CHAPTER 27

Using the Forms Tag Helpers

In this chapter, I describe the built-in tag helpers that are used to create HTML forms. These tag helpers ensure forms are submitted to the correct action or page handler method and that elements accurately represent specific model properties. Table 27-1 puts the form tag helpers in context.

Table 27-1. *Putting Form Tag Helpers in Context*

Question	Answer
What are they?	These built-in tag helpers transform HTML form elements.
Why are they useful?	These tag helpers ensure that HTML forms reflect the application's routing configuration and data model.
How are they used?	Tag helpers are applied to HTML elements using `asp-*` attributes.
Are there any pitfalls or limitations?	These tag helpers are reliable and predictable and present no serious issues.
Are there any alternatives?	You don't have to use tag helpers and can define forms without them if you prefer.

Table 27-2 summarizes the chapter.

Table 27-2. *Chapter Summary*

Problem	Solution	Listing
Specifying how a form will be submitted	Use the form tag helper attributes	10–13
Transforming `input` elements	Use the input tag helper attributes	14–22
Transforming `label` elements	Use the label tag helper attributes	23
Populating `select` elements	Use the select tag helper attributes	24–26
Transforming text areas	Use the text area tag helper attributes	27
Protecting against cross-site request forgery	Enable the anti-forgery feature	28–32

Preparing for This Chapter

This chapter uses the WebApp project from Chapter 26. To prepare for this chapter, replace the contents of the `_SimpleLayout.cshtml` file in the `Views/Shared` folder with those shown in Listing 27-1.

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

Listing 27-1. The Contents of the _SimpleLayout.cshtml File in the Views/Shared Folder

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

This chapter uses controller views and Razor Pages to present similar content. To differentiate more readily between controllers and pages, add the route shown in Listing 27-2 to the Startup class.

Listing 27-2. Adding a Route in the Startup.cs File in the WebApp Folder

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

namespace WebApp {
    public class Startup {

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

        public IConfiguration Configuration { get; set; }

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

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

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

The new route introduces a static path segment that makes it obvious that a URL targets a controller.

Dropping the Database

Open a new PowerShell command prompt, navigate to the folder that contains the `WebApp.csproj` file, and run the command shown in Listing 27-3 to drop the database.

Listing 27-3. Dropping the Database

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

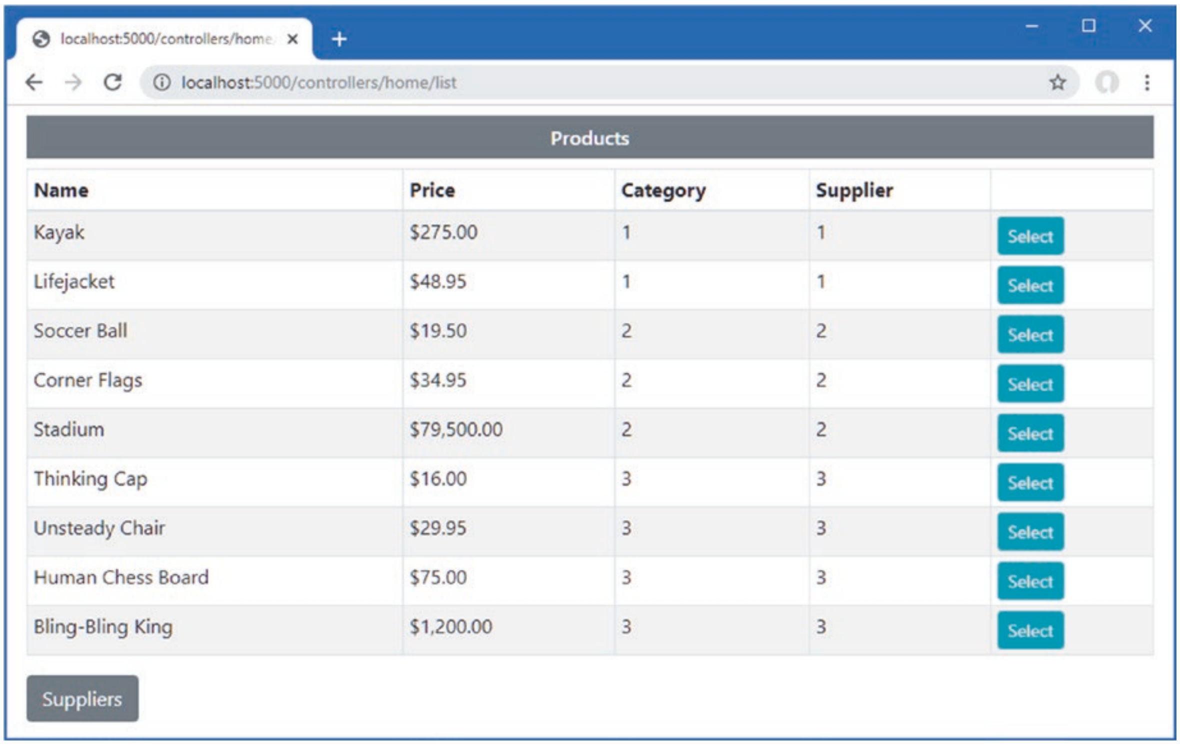
Running the Example Application

Select Start Without Debugging or Run Without Debugging from the Debug menu or use the PowerShell command prompt to run the command shown in Listing 27-4.

Listing 27-4. Running the Example Application

```
dotnet run
```

Use a browser to request `http://localhost:5000/controllers/home/list`, which will display a list of products, as shown in Figure 27-1.


localhost:5000/controllers/home
localhost:5000/controllers/home/list
Products

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 27-1. *Running the example application*

Understanding the Form Handling Pattern

Most HTML forms exist within a well-defined pattern, shown in Figure 27-2. First, the browser sends an HTTP GET request, which results in an HTML response containing a form, making it possible for the user to provide the application with data. The user clicks a button that submits the form data with an HTTP POST request, which allows the application to receive and process the user's data. Once the data has been processed, a response is sent that redirects the browser to a URL that provides confirmation of the user's actions.

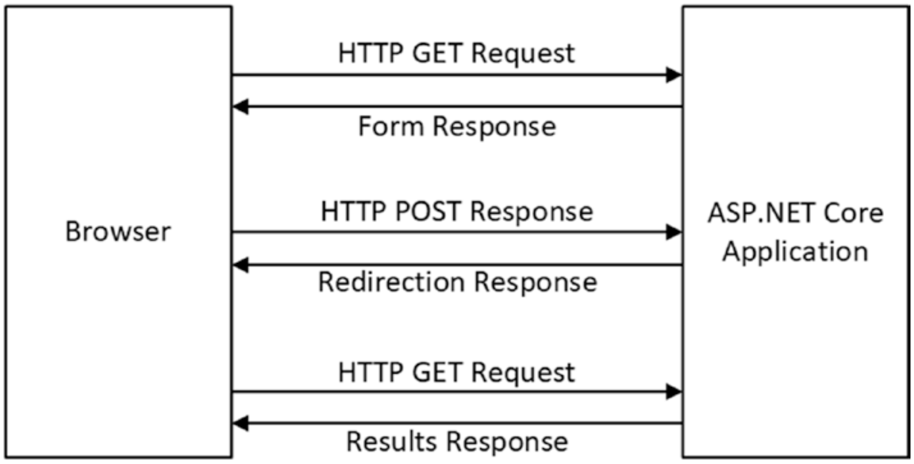


Figure 27-2. *The HTML Post/Redirect/Get pattern*

This is known as the Post/Redirect/Get pattern, and the redirection is important because it means the user can click the browser's reload button without sending another POST request, which can lead to inadvertently repeating an operation.

In the sections that follow, I show how to follow the pattern with controllers and Razor Pages. I start with a basic implementation of the pattern and then demonstrate improvements using tag helpers and, in Chapter 28, the model binding feature.

Creating a Controller to Handle Forms

Controllers that handle forms are created by combining features described in earlier chapters. Add a class file named `FormController.cs` to the `Controllers` folder with the code shown in Listing 27-5.

Listing 27-5. The Contents of the FormController.cs File in the Controllers Folder

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

namespace WebApp.Controllers {

    public class FormController : Controller {
        private DataContext context;

        public FormController(DataContext dbContext) {
            context = dbContext;
        }

        public async Task<IActionResult> Index(long id = 1) {
            return View("Form", await context.Products.FindAsync(id));
        }
        [HttpPost]
        public IActionResult SubmitForm() {
            foreach (string key in Request.Form.Keys
                    .Where(k => !k.StartsWith("_"))) {
                TempData[key] = string.Join(", ", Request.Form[key]);
            }
            return RedirectToAction(nameof(Results));
        }

        public IActionResult Results() {
            return View(TempData);
        }
    }
}
```

The Index action method selects a view named Form, which will render an HTML form to the user. When the user submits the form, it will be received by the SubmitForm action, which has been decorated with the HttpPost attribute so that it can only receive HTTP POST requests. This action method processes the HTML form data available through the HttpRequest.Form property so that it can be stored using the temp data feature. The temp data feature can be used to pass data from one request to another but can be used only to store simple data types. Each form data value is presented as a string array, which I convert to a single comma-separated string for storage. The browser is redirected to the Results action method, which selects the default view and provides the temp data as the view model.

■ **Tip** Only form data values whose name doesn't begin with an underscore are displayed. I explain why in the "Using the Anti-forgery Feature" section, later in this chapter.

To provide the controller with views, create the Views/Form folder and add to it a Razor view file named Form.cshtml with the content shown in Listing 27-6.

Listing 27-6. The Contents of the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<form action="/controllers/form/submitform" method="post">
    <div class="form-group">
        <label>Name</label>
        <input class="form-control" name="Name" value="@Model.Name" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

This view contains a simple HTML form that is configured to submit its data to the SubmitForm action method using a POST request. The form contains an input element whose value is set using a Razor expression. Next, add a Razor view named Results.cshtml to the Views/Forms folder with the content shown in Listing 27-7.

Listing 27-7. The Contents of the Results.cshtml File in the Views/Form Folder

