbody>
            <tr><th>Id</th><td>@Model.ProductId</td></tr>
            <tr><th>Name</th><td>@Model.Name</td></tr>
            <tr><th>Price</th><td>@Model.Price.ToString("c")</td></tr>
            <tr><th>Category</th><td>@Model.Category.Name</td></tr>
            <tr><th>Supplier</th><td>@Model.Supplier.Name</td></tr>
        </tbody>
        <tfoot>
            <tr><td colspan="5"><a asp-action="list" asp-controller="home">List Products</a></td></tr>
        </tfoot>
    </table>
</div>
```

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.



Name	Price	Category	Supplier	Actions
Kayak	$275.00	Watersports	Splash Dudes	Select
Lifejacket	$48.95	Watersports	Splash Dudes	Select
Soccer Ball	$19.50	Soccer	Soccer Town	Select
Corner Flags	$34.95	Soccer	Soccer Town	Select
Stadium	$79,500.00	Soccer	Soccer Town	Select
Thinking Cap	$16.00	Chess	Chess Co	Select
Unsteady Chair	$29.95	Chess	Chess Co	Select
Human Chess Board	$75.00	Chess	Chess Co	Select
Bling-Bling King	$1,200.00	Chess	Chess Co	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.

2.8 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.

Create folder WebApp/Pages and create a subfolder named Suppliers.

Create a Razor Page named List.cshtml in Pages/Suppliers with content as below:

```
@page
@model WebApp.Pages.Suppliers.ListModel
@{
    ViewData["Title"] = "Suppliers";
}
<!DOCTYPE html>
<html>
<head>
    <link href="/lib/bootstrap/dist/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <h5 class="bg-primary text-white text-center m-2 p-2">Suppliers</h5>
    <ul class="list-group m-2">
        @foreach (WebApp.Models.Supplier s in Model.Suppliers)
        {
            <li class="list-group-item">@s.Name</li>
        }
    </ul>
    </body>
</html>
```

The code-behind of this view is List.cshtml.cs with content as below:

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

namespace WebApp.Pages.Suppliers
{
    public class ListModel : PageModel
    {
        private DataContext context;
        public IEnumerable<Supplier> Suppliers { get; set; }
        public ListModel(DataContext ctx)
        {
            context = ctx;
        }
        public void OnGet()
        {
            Suppliers = context.Suppliers;
        }
    }
}
```
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 Index.cshtml File in the Views/Home Folder

```
@model IEnumerable<Product>
@{
    ViewData["Title"] = "Home Page";
}
<div class="w-100"><img src="@Url.Content("~/images/new_york.jpg")"/></div>
<div class="text-center">
    <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>Actions</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>
</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="/suppliers/list">Suppliers</a>
```

```
...
```

Now, modify List.cshtml add attribute to @page directive like below:

```
@page "/lists/suppliers"
@model WebApp.Pages.Suppliers.ListModel
@{
   ViewData["Title"] = "Suppliers";
}
<!DOCTYPE html>
<html>
<head>
   <link href="/lib/bootstrap/dist/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
   <h5 class="bg-primary text-white text-center m-2 p-2">Suppliers</h5>
   <ul class="list-group m-2">
      @foreach (WebApp.Models.Supplier s in Model.Suppliers)
      {
         <li class="list-group-item">@s.Name</li>
      }
   </ul>
   </body>
</html>
```

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.