```
@model TempDataDictionary
@{ Layout = "_SimpleLayout"; }

<table class="table table-striped table-bordered table-sm">
    <thead>
        <tr class="bg-primary text-white text-center">
            <th colspan="2">Form Data</th>
        </tr>
    </thead>
    <tbody>
        @foreach (string key in Model.Keys) {
            <tr>
                <th>@key</th>
                <td>@Model[key]</td>
            </tr>
        }
    </tbody>
</table>
<a class="btn btn-primary" asp-action="Index">Return</a>
```

This view displays the form data back to the user. I'll show you how to process form data in more useful ways in Chapter 31, but for this chapter the focus is on creating the forms, and seeing the data contained in the form is enough to get started.

Restart ASP.NET Core and use a browser to request `http://localhost:5000/controllers/form` to see the HTML form. Enter a value into the text field and click Submit to send a POST request, which will be handled by the `SubmitForm` action. The form data will be stored as temp data, and the browser will be redirected, producing the response shown in Figure 27-3.

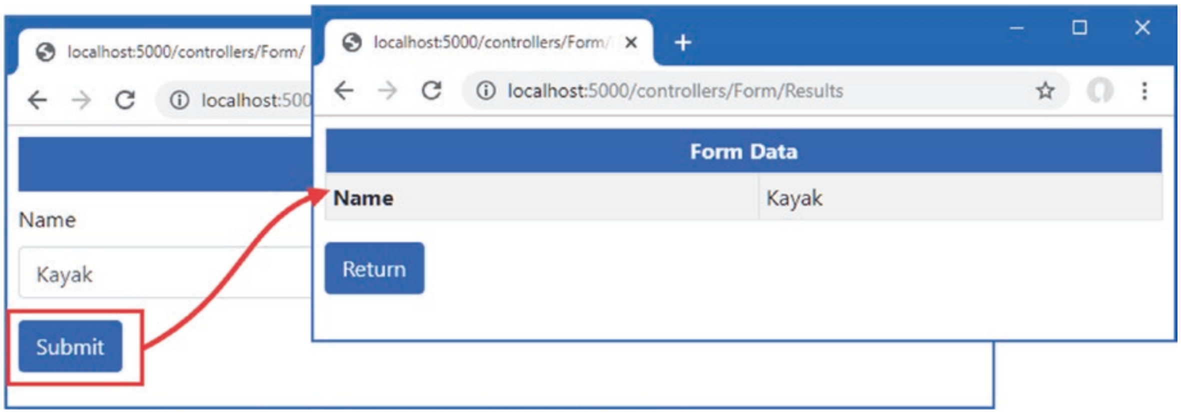

localhost:5000/controllers/Form/
localhost:500
Name
Kayak
Submit
localhost:5000/controllers/Form/
localhost:5000/controllers/Form/Results
Form Data
Name
Kayak
Return

Figure 27-3. *Using a controller to render and process an HTML form*

Creating a Razor Page to Handle Forms

The same pattern can be implemented using Razor Pages. One page is required to render and process the form data, and a second page displays the results. Add a Razor Page named `FormHandler.cshtml` to the Pages folder with the contents shown in Listing 27-8.

Listing 27-8. The Contents of the FormHandler.cshtml File in the Pages Folder

```
@page "/pages/form/{id:long?}"
@model FormHandlerModel
@using Microsoft.AspNetCore.Mvc.RazorPages

<div class="m-2">
    <h5 class="bg-primary text-white text-center p-2">HTML Form</h5>
    <form asp-page="FormHandler" method="post">
        <div class="form-group">
            <label>Name</label>
            <input class="form-control" name="Name" value="@Model.Product.Name" />
        </div>
        <button type="submit" class="btn btn-primary">Submit</button>
    </form>
</div>
```

```
@functions {

    [IgnoreAntiforgeryToken]
    public class FormHandlerModel : PageModel {
        private DataContext context;

        public FormHandlerModel(DataContext dbContext) {
            context = dbContext;
        }

        public Product Product { get; set; }

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

        public IActionResult OnPost() {
            foreach (string key in Request.Form.Keys
                    .Where(k => !k.StartsWith("_"))) {
                TempData[key] = string.Join(", ", Request.Form[key]);
            }
            return RedirectToPage("FormResults");
        }
    }
}
```

The OnGetAsync handler methods retrieves a Product from the database, which is used by the view to set the value for the input element in the HTML form. The form is configured to send an HTTP POST request that will be processed by the OnPost handler method. The form data is stored as temp data, and the browser is sent a redirection to a form named FormResults. To create the page that the browser will be redirected to, add a Razor Page named FormResults.cshtml to the Pages folder with the content shown in Listing 27-9.

■ **Tip** The page model class in Listing 27-8 is decorated with the IgnoreAntiforgeryToken attribute, which is described in the "Using the Anti-forgery Feature" section.

Listing 27-9. The Contents of the FormResults.cshtml File in the Pages Folder

```
@page "/pages/results"

<div class="m-2">
    <table class="table table-striped table-bordered table-sm">
        <thead>
            <tr class="bg-primary text-white text-center">
                <th colspan="2">Form Data</th>
            </tr>
        </thead>
        <tbody>
            @foreach (string key in TempData.Keys) {
                <tr>
                    <th>@key</th>
                    <td>@TempData[key]</td>
                </tr>
            }
        </tbody>
    </table>
    <a class="btn btn-primary" asp-page="FormHandler">Return</a>
</div>
```

No code is required for this page, which accesses temp data directly and displays it in a table. Use a browser to navigate to `http://localhost:5000/pages/form`, enter a value into the text field, and click the Submit button. The form data will be processed by the `OnPost` method defined in Listing 27-9, and the browser will be redirected to `/pages/results`, which displays the form data, as shown in Figure 27-4.

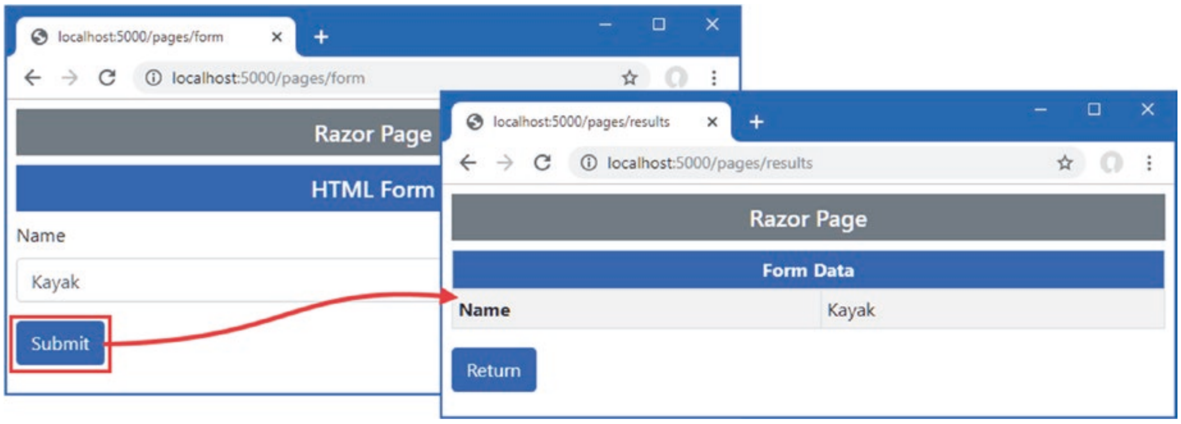

localhost:5000/pages/form
localhost:5000/pages/form
Razor Page
HTML Form
Name
Kayak
Submit
localhost:5000/pages/results
localhost:5000/pages/results
Razor Page
Form Data
Name
Kayak
Return

Figure 27-4. *Using Razor Pages to render and process an HTML form*

Using Tag Helpers to Improve HTML Forms

The examples in the previous section show the basic mechanisms for dealing with HTML forms, but ASP.NET Core includes tag helpers that transform form elements. In the sections that follow, I describe the tag helpers and demonstrate their use.

Working with Form Elements

The `FormTagHelper` class is the built-in tag helper for `form` elements and is used to manage the configuration of HTML forms so that they target the right action or page handler without the need to hard-code URLs. This tag helper supports the attributes described in Table 27-3.

Table 27-3. *The Built-in Tag Helper Attributes for Form Elements*

Name	Description
`asp-controller`	This attribute is used to specify the `controller` value to the routing system for the `action` attribute URL. If omitted, then the controller rendering the view will be used.
`asp-action`	This attribute is used to specify the action method for the `action` value to the routing system for the `action` attribute URL. If omitted, then the action rendering the view will be used.
`asp-page`	This attribute is used to specify the name of a Razor Page.
`asp-page-handler`	This attribute is used to specify the name of the handler method that will be used to process the request. You can see an example of this attribute in the SportsStore application in Chapter 9.
`asp-route-*`	Attributes whose name begins with `asp-route-` are used to specify additional values for the action attribute URL so that the `asp-route-id` attribute is used to provide a value for the `id` segment to the routing system.
`asp-route`	This attribute is used to specify the name of the route that will be used to generate the URL for the `action` attribute.
`asp-antiforgery`	This attribute controls whether anti-forgery information is added to the view, as described in the "Using the Anti-forgery Feature" section.
`asp-fragment`	This attribute specifies a fragment for the generated URL.

Setting the Form Target

The FormTagHelper transforms form elements so they target an action method or Razor Page without the need for hard-coded URLs. The attributes supported by this tag helper work in the same way as for anchor elements, described in Chapter 26, and use attributes to provide values that help generate URLs through the ASP.NET Core routing system. Listing 27-10 modifies the form element in the Form view to apply the tag helper.

■ **Note** If a form element is defined without a method attribute, then the tag helper will add one with the post value, meaning that the form will be submitted using an HTTP POST request. This can lead to surprising results if you omitted the method attribute because you expect the browser to follow the HTML5 specification and send the form using an HTTP GET request. It is a good idea to always specify the method attribute so that it is obvious how the form should be submitted.

Listing 27-10. Using a Tag Helper in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<form asp-action="submitform" method="post">
    <div class="form-group">
        <label>Name</label>
        <input class="form-control" name="Name" value="@Model.Name" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

The asp-action attribute is used to specify the name of the action that will receive the HTTP request. The routing system is used to generate the URLs, just as for the anchor elements described in Chapter 26. The asp-controller attribute has not been used in Listing 27-10, which means the controller that rendered the view will be used in the URL.

The asp-page attribute is used to select a Razor Page as the target for the form, as shown in Listing 27-11.

Listing 27-11. Setting the Form Target in the FormHandler.cshtml File in the Pages Folder

```
...
<div class="m-2">
    <h5 class="bg-primary text-white text-center p-2">HTML Form</h5>
    <form asp-page="FormHandler" method="post">
        <div class="form-group">
            <label>Name</label>
            <input class="form-control" name="Name" value="@Model.Product.Name" />
        </div>
        <button type="submit" class="btn btn-primary">Submit</button>
    </form>
</div>
...
```

Use a browser to navigate to http://localhost:5000/controllers/form and examine the HTML received by the browser; you will see that the tag helper as added the action attribute to the form element like this:

```
...
<form method="post" action="controllers/Form/submitform">
...
```

This is the same URL that I defined statically when I created the view but with the advantage that changes to the routing configuration will be reflected automatically in the form URL. Request `http://localhost:5000/pages/form`, and you will see that the `form` element has been transformed to target the page URL, like this:

```
...
<form method="post" action="/pages/form">
...
```

Transforming Form Buttons

The buttons that send forms can be defined outside of the `form` element. In these situations, the button has a `form` attribute whose value corresponds to the `id` attribute of the form element it relates to and a `formaction` attribute that specifies the target URL for the form.

The tag helper will generate the `formaction` attribute through the `asp-action`, `asp-controller`, or `asp-page` attributes, as shown in Listing 27-12.

Listing 27-12. Transforming a Button in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<form asp-action="submitform" method="post" id="htmlform">
    <div class="form-group">
        <label>Name</label>
        <input class="form-control" name="Name" value="@Model.Name" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>

<button form="htmlform" asp-action="submitform" class="btn btn-primary mt-2">
    Sumit (Outside Form)
</button>
```

The value of the `id` attribute added to the `form` element is used by the `button` as the value of the `form` attribute, which tells the browser which form to submit when the button is clicked. The attributes described in Table 27-3 are used to identify the target for the form, and the tag helper will use the routing system to generate a URL when the view is rendered. Listing 27-13 applies the same technique to the Razor Page.

Listing 27-13. Transforming a Button in the FormHandler.cshtml File in the Pages Folder

```
...
<div class="m-2">
    <h5 class="bg-primary text-white text-center p-2">HTML Form</h5>
    <form asp-page="FormHandler" method="post" id="htmlform">
        <div class="form-group">
            <label>Name</label>
            <input class="form-control" name="Name" value="@Model.Product.Name" />
        </div>
        <button type="submit" class="btn btn-primary">Submit</button>
    </form>
    <button form="htmlform" asp-page="FormHandler" class="btn btn-primary mt-2">
        Sumit (Outside Form)
    </button>
</div>
...
```

Use a browser to request `http://localhost:5000/controllers/form` or `http://localhost:5000/pages/form` and inspect the HTML sent to the browser. You will see the `button` element outside of the form has been transformed like this:

```
...
<button form="htmlform" class="btn btn-primary mt-2"
        formaction="/controllers/Form/submitform">
    Sumit (Outside Form)
</button>
...
```

Clicking the button submits the form, just as for a button that is defined within the form element, as shown in Figure 27-5.

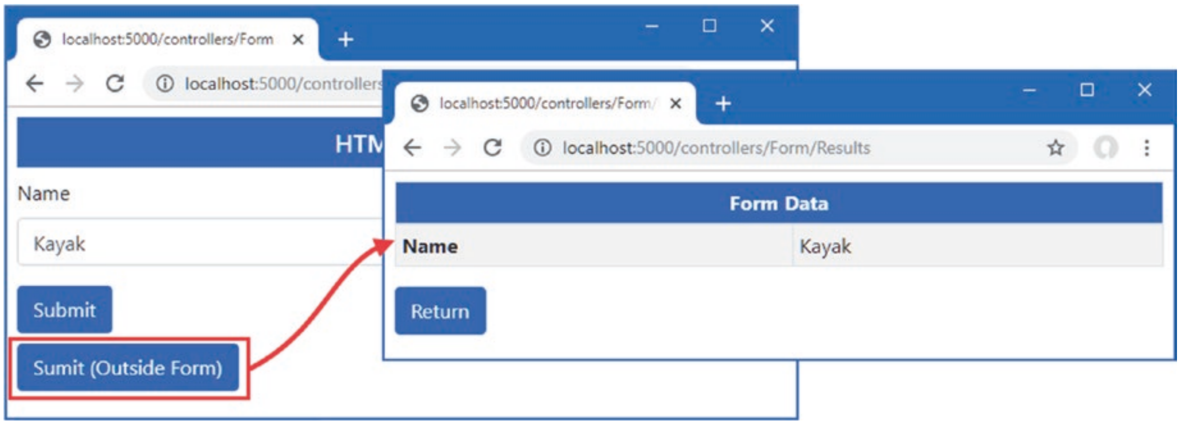

localhost:5000/controllers/Form
localhost:5000/controllers
HTM
Name
Kayak
Submit
Sumit (Outside Form)
localhost:5000/controllers/Form/
localhost:5000/controllers/Form/Results
Form Data
Name
Kayak
Return

Figure 27-5. Defining a button outside of a form element

Working with input Elements

The `input` element is the backbone of HTML forms and provides the main means by which a user can provide an application with unstructured data. The `InputTagHelper` class is used to transform `input` elements so they reflect the data type and format of a view model property they are used to gather, using the attributes described in Table 27-4.

Table 27-4. The Built-in Tag Helper Attributes for input Elements

Name	Description
`asp-for`	This attribute is used to specify the view model property that the `input` element represents.
`asp-format`	This attribute is used to specify a format used for the value of the view model property that the `input` element represents.

The `asp-for` attribute is set to the name of a view model property, which is then used to set the `name`, `id`, `type`, and `value` attributes of the `input` element. Listing 27-14 modifies the `input` element in the controller view to use the `asp-for` attribute.

Listing 27-14. Configuring an input Element in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<form asp-action="submitform" method="post" id="htmlform">
    <div class="form-group">
        <label>Name</label>
        <input class="form-control" asp-for="Name" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>

<button form="htmlform" asp-action="submitform" class="btn btn-primary mt-2">
    Sumit (Outside Form)
</button>
```

This tag helper uses a model expression, described in Listing 27-14, which is why the value for the `asp-for` attribute is specified without the `@` character. If you inspect the HTML the application returns when using a browser to request `http://localhost:5000/controllers/form`, you will see the tag helper has transformed the `input` element like this:

```
...
<div class="form-group">
    <label>Name</label>
    <input class="form-control" type="text" id="Name" name="Name" value="Kayak">
</div>
...
```

The values for the `id` and `name` attributes are obtained through the model expression, ensuring that you don't introduce typos when creating the form. The other attributes are more complex and are described in the sections that follow.

SELECTING MODEL PROPERTIES IN RAZOR PAGES

The `asp-for` attribute for this and the other tag helpers described in this chapter can be used for Razor Pages, but the value for the `name` and `id` attributes in the transformed element includes the name of the page model property. For example, this element selects the `Name` property through the page model's `Product` property:

```
...
<input class="form-control" asp-for="Product.Name" />
...
```

The transformed element will have the following `id` and `name` attributes:

```
...
<input class="form-control" type="text" id="Product_Name" name="Product.Name" >
...
```

This difference is important when using the model binding feature to receive form data, as described in Chapter 28.

Transforming the input Element type Attribute

The `input` element's `type` attribute tells the browser how to display the element and how it should restrict the values the user enters. The `input` element in Listing 27-14 is configured to the `text` type, which is the default `input` element type and offers no restrictions. Listing 27-15 adds another `input` element to the form, which will provide a more useful demonstration of how the `type` attribute is handled.

Listing 27-15. Adding an input Element in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<form asp-action="submitform" method="post" id="htmlform">
    <div class="form-group">
        <label>Id</label>
        <input class="form-control" asp-for="ProductId" />
    </div>
    <div class="form-group">
        <label>Name</label>
        <input class="form-control" asp-for="Name" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>

<button form="htmlform" asp-action="submitform" class="btn btn-primary mt-2">
    Sumit (Outside Form)
</button>
```

The new element uses the `asp-for` attribute to select the view model's `ProductId` property. Use a browser to request `http://localhost:5000/controllers/form` to see how the tag helper has transformed the element.

```
...
<div class="form-group">
    <label>Id</label>
    <input class="form-control" type="number" data-val="true"
        data-val-required="The ProductId field is required."
        id="ProductId" name="ProductId" value="1">
</div>
...
```

The value of the `type` attribute is determined by the type of the view model property specified by the `asp-for` attribute. The type of the `ProductId` property is the C# `long` type, which has led the tag helper to set the `input` element's type attribute to `number`, which restricts the element so it will accept only numeric characters. The `data-val` and `data-val-required` attributes are added to the `input` element to assist with validation, which is described in Chapter 29. Table 27-5 describes how different C# types are used to set the type attribute of `input` elements.