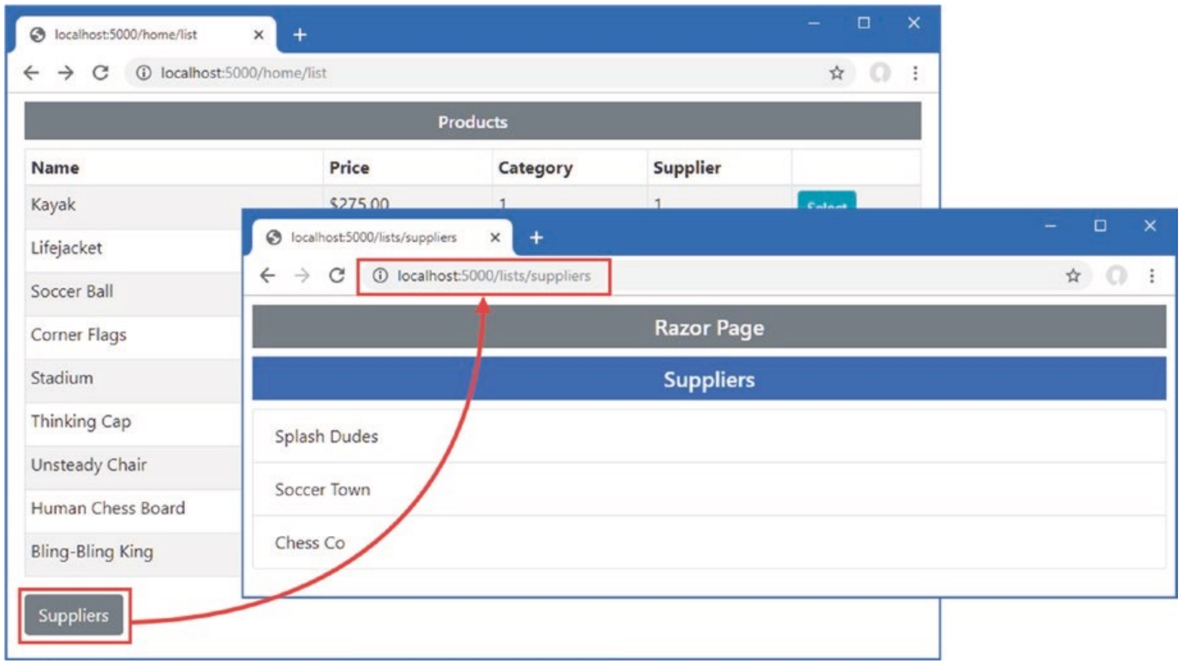


Figure 26-4. *Targeting a Razor Page with an anchor element*

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.

2.9 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.

2.9.1 Managing JavaScript Files

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

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

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

2.9.2 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.)

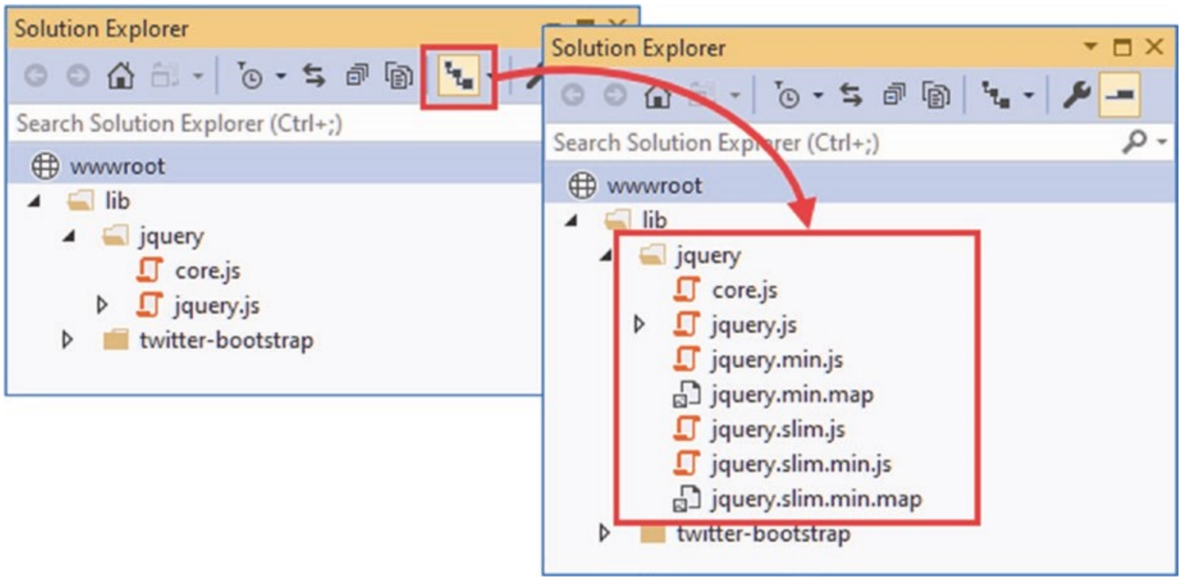


Figure 26-5. *Disabling file nesting in the Visual Studio Solution Explorer*

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.

2.9.3 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 _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.

2.9.3.1 *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.

2.9.4 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 aspfallback-test attribute is used to define a fragment of JavaScript that will be evaluated at the browser. If the fragment evaluates as false, then the fallback files will be requested.