■ **Note** There is latitude in how the `type` attribute is interpreted by browsers. Not all browsers respond to all the `type` values that are defined in the HTML5 specification, and when they do, there are differences in how they are implemented. The `type` attribute can be a useful hint for the kind of data that you are expecting in a form, but you should use the model validation feature to ensure that users provide usable data, as described in Chapter 29.

Table 27-5. *C# Property Types and the Input Type Elements They Generate*

C# Type	input Element type Attribute
`byte`, `sbyte`, `int`, `uint`, `short`, `ushort`, `long`, `ulong`	`number`
`float`, `double`, `decimal`	`text`, with additional attributes for model validation, as described in Chapter 29
`bool`	`checkbox`
`string`	`text`
`DateTime`	`datetime`

The `float`, `double`, and `decimal` types produce `input` elements whose `type` is `text` because not all browsers allow the full range of characters that can be used to express legal values of this type. To provide feedback to the user, the tag helper adds attributes to the `input` element that are used with the validation features described in Chapter 29.

You can override the default mappings shown in Table 27-5 by explicitly defining the `type` attribute on `input` elements. The tag helper won't override the value you define, which allows you to specify a `type` attribute value.

The drawback of this approach is that you must remember to set the `type` attribute in all the views where `input` elements are generated for a given model property. A more elegant—and reliable approach—is to apply one of the attributes described in Table 27-6 to the property in the C# model class.

■ **Tip** The tag helper will set the type attribute of input elements to text if the model property isn't one of the types in Table 27-5 and has not been decorated with an attribute.

Table 27-6. The Input Type Elements Attributes

Attribute	input Element type Attribute
[HiddenInput]	hidden
[Text]	text
[Phone]	tel
[Url]	url
[EmailAddress]	email
[DataType(DataType.Password)]	password
[DataType(DataType.Time)]	time
[DataType(DataType.Date)]	date

Formatting input Element Values

When the action method provides the view with a view model object, the tag helper uses the value of the property given to the asp-for attribute to set the input element's value attribute. The asp-format attribute is used to specify how that data value is formatted. To demonstrate the default formatting, Listing 27-16 adds a new input element to the Form view.

Listing 27-16. Adding an Element in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<form asp-action="submitform" method="post" id="htmlform">
    <div class="form-group">
        <label>Id</label>
        <input class="form-control" asp-for="ProductId" />
    </div>
    <div class="form-group">
        <label>Name</label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label>Price</label>
        <input class="form-control" asp-for="Price" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>

<button form="htmlform" asp-action="submitform" class="btn btn-primary mt-2">
    Sumit (Outside Form)
</button>
```

Use a browser to navigate to http://localhost:5000/controllers/form/index/5 and examine the HTML the browser receives. By default, the value of the input element is set using the value of the model property, like this:

```
...
<input class="form-control" type="text" data-val="true"
    data-val-number="The field Price must be a number."
    data-val-required="The Price field is required."
    id="Price" name="Price" value="79500.00">
...
```

This format, with two decimal places, is how the value is stored in the database. In Chapter 26, I used the Column attribute to select a SQL type to store Price values, like this:

```
...
[Column(TypeName = "decimal(8, 2)")]
public decimal Price { get; set; }
...
```

This type specifies a maximum precision of eight digits, two of which will appear after the decimal place. This allows a maximum value of 999,999.99, which is enough to represent prices for most online stores. The asp-format attribute accepts a format string that will be passed to the standard C# string formatting system, as shown in Listing 27-17.

Listing 27-17. Formatting a Data Value in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<form asp-action="submitform" method="post" id="htmlform">
    <div class="form-group">
        <label>Id</label>
        <input class="form-control" asp-for="ProductId" />
    </div>
    <div class="form-group">
        <label>Name</label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label>Price</label>
        <input class="form-control" asp-for="Price" asp-format="{0:#,###.00}" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>

<button form="htmlform" asp-action="submitform" class="btn btn-primary mt-2">
    Sumit (Outside Form)
</button>
```

The attribute value is used verbatim, which means you must include the curly brace characters and the 0: reference, as well as the format you require. Refresh the browser, and you will see that the value for the input element has been formatted, like this:

```
...
<input class="form-control" type="text" data-val="true"
    data-val-number="The field Price must be a number."
    data-val-required="The Price field is required."
    id="Price" name="Price" value="79,500.00">
...
```

This feature should be used with caution because you must ensure that the rest of the application is configured to support the format you use and that the format you create contains only legal characters for the input element type.

Applying Formatting via the Model Class

If you always want to use the same formatting for a model property, then you can decorate the C# class with the `DisplayFormat` attribute, which is defined in the `System.ComponentModel.DataAnnotations` namespace. The `DisplayFormat` attribute requires two arguments to format a data value: the `DataFormatString` argument specifies the formatting string, and setting the `ApplyFormatInEditMode` to `true` specifies that formatting should be used when values are being applied to elements used for editing, including the `input` element. Listing 27-18 applies the attribute to the `Price` property of the `Product` class, specifying a different formatting string from earlier examples.

Listing 27-18. Applying a Formatting Attribute to the Product.cs File in the Models Folder

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

namespace WebApp.Models {
    public class Product {

        public long ProductId { get; set; }

        public string Name { get; set; }

        [Column(TypeName = "decimal(8, 2)")]
        [DisplayFormat(DataFormatString = "{0:c2}", ApplyFormatInEditMode = true)]
        public decimal Price { get; set; }

        public long CategoryId { get; set; }
        public Category Category { get; set; }

        public long SupplierId { get; set; }
        public Supplier Supplier { get; set; }
    }
}
```

The `asp-format` attribute takes precedence over the `DisplayFormat` attribute, so I have removed the attribute from the view, as shown in Listing 27-19.

Listing 27-19. Removing an Attribute in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<form asp-action="submitform" method="post" id="htmlform">
    <div class="form-group">
        <label>Id</label>
        <input class="form-control" asp-for="ProductId" />
    </div>
    <div class="form-group">
        <label>Name</label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label>Price</label>
        <input class="form-control" asp-for="Price" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>

<button form="htmlform" asp-action="submitform" class="btn btn-primary mt-2">
    Sumit (Outside Form)
</button>
```

Restart ASP.NET Core and use a browser to request `http://localhost:5000/controllers/form/index/5`, and you will see that the formatting string defined by the attribute has been applied, as shown in Figure 27-6.

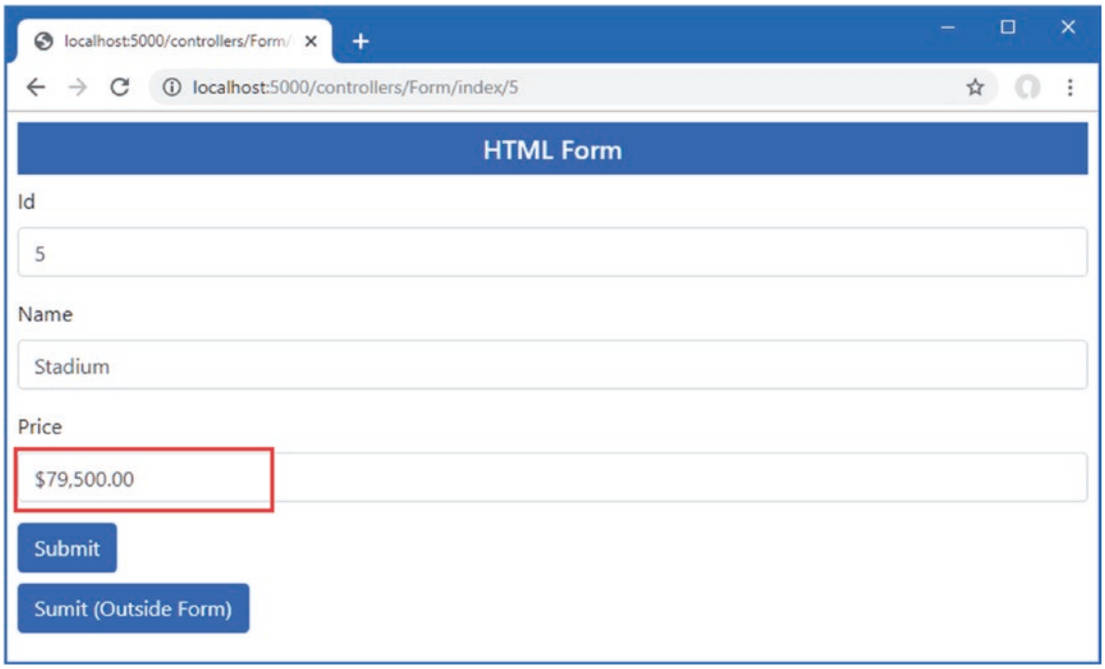

localhost:5000/controllers/Form/
localhost:5000/controllers/Form/index/5
HTML Form
Id
5
Name
Stadium
Price
$79,500.00
Submit
Sumit (Outside Form)

Figure 27-6. *Formatting data values*

I chose this format to demonstrate the way the formatting attribute works, but, as noted previously, care must be taken to ensure that the application is able to process the formatted values using the model binding and validation features described in Chapters 28 and 29.

Displaying Values from Related Data in input Elements

When using Entity Framework Core, you will often need to display data values that are obtained from related data, which is easily done using the `asp-for` attribute because a model expression allows the nested navigation properties to be selected. First, Listing 27-20 includes related data in the view model object provided to the view.

Listing 27-20. Including Related Data in the FormController.cs File in the Controllers Folder

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

namespace WebApp.Controllers {

    public class FormController : Controller {
        private DataContext context;

        public FormController(DataContext dbContext) {
            context = dbContext;
        }

        public async Task<IActionResult> Index(long id = 1) {
            return View("Form", await context.Products.Include(p => p.Category)
                .Include(p => p.Supplier).FirstAsync(p => p.ProductId == id));
        }
        [HttpPost]
        public IActionResult SubmitForm() {
            foreach (string key in Request.Form.Keys
                    .Where(k => !k.StartsWith("_"))) {
                TempData[key] = string.Join(", ", Request.Form[key]);
            }
            return RedirectToAction(nameof(Results));
        }

        public IActionResult Results() {
            return View(TempData);
        }
    }
}
```

Notice that I don't need to worry about dealing with circular references in the related data because the view model object isn't serialized. The circular reference issue is important only for web service controllers. In Listing 27-21, I have updated the `Form` view to include `input` elements that use the `asp-for` attribute to select related data.

Listing 27-21. Displaying Related Data in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<form asp-action="submitform" method="post" id="htmlform">
    <div class="form-group">
        <label>Id</label>
        <input class="form-control" asp-for="ProductId" />
    </div>
    <div class="form-group">
        <label>Name</label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label>Price</label>
        <input class="form-control" asp-for="Price" />
    </div>
    <div class="form-group">
        <label>Category</label>
        <input class="form-control" asp-for="Category.Name" />
    </div>
    <div class="form-group">
        <label>Supplier</label>
        <input class="form-control" asp-for="Supplier.Name" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>

<button form="htmlform" asp-action="submitform" class="btn btn-primary mt-2">
    Sumit (Outside Form)
</button>
```

The value of the asp-for attribute is expressed relative to the view model object and can include nested properties, allowing me to select the Name properties of the related objects that Entity Framework Core has assigned to the Category and Supplier navigation properties. The same technique is used in Razor Pages, except that the properties are expressed relative to the page model object, as shown in Listing 27-22.

Listing 27-22. Displayed Related Data in the FormHandler.cshtml File in the Pages Folder

```
@page "/pages/form/{id:long?}"
@model FormHandlerModel
@using Microsoft.AspNetCore.Mvc.RazorPages
@using Microsoft.EntityFrameworkCore

<div class="m-2">
    <h5 class="bg-primary text-white text-center p-2">HTML Form</h5>
    <form asp-page="FormHandler" method="post" id="htmlform">
        <div class="form-group">
            <label>Name</label>
            <input class="form-control" asp-for="Product.Name" />
        </div>
        <div class="form-group">
            <label>Price</label>
            <input class="form-control" asp-for="Product.Price" />
        </div>
        <div class="form-group">
            <label>Category</label>
            <input class="form-control" asp-for="Product.Category.Name" />
        </div>
        <div class="form-group">
            <label>Supplier</label>
            <input class="form-control" asp-for="Product.Supplier.Name" />
        </div>
        <button type="submit" class="btn btn-primary">Submit</button>
    </form>
    <button form="htmlform" asp-page="FormHandler" class="btn btn-primary mt-2">
        Sumit (Outside Form)
    </button>
</div>

@functions {

    [IgnoreAntiforgeryToken]
    public class FormHandlerModel : PageModel {
        private DataContext context;

        public FormHandlerModel(DataContext dbContext) {
            context = dbContext;
        }

        public Product Product { get; set; }

        public async Task OnGetAsync(long id = 1) {
            Product = await context.Products.Include(p => p.Category)
                .Include(p => p.Supplier).FirstAsync(p => p.ProductId == id);
        }
        public IActionResult OnPost() {
            foreach (string key in Request.Form.Keys
                    .Where(k => !k.StartsWith("_"))) {
                TempData[key] = string.Join(", ", Request.Form[key]);
            }
            return RedirectToPage("FormResults");
        }
    }
}
```

To see the effect, restart ASP.NET Core so the changes to the controller take effect, and use a browser to request `http://localhost:5000/controller/form`, which produces the response shown on the left of Figure 27-7. Use the browser to request `http://localhost:5000/pages/form`, and you will see the same features used by the Razor Page, as shown on the right of Figure 27-7.

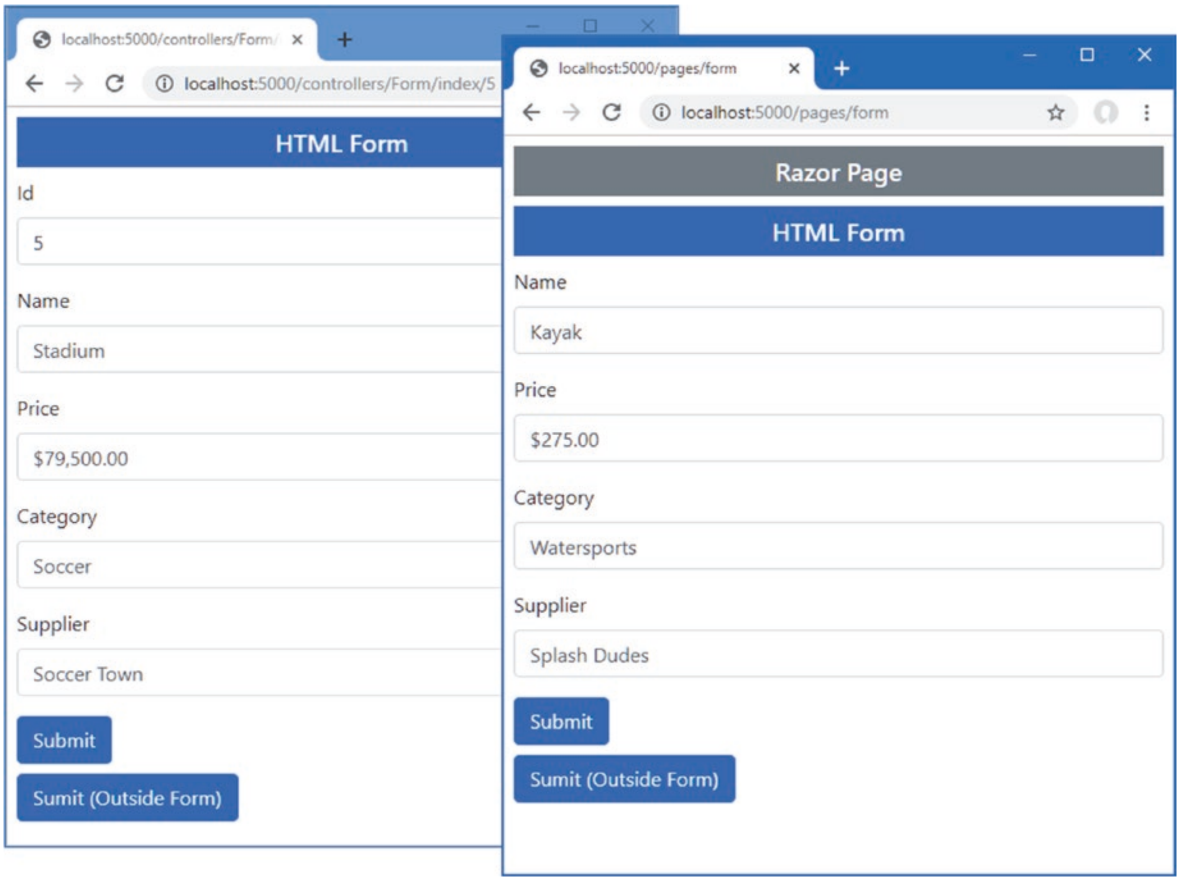


Figure 27-7. *Displaying related data*

Working with label Elements

The `LabelTagHelper` class is used to transform `label` elements so the `for` attribute is set consistently with the approach used to transform `input` elements. Table 27-7 describes the attribute supported by this tag helper.

Table 27-7. *The Built-in Tag Helper Attribute for label Elements*

Name	Description
`asp-for`	This attribute is used to specify the view model property that the `label` element describes.

The tag helper sets the content of the `label` element so that it contains the name of the selected view model property. The tag helper also sets the `for` attribute, which denotes an association with a specific `input` element. This aids users who rely on screen readers and allows an `input` element to gain the focus when its associated `label` is clicked.

Listing 27-23 applies the `asp-for` attribute to the `Form` view to associate each `label` element with the `input` element that represents the same view model property.