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

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

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

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

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

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

2.9.5 Managing CSS Stylesheets

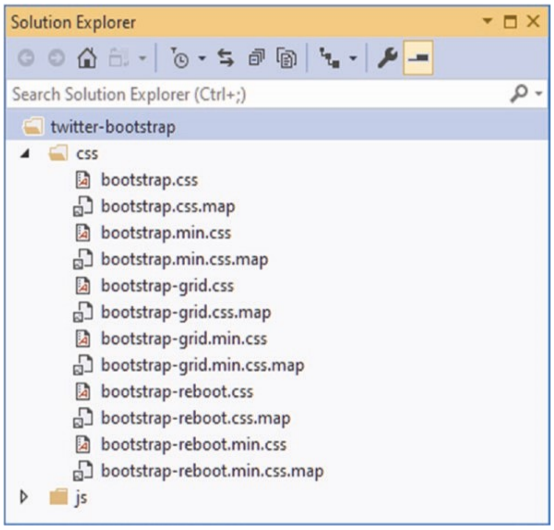
The LinkTagHelper class is the built-in tag helper for link elements and is used to manage the inclusion of CSS style sheets in a view. This tag helper supports the attributes described in Table 26-6, which I demonstrate in the following sections.

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

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

2.9.5.1 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/twitterbootstrap/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.



```
twitter-bootstrap
  css
    bootstrap.css
    bootstrap.css.map
    bootstrap.min.css
    bootstrap.min.css.map
    bootstrap-grid.css
    bootstrap-grid.css.map
    bootstrap-grid.min.css
    bootstrap-grid.min.css.map
    bootstrap-reboot.css
    bootstrap-reboot.css.map
    bootstrap-reboot.min.css
    bootstrap-reboot.min.css.map
  js
```

Figure 26-6. *The Bootstrap CSS files*

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

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

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

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

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

2.9.5.2 Working with Content Delivery Networks

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

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

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

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

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

```
...
<head>
   <title></title>
   <link href="https://cdnjs.cloudflare.com/.../bootstrap.min.css" rel="stylesheet">
   <meta name="x-stylesheet-fallback-test" content="" class="btn">
   <script>
    ! function(a, b, c, d) {
      var e, f = document,
      g = f.getElementsByTagName("SCRIPT"),
        h = g[g.length1].previousElementSibling,
        i = f.defaultView && f.defaultView.getComputedStyle ?
        f.defaultView.getComputedStyle(h) : h.currentStyle;
      if (i && i[a] !== b)
        for (e = 0; e < c.length; e++)
          f.write('<link href="' + c[e] + '" ' + d + "/>")
    }("display", "inline-block", ["/lib/twitter-bootstrap/css/bootstrap.min.css"],
        "rel=\u0022stylesheet\u0022 ");
  </script>
</head>
...
```

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

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

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

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

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

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

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

2.9.6 Working with Image Elements

The **ImageTagHelper** class is used to provide cache busting for images through the src attribute of img elements, allowing an application to take advantage of caching while ensuring that modifications to images are reflected immediately. The

ImageTagHelper class operates in img elements that define the asp-append-version attribute, which is described in Table 26-7 for quick reference.

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

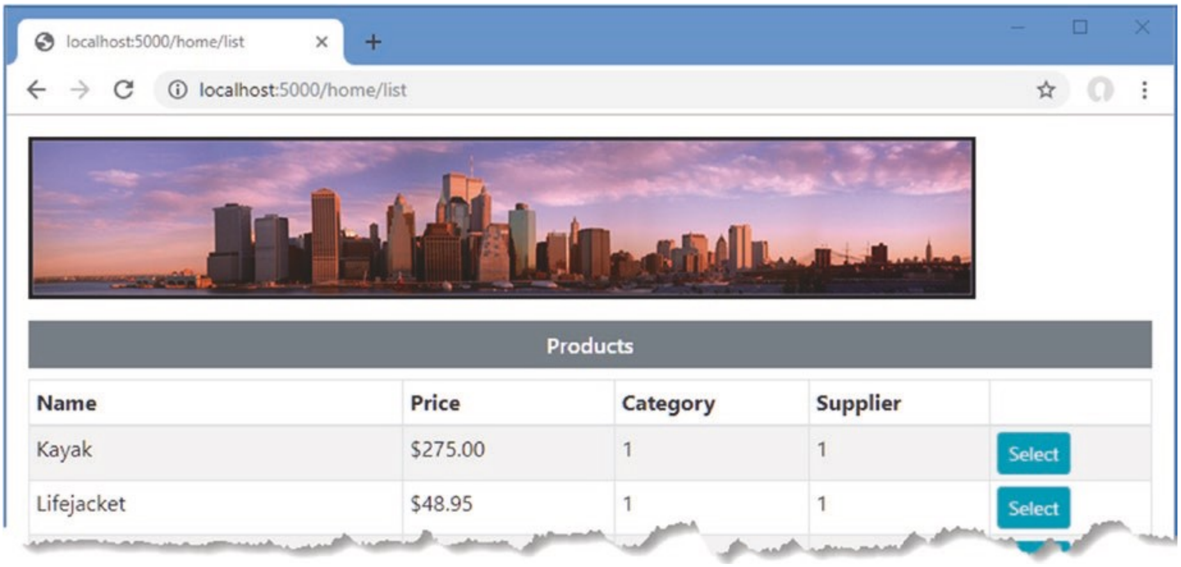
Name	**Description**
asp-append-version	This attribute is used to enable cache busting, as described in the "Understanding Cache Busting" sidebar.