Listing 27-23. Transforming label Elements in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<form asp-action="submitform" method="post" id="htmlform">
    <div class="form-group">
        <label asp-for="ProductId"></label>
        <input class="form-control" asp-for="ProductId" />
    </div>
    <div class="form-group">
        <label asp-for="Name"></label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label asp-for="Price"></label>
        <input class="form-control" asp-for="Price" />
    </div>
    <div class="form-group">
        <label asp-for="Category.Name">Category</label>
        <input class="form-control" asp-for="Category.Name" />
    </div>
    <div class="form-group">
        <label asp-for="Supplier.Name">Supplier</label>
        <input class="form-control" asp-for="Supplier.Name" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>

<button form="htmlform" asp-action="submitform" class="btn btn-primary mt-2">
    Sumit (Outside Form)
</button>
```

You can override the content for a `label` element by defining it yourself, which is what I have done for the related data properties in Listing 27-23. The tag helper would have set the content for both these `label` elements to be `Name`, which is not a useful description. Defining the element content means the `for` attribute will be applied, but a more useful name will be displayed to the user. Use a browser to request `http://localhost:5000/controllers/form` to see the names used for each element, as shown in Figure 27-8.

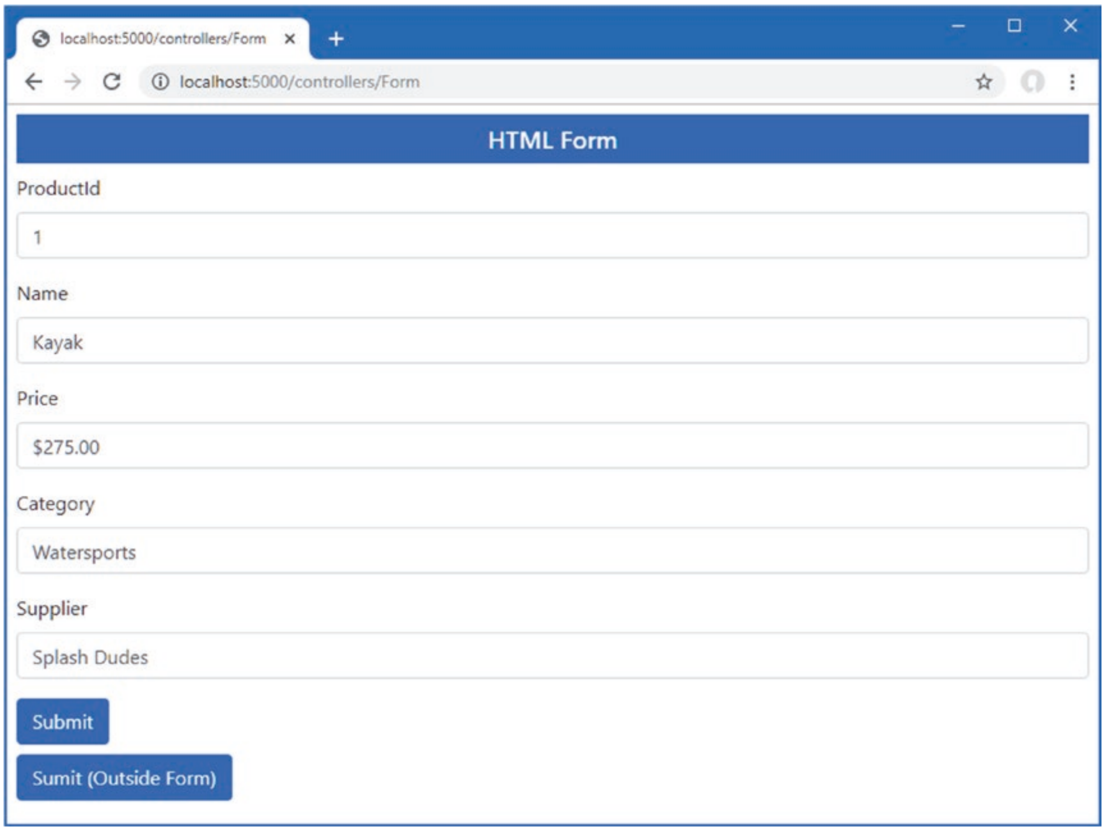


Figure 27-8. *Transforming label elements*

Working with Select and Option Elements

The `select` and `option` elements are used to provide the user with a fixed set of choices, rather than the open data entry that is possible with an `input` element. The `SelectTagHelper` is responsible for transforming `select` elements and supports the attributes described in Table 27-8.

Table 27-8. *The Built-in Tag Helper Attributes for select Elements*

Name	Description
`asp-for`	This attribute is used to specify the view or page model property that the `select` element represents.
`asp-items`	This attribute is used to specify a source of values for the option elements contained within the `select` element.

The `asp-for` attribute sets the value of the `for` and `id` attributes to reflect the model property that it receives. In Listing 27-24, I have replaced the `input` element for the category with a `select` element that presents the user with a fixed range of values.

Listing 27-24. Using a select Element in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<form asp-action="submitform" method="post" id="htmlform">
    <div class="form-group">
        <label asp-for="ProductId"></label>
        <input class="form-control" asp-for="ProductId" />
    </div>
    <div class="form-group">
        <label asp-for="Name"></label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label asp-for="Price"></label>
        <input class="form-control" asp-for="Price" />
    </div>
    <div class="form-group">
        <label asp-for="Category.Name">Category</label>
        <select class="form-control" asp-for="CategoryId">
            <option value="1">Watersports</option>
            <option value="2">Soccer</option>
            <option value="3">Chess</option>
        </select>
    </div>
    <div class="form-group">
        <label asp-for="Supplier.Name">Supplier</label>
        <input class="form-control" asp-for="Supplier.Name" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>

<button form="htmlform" asp-action="submitform" class="btn btn-primary mt-2">
    Sumit (Outside Form)
</button>
```

I have manually populated the select element with option elements that provide a range of categories for the user to choose from. If you use a browser to request http://localhost:5000/controllers/form/index/5 and examine the HTML response, you will see that the tag helper has transformed the select element like this:

```
...
<div class="form-group">
    <label for="Category_Name">Category</label>
    <select class="form-control" data-val="true"
          data-val-required="The CategoryId field is required."
          id="CategoryId" name="CategoryId">
        <option value="1">Watersports</option>
        <option value="2" selected="selected">Soccer</option>
        <option value="3">Chess</option>
    </select>
</div>
...
```

Notice that selected attribute has been added to the option element that corresponds to the view model's CategoryId value, like this:

```
...
<option value="2" selected="selected">Soccer</option>
...
```

The task of selecting an option element is performed by the OptionTagHelper class, which receives instructions from the SelectTagHelper through the TagHelperContext.Items collection, described in Chapter 25. The result is that the select element displays the name of the category associated with the Product object's CategoryId value.

Populating a select Element

Explicitly defining the option elements for a select element is a useful approach for choices that always have the same possible values but doesn't help when you need to provide options that are taken from the data model or where you need the same set of options in multiple views and don't want to manually maintain duplicated content.

The asp-items attribute is used to provide the tag helper with a list sequence of SelectListItem objects for which option elements will be generated. Listing 27-25 modifies the Index action of the Form controller to provide the view with a sequence of SelectListItem objects through the view bag.

Listing 27-25. Providing a Data Sequence in the FormController.cs File in the Controllers Folder

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

namespace WebApp.Controllers {

    public class FormController : Controller {
        private DataContext context;

        public FormController(DataContext dbContext) {
            context = dbContext;
        }

        public async Task<IActionResult> Index(long id = 1) {
            ViewBag.Categories
                = new SelectList(context.Categories, "CategoryId", "Name");
            return View("Form", await context.Products.Include(p => p.Category)
                .Include(p => p.Supplier).FirstAsync(p => p.ProductId == id));
        }

        [HttpPost]
        public IActionResult SubmitForm() {
            foreach (string key in Request.Form.Keys
                    .Where(k => !k.StartsWith("_"))) {
                TempData[key] = string.Join(", ", Request.Form[key]);
            }
            return RedirectToAction(nameof(Results));
        }

        public IActionResult Results() {
            return View(TempData);
        }
    }
}
```

SelectListItem objects can be created directly, but ASP.NET Core provides the SelectList class to adapt existing data sequences. In this case, I pass the sequence of Category objects obtained from the database to the SelectList constructor, along with the names of the properties that should be used as the values and labels for option elements. In Listing 27-26, I have updated the Form view to use the SelectList.

Listing 27-26. Using a SelectList in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<form asp-action="submitform" method="post" id="htmlform">
    <div class="form-group">
        <label asp-for="ProductId"></label>
        <input class="form-control" asp-for="ProductId" />
    </div>
    <div class="form-group">
        <label asp-for="Name"></label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label asp-for="Price"></label>
        <input class="form-control" asp-for="Price" />
    </div>
    <div class="form-group">
        <label asp-for="Category.Name">Category</label>
        <select class="form-control" asp-for="CategoryId"
            asp-items="@ViewBag.Categories">
        </select>
    </div>
    <div class="form-group">
        <label asp-for="Supplier.Name">Supplier</label>
        <input class="form-control" asp-for="Supplier.Name" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>

<button form="htmlform" asp-action="submitform" class="btn btn-primary mt-2">
    Sumit (Outside Form)
</button>
```

Restart ASP.NET Core so the changes to the controller take effect and use a browser to request `http://localhost:5000/controllers/form/index/5`. There is no visual change to the content presented to the user, but the `option` elements used to populate the `select` element have been generated from the database, like this:

```
...
<div class="form-group">
    <label for="Category_Name">Category</label>
    <select class="form-control" data-val="true"
           data-val-required="The CategoryId field is required."
           id="CategoryId" name="CategoryId">
        <option value="1">Watersports</option>
        <option selected="selected" value="2">Soccer</option>
        <option value="3">Chess</option>
    </select>
</div>
...
```

This approach means that the options presented to the user will automatically reflect new categories added to the database.

Working with Text Areas

The `textarea` element is used to solicit a larger amount of text from the user and is typically used for unstructured data, such as notes or observations. The `TextAreaTagHelper` is responsible for transforming `textarea` elements and supports the single attribute described in Table 27-9.

Table 27-9. *The Built-in Tag Helper Attributes for TextArea Elements*

Name	Description
`asp-for`	This attribute is used to specify the view model property that the `textarea` element represents.

The `TextAreaTagHelper` is relatively simple, and the value provided for the `asp-for` attribute is used to set the `id` and `name` attributes on the `textarea` element. The value of the property selected by the `asp-for` attribute is used as the content for the `textarea` element. Listing 27-27 replaces the `input` element for the `Supplier.Name` property with a text area to which the `asp-for` attribute has been applied.

Listing 27-27. Using a Text Area in the Form.cshtml File in the Views/Form Folder

```
@model Product
@{  Layout = "_SimpleLayout"; }

<h5 class="bg-primary text-white text-center p-2">HTML Form</h5>

<form asp-action="submitform" method="post" id="htmlform">
    <div class="form-group">
        <label asp-for="ProductId"></label>
        <input class="form-control" asp-for="ProductId" />
    </div>
    <div class="form-group">
        <label asp-for="Name"></label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label asp-for="Price"></label>
        <input class="form-control" asp-for="Price" />
    </div>
    <div class="form-group">
        <label asp-for="Category.Name">Category</label>
        <select class="form-control" asp-for="CategoryId"
             asp-items="@ViewBag.Categories">
        </select>
    </div>
    <div class="form-group">
        <label asp-for="Supplier.Name">Supplier</label>
        <textarea class="form-control" asp-for="Supplier.Name"></textarea>
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>

<button form="htmlform" asp-action="submitform" class="btn btn-primary mt-2">
    Sumit (Outside Form)
</button>
```

Use a browser to request http://`localhost:5000/controllers/form` and examine the HTML received by the browser to see the transformation of the `textarea` element.

```
...
<div class="form-group">
    <label for="Supplier_Name">Supplier</label>
    <textarea class="form-control" id="Supplier_Name" name="Supplier.Name">
        Soccer Town
    </textarea>
</div>
...
```

The TextAreaTagHelper is relatively simple, but it provides consistency with the rest of the form element tag helpers that I have described in this chapter.

Using the Anti-forgery Feature

When I defined the controller action method and page handler methods that process form data, I filtered out form data whose name begins with an underscore, like this:

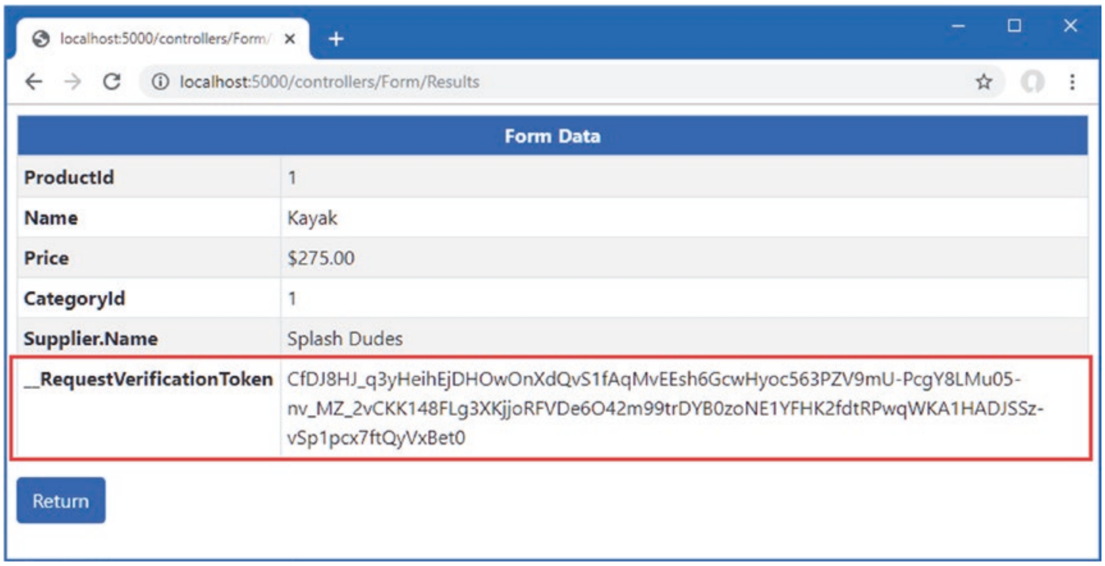
```
...
[HttpPost]
public IActionResult SubmitForm() {
    foreach (string key in Request.Form.Keys
            .Where(k => !k.StartsWith("_"))) {
        TempData[key] = string.Join(", ", Request.Form[key]);
    }
    return RedirectToAction(nameof(Results));
}
...
```

I applied this filter to hide a feature to focus on the values provided by the HTML elements in the form. Listing 27-28 removes the filter from the action method so that all the data received from the HTML form is stored in temp data.

Listing 27-28. Removing a Filter in the FormController.cs File in the Controllers Folder

```
...
[HttpPost]
public IActionResult SubmitForm() {
    foreach (string key in Request.Form.Keys) {
        TempData[key] = string.Join(", ", Request.Form[key]);
    }
    return RedirectToAction(nameof(Results));
}
...
```

Restart ASP.NET Core and use a browser to request `http://localhost:5000/controllers`. Click the Submit button to send the form to the application, and you will see a new item in the results, as shown in Figure 27-9.


localhost:5000/controllers/Form/
localhost:5000/controllers/Form/Results
Return

Form Data	
ProductId	1
Name	Kayak
Price	$275.00
CategoryId	1
Supplier.Name	Splash Dudes
__RequestVerificationToken	CfDJ8HJ_q3yHeihEjDHOwOnXdQvS1fAqMvEEsh6GcwHyoc563PZV9mU-PcgY8LMu05-nv_MZ_2vCKK148FLg3XKjjoRFVDe6O42m99trDYB0zoNE1YFHK2fdtRPwqWKA1HADJSSz-vSp1pcx7ftQyVxBet0

Figure 27-9. *Showing all form data*

The _RequestVerificationToken form value displayed in the results is a security feature that is applied by the FormTagHelper to guard against cross-site request forgery. Cross-site request forgery (CSRF) exploits web applications by taking advantage of the way that user requests are typically authenticated. Most web applications—including those created using ASP.NET Core—use cookies to identify which requests are related to a specific session, with which a user identity is usually associated.

CSRF—also known as *XSRF*—relies on the user visiting a malicious website after using your web application and without explicitly ending their session. The application still regards the user's session as being active, and the cookie that the browser has stored has not yet expired. The malicious site contains JavaScript code that sends a form request to your application to perform an operation without the user's consent—the exact nature of the operation will depend on the application being attacked. Since the JavaScript code is executed by the user's browser, the request to the application includes the session cookie, and the application performs the operation without the user's knowledge or consent.

■ **Tip** CSRF is described in detail at http://en.wikipedia.org/wiki/Cross-site_request_forgery.

If a form element doesn't contain an action attribute—because it is being generated from the routing system with the asp-controller, asp-action, and asp-page attributes—then the FormTagHelper class automatically enables an anti-CSRF feature, whereby a security token is added to the response as a cookie. A hidden input element containing the same security token is added to the HTML form, and it is this token that is shown in Figure 27-9.

Enabling the Anti-forgery Feature in a Controller

By default, controllers accept POST requests even when they don't contain the required security tokens. To enable the anti-forgery feature, an attribute is applied to the controller class, as shown in Listing 27-29.

Listing 27-29. Enabling the Anti-forgery Feature in the FormController.cs File in the Controllers Folder

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

```
namespace WebApp.Controllers {

    [AutoValidateAntiforgeryToken]
    public class FormController : Controller {
        private DataContext context;

        public FormController(DataContext dbContext) {
            context = dbContext;
        }

        public async Task<IActionResult> Index(long id = 1) {
            ViewBag.Categories
                = new SelectList(context.Categories, "CategoryId", "Name");
            return View("Form", await context.Products.Include(p => p.Category)
                .Include(p => p.Supplier).FirstAsync(p => p.ProductId == id));
        }

        [HttpPost]
        public IActionResult SubmitForm() {
            foreach (string key in Request.Form.Keys) {
                TempData[key] = string.Join(", ", Request.Form[key]);
            }
            return RedirectToAction(nameof(Results));
        }

        public IActionResult Results() {
            return View(TempData);
        }
    }
}
```

Not all requests require an anti-forgery token, and the AutoValidateAntiforgeryToken ensures that checks are performed for all HTTP methods except GET, HEAD, OPTIONS, and TRACE.

■ **Tip** Two other attributes can be used to control token validation. The IgnoreValidationToken attribute suppresses validation for an action method or controller. The ValidateAntiForgeryToken attribute does the opposite and enforces validation, even for requests that would not normally require validation, such as HTTP GET requests. I recommend using the AutoValidateAntiforgeryToken attribute, as shown in the listing.

Testing the anti-CSRF feature is a little tricky. I do it by requesting the URL that contains the form (http://localhost:5000/controllers/forms for this example) and then using the browser's F12 developer tools to locate and remove the hidden input element from the form (or change the element's value). When I populate and submit the form, it is missing one part of the required data, and the request will fail.

Enabling the Anti-forgery Feature in a Razor Page

The anti-forgery feature is enabled by default in Razor Pages, which is why I applied the IgnoreAntiforgeryToken attribute to the page handler method in Listing 27-29 when I created the FormHandler page. Listing 27-30 removes the attribute to enable the validation feature.