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

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

```
<!DOCTYPE html>
<html>
<head>
   <title>@ViewBag.Title</title>
   <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
   <div class="m-2">
      <img src="/images/city.png" asp-append-version="true" class="m-2" />
      @RenderBody()
   </div>
</body>
</html>
```

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



Name	Price	Category	Supplier	
Kayak	$275.00	1	1	Select
Lifejacket	$48.95	1	1	Select

Figure 26-7. *Using an image*

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

```
...
<img src="/images/city.png?v=KaMNDSZFAJufRcRDpKh0K_IlPNc7E" class="m-2">
...
```

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

2.9.7 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.

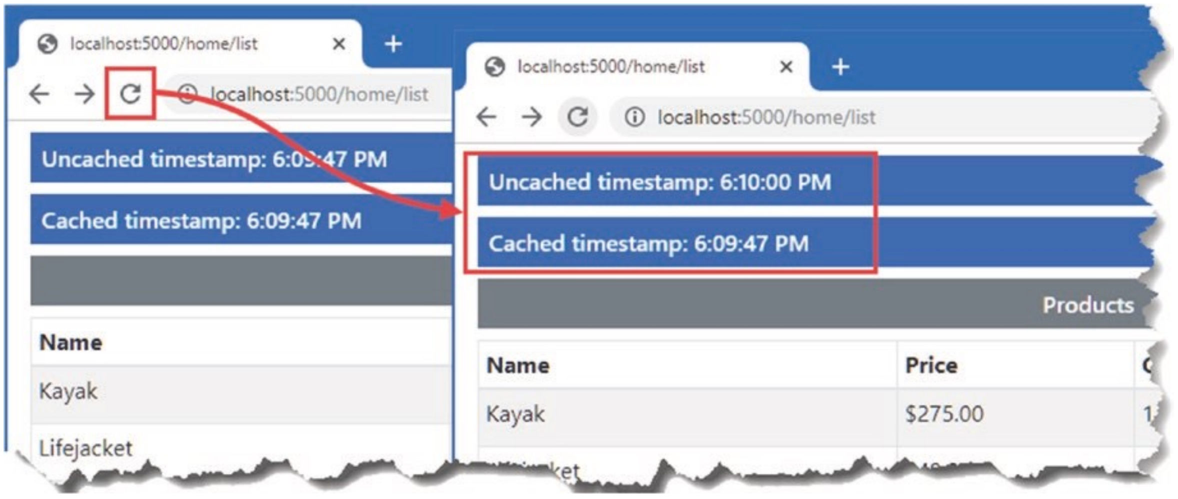


Figure 26-8. *Using the caching tag helper*

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.

2.9.7.1 Setting Cache Expiry

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

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

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

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

2.9.7.1.1 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.

2.9.7.1.2 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.

2.9.7.1.3 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 varyby. 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.

2.9.8 Using the Hosting Environment Tag Helper

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

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

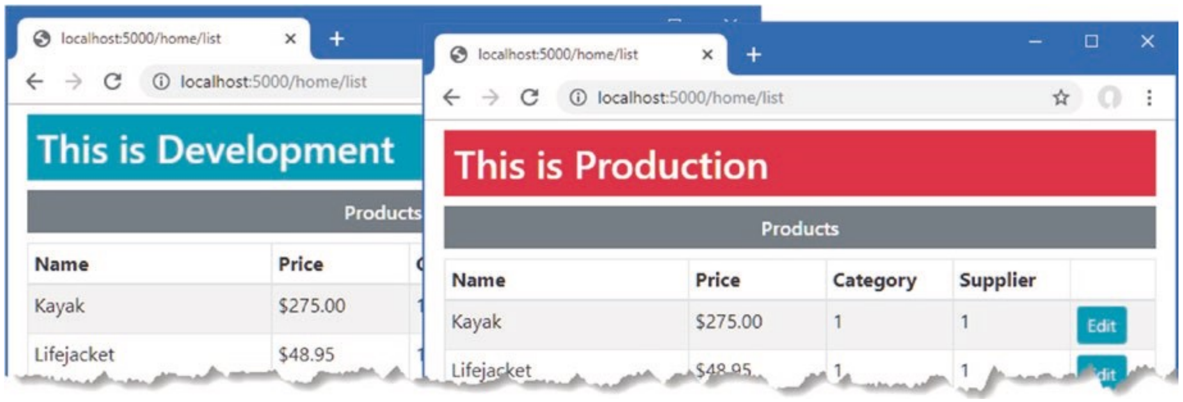
Name	Description
names	This attribute is used to specify a comma-separated list of hosting environment names for which the content contained within the environment element will be included in the HTML sent to the client.