Listing 27-30. Enabling Request Validation in the FormHandler.cshtml File in the Pages Folder

```
@page "/pages/form/{id:long?}"
@model FormHandlerModel
@using Microsoft.AspNetCore.Mvc.RazorPages
@using Microsoft.EntityFrameworkCore

<div class="m-2">
    <h5 class="bg-primary text-white text-center p-2">HTML Form</h5>
    <form asp-page="FormHandler" method="post" id="htmlform">
        <div class="form-group">
            <label>Name</label>
            <input class="form-control" asp-for="Product.Name" />
        </div>

        <div class="form-group">
            <label>Price</label>
            <input class="form-control" asp-for="Product.Price" />
        </div>
        <div class="form-group">
            <label>Category</label>
            <input class="form-control" asp-for="Product.Category.Name" />
        </div>
        <div class="form-group">
            <label>Supplier</label>
            <input class="form-control" asp-for="Product.Supplier.Name" />
        </div>
        <button type="submit" class="btn btn-primary">Submit</button>
    </form>
    <button form="htmlform" asp-page="FormHandler" class="btn btn-primary mt-2">
        Sumit (Outside Form)
    </button>
</div>

@functions {

    //[IgnoreAntiforgeryToken]
    public class FormHandlerModel : PageModel {
        private DataContext context;

        public FormHandlerModel(DataContext dbContext) {
            context = dbContext;
        }

        public Product Product { get; set; }

        public async Task OnGetAsync(long id = 1) {
            Product = await context.Products.Include(p => p.Category)
                .Include(p => p.Supplier).FirstAsync(p => p.ProductId == id);
        }

        public IActionResult OnPost() {
            foreach (string key in Request.Form.Keys
                    .Where(k => !k.StartsWith("_"))) {
                TempData[key] = string.Join(", ", Request.Form[key]);
            }
            return RedirectToPage("FormResults");
        }
    }
}
```

Testing the validation feature is done in the same way as for controllers and requires altering the HTML document using the browser's developer tools before submitting the form to the application.

Using Anti-forgery Tokens with JavaScript Clients

By default, the anti-forgery feature relies on the ASP.NET Core application being able to include an element in an HTML form that the browser sends back when the form is submitted. This doesn't work for JavaScript clients because the ASP.NET Core application provides data and not HTML, so there is no way to insert the hidden element and receive it in a future request.

For web services, the anti-forgery token can be sent as a JavaScript-readable cookie, which the JavaScript client code reads and includes as a header in its POST requests. Some JavaScript frameworks, such an Angular, will automatically detect the cookie and include a header in requests. For other frameworks and custom JavaScript code, additional work is required.

Listing 27-31 shows the changes required to the ASP.NET Core application to configure the anti-forgery feature for use with JavaScript clients.

Listing 27-31. Configuring the Anti-forgery Token in the Startup.cs File in the WebApp Folder

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

namespace WebApp {
    public class Startup {

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

        public IConfiguration Configuration { get; set; }

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

            services.Configure<AntiforgeryOptions>(opts => {
                opts.HeaderName = "X-XSRF-TOKEN";
            });
        }

        public void Configure(IApplicationBuilder app, DataContext context,
                IAntiforgery antiforgery) {
```

```
            app.UseRequestLocalization();

            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseRouting();

            app.Use(async (context, next) => {
                if (!context.Request.Path.StartsWithSegments("/api")) {
                    context.Response.Cookies.Append("XSRF-TOKEN",
                       antiforgery.GetAndStoreTokens(context).RequestToken,
                       new CookieOptions { HttpOnly = false });
                }
                await next();
            });

            app.UseEndpoints(endpoints => {
                endpoints.MapControllers();
                endpoints.MapControllerRoute("forms",
                    "controllers/{controller=Home}/{action=Index}/{id?}");
                endpoints.MapDefaultControllerRoute();
                endpoints.MapRazorPages();
            });
            SeedData.SeedDatabase(context);
        }
    }
}
```

The options pattern is used to configure the anti-forgery feature, through the `AntiforgeryOptions` class. The `HeaderName` property is used to specify the name of a header through which anti-forgery tokens will be accepted, which is `X-XSRF-TOKEN` in this case.

A custom middleware component is required to set the cookie, which is named `XSRF-TOKEN` in this example. The value of the cookie is obtained through the `IAntiForgery` service and must be configured with the `HttpOnly` option set to `false` so that the browser will allow JavaScript code to read the cookie.

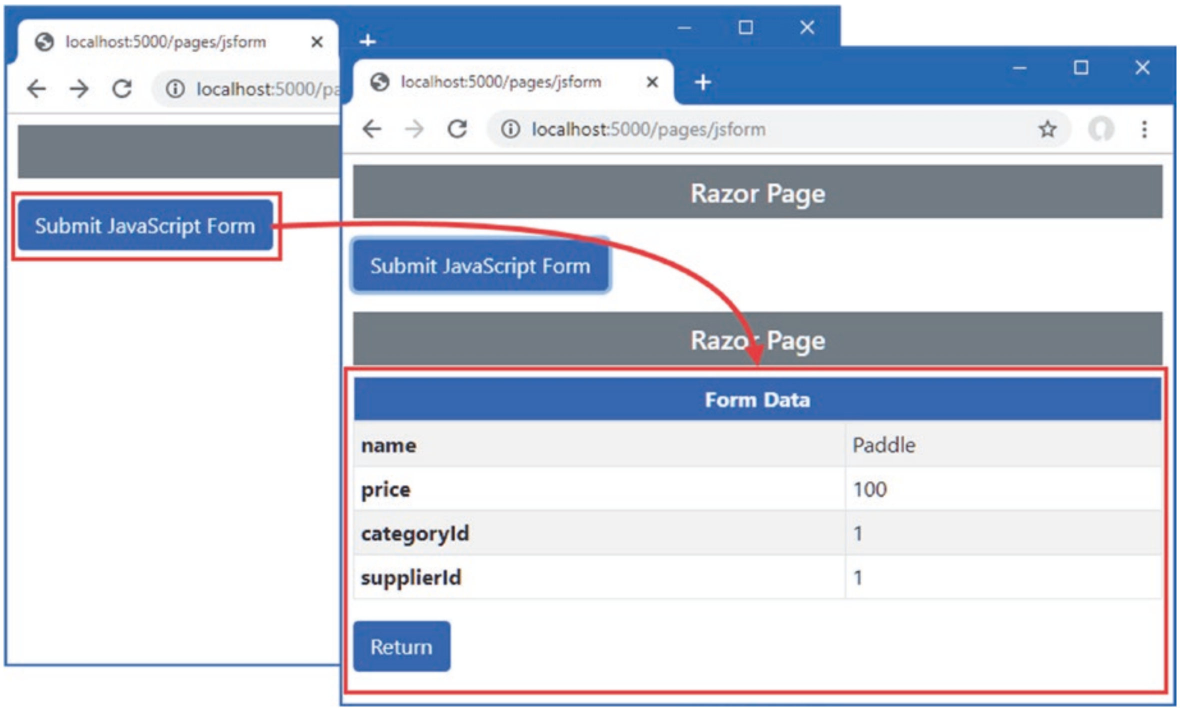
■ **Tip** I have followed the names that are supported by Angular in this example. Other frameworks follow their own conventions but can usually be configured to use any set of cookie and header names.

To create a simple JavaScript client that uses the cookie and header, add a Razor Page named `JavaScriptForm.cshtml` to the `Pages` folder with the content shown in Listing 27-32.

Listing 27-32. The Contents of the JavaScriptForm.cshtml File in the Pages Folder

```
@page "/pages/jsform"

<script type="text/javascript">
    async function sendRequest() {
        const token = document.cookie
            .replace(/(?:(?:^|.*;\s*)XSRF-TOKEN\s*\=\s*([^;]*).*$)|^.*$/, "$1");

        let form = new FormData();
        form.append("name", "Paddle");
        form.append("price", 100);
        form.append("categoryId", 1);
        form.append("supplierId", 1);
        let response = await fetch("@Url.Page("FormHandler")", {
            method: "POST",
            headers: { "X-XSRF-TOKEN": token },
            body: form
        });
        document.getElementById("content").innerHTML = await response.text();
    }

    document.addEventListener("DOMContentLoaded",
        () => document.getElementById("submit").onclick = sendRequest);
</script>

<button class="btn btn-primary m-2" id="submit">Submit JavaScript Form</button>
<div id="content"></div>
```

The JavaScript code in this Razor Page responds to a button click by sending an HTTP POST request to the `FormHandler` Razor Page. The value of the `XSRF-TOKEN` cookie is read and included in the `X-XSRF-TOKEN` request header. The response from the `FormHandler` page is a redirection to the `Results` page, which the browser will follow automatically. The response from the `Results` page is read by the JavaScript code and inserted into an element so it can be displayed to the user. To test the JavaScript code, use a browser to request `http://localhost:5000/pages/jsform` and click the button. The JavaScript code will submit the form and display the response, as shown in Figure 27-10.



name	Paddle
price	100
categoryId	1
supplierId	1

Figure 27-10. *Using a security token in JavaScript code*

Summary

In this chapter, I explained the features that ASP.NET Core provides for creating HTML forms. I showed you how tag helpers are used to select the form target and associate `input`, `textarea`, and `select` elements with view model or page model properties. In the next chapter, I describe the model binding feature, which extracts data from requests so that it can easily be consumed in action and handler methods.