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

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

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

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



This is Development

Products

Name	Price
Kayak	$275.00
Lifejacket	$48.95

This is Production

Products

Name	Price	Category	Supplier	
Kayak	$275.00	1	1	Edit
Lifejacket	$48.95	1	1	Edit

Figure 26-9. *Managing content using the hosting environment*

3 Razor page handler

Razor Page handler methods use the same IActionResult interface to control the responses they generate. To make page model classes easier to develop, handler methods have an implied result that displays the view part of the page. Listing below makes the result explicit.

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

namespace WebApp.Pages.Suppliers
{
    public class ListModel : PageModel
    {
        private DataContext context;
        public IEnumerable<Supplier> Suppliers { get; set; }
        public long? id { get; set; }
        public ListModel(DataContext ctx)
        {
            context = ctx;
        }
        public async Task<IActionResult> OnGetAsync(long? id)
        {
            this.id = id;
            if (id == null)
                Suppliers = context.Suppliers;
            else
            {
                Supplier = await context.Suppliers.FindAsync(id);
                if (Supplier == null) return NotFound();
            }
            return Page();
        }
    }
}
```

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

The PageModel class provides other methods that create different action results to produce different outcomes, as described in table below.

Name	**Description**
Page()	This IActionResult returned by this method produces a 200 OK status code and renders the view part of the Razor Page.
NotFound()	The IActionResult returned by this method produces a 404 NOT FOUND status code.
BadRequest(state)	The IActionResult returned by this method produces a 400 BAD REQUEST status code. The method accepts an optional model state object that describes the problem to the client.
File(name, type)	The IActionResult returned by this method produces a 200 OK response, sets the Content-Type header to the specified type, and sends the specified file to the client.

Redirect(path)
RedirectPermanent(path)
	The IActionResult returned by these methods produces 302 FOUND and 301 MOVED PERMANENTLY responses, which redirect the client to the specified URL.

RedirectToAction(name)
RedirectToActionPermanent(name)
	The IActionResult returned by these methods produces 302 FOUND and 301 MOVED PERMANENTLY responses, which redirect the client to the specified action method.

RedirectToPage(name)
RedirectToPagePermanent(name)
	The IActionResult returned by these methods produce 302 FOUND and 301 MOVED PERMANENTLY responses that redirect the client to another Razor Page. If no name is supplied, the client is redirected to the current page.

StatusCode(code)	The IActionResult returned by this method produces a response with the specific status code.

3.1 Enable default tag handlers

Add file _ViewImports.cshtml in folder Pages with content below:

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

3.2 Enable optional parameters using @page

Update content of List.cshtml in folder Pages/Suppliers:

```
@page "/lists/suppliers/{id:long?}"
@model WebApp.Pages.Suppliers.ListModel
@{
    ViewData["Title"] = "Suppliers";
}
<!DOCTYPE html>
<html>
<head>
    <link href="/lib/bootstrap/dist/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    @if (Model.id == null)
    {
        <h5 class="bg-primary text-white text-center m-2 p-2">Suppliers</h5>
        <ul class="list-group m-2">
            @foreach (WebApp.Models.Supplier s in Model.Suppliers)
            {
                <li class="list-group-item">
                    <a asp-page="/Suppliers/List" asp-route-id="@(s.SupplierId)">@s.Name</a></li>
            }
        </ul>
    }
    else
    {
        <h5 class="bg-primary text-white text-center m-2 p-2">Supplier @Model.Supplier.Name</h5>
        <table class="table table-striped">
            <tr><th>Id</th><td>@Model.Supplier.SupplierId</td></tr>
            <tr><th>Name</th><td>@Model.Supplier.Name</td></tr>
            <tr><th>City</th><td>@Model.Supplier.City</td></tr>
        </table>
        <a asp-page="/Suppliers/List" asp-route-id="@(null)">All Suppliers</a>
    }
</body>
</html>
```

4 Summary

In this lesson we have discussed about built-in tag helpers including global tag helpers, asp-..., etc. We have also discussed about static file including styles and scripts. Finally, we have discussed about Razor page handlers. Next lesson we will discuss about Application Programming Interface (API).