

The generated HTML:

HTML

```
<a href="/Blogs/Home/AboutBlog">About Blog</a>
```

💡 Tip

To support areas in an MVC app, the route template must include a reference to the area, if it exists. That template is represented by the second parameter of the `routes.MapRoute` method call in *Startup.Configure*:

C#

```
app.UseMvc(routes =>
{
    // need route and attribute on controller: [Area("Blogs")]
    routes.MapRoute(name: "mvcAreaRoute",
        template: "{area:exists}/{controller=Home}/{action=Index}");

    // default route for non-areas
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

asp-protocol

The `asp-protocol` attribute is for specifying a protocol (such as `https`) in your URL. For example:

CSSHTML

```
<a asp-protocol="https"
    asp-controller="Home"
    asp-action="About">About</a>
```

The generated HTML:

HTML

```
<a href="https://localhost/Home/About">About</a>
```

The host name in the example is localhost. The Anchor Tag Helper uses the website's public domain when generating the URL.

asp-host

The `asp-host` attribute is for specifying a host name in your URL. For example:

C#HTML

```
<a asp-protocol="https"  
    asp-host="microsoft.com"  
    asp-controller="Home"  
    asp-action="About">About</a>
```

The generated HTML:

HTML

```
<a href="https://microsoft.com/Home/About">About</a>
```

asp-page

The `asp-page` attribute is used with Razor Pages. Use it to set an anchor tag's `href` attribute value to a specific page. Prefixing the page name with `/` creates a URL for a matching page from the root of the app:

With the sample code, the following markup creates a link to the attendee Razor Page:

C#HTML

```
<a asp-page="/Attendee">All Attendees</a>
```

The generated HTML:

HTML

```
<a href="/Attendee">All Attendees</a>
```

The `asp-page` attribute is mutually exclusive with the `asp-route`, `asp-controller`, and `asp-action` attributes. However, `asp-page` can be used with `asp-route-{value}` to control routing, as the following markup demonstrates:

C#HTML

```
<a asp-page="/Attendee"
    asp-route-attendeeid="10">View Attendee</a>
```

The generated HTML:

HTML

```
<a href="/Attendee?attendeeid=10">View Attendee</a>
```

If the referenced page doesn't exist, a link to the current page is generated using an [ambient value](#) from the request. No warning is indicated, except at the debug log level.

asp-page-handler

The [asp-page-handler](#) attribute is used with Razor Pages. It's intended for linking to specific page handlers.

Consider the following page handler:

C#

```
public void OnGetProfile(int attendeeId)
{
    ViewData["AttendeeId"] = attendeeId;

    // code omitted for brevity
}
```

The page model's associated markup links to the `OnGetProfile` page handler. Note the `On<Verb>` prefix of the page handler method name is omitted in the `asp-page-handler` attribute value. When the method is asynchronous, the `Async` suffix is omitted, too.

CSHTML

```
<a asp-page="/Attendee"
    asp-page-handler="Profile"
    asp-route-attendeeid="12">Attendee Profile</a>
```

The generated HTML:

HTML

```
<a href="/Attendee?attendeeid=12&handler=Profile">Attendee Profile</a>
```

Additional resources

- [Areas in ASP.NET Core](#)
- [Introduction to Razor Pages in ASP.NET Core](#)
- [Compatibility version for ASP.NET Core MVC](#)

Cache Tag Helper in ASP.NET Core MVC

Article • 06/17/2024

By [Peter Kellner](#)

The Cache Tag Helper provides the ability to improve the performance of your ASP.NET Core app by caching its content to the internal ASP.NET Core cache provider.

For an overview of Tag Helpers, see [Tag Helpers in ASP.NET Core](#).

The following Razor markup caches the current date:


```
CSHTML

<cache>@DateTime.Now</cache>
```

The first request to the page that contains the Tag Helper displays the current date. Additional requests show the cached value until the cache expires (default 20 minutes) or until the cached date is evicted from the cache.

Cache Tag Helper Attributes

enabled

 Expand table

Attribute Type	Examples	Default
Boolean	true, false	true

`enabled` determines if the content enclosed by the Cache Tag Helper is cached. The default is `true`. If set to `false`, the rendered output is **not** cached.

Example:

```
CSHTML

<cache enabled="true">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

expires-on

[Expand table](#)

Attribute Type	Example
DateTimeOffset	@new DateTime(2025,1,29,17,02,0)

`expires-on` sets an absolute expiration date for the cached item.

The following example caches the contents of the Cache Tag Helper until 5:02 PM on January 29, 2025:

CSHTML

```
<cache expires-on="@new DateTime(2025,1,29,17,02,0)">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

expires-after

[Expand table](#)

Attribute Type	Example	Default
TimeSpan	@TimeSpan.FromSeconds(120)	20 minutes

`expires-after` sets the length of time from the first request time to cache the contents.

Example:

CSHTML

```
<cache expires-after="@TimeSpan.FromSeconds(120)">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

The Razor View Engine sets the default `expires-after` value to twenty minutes.

expires-sliding

[Expand table](#)

Attribute Type	Example
TimeSpan	@TimeSpan.FromSeconds(60)

Sets the time that a cache entry should be evicted if its value hasn't been accessed.

Example:

CSHTML
<pre><cache expires-sliding="@TimeSpan.FromSeconds(60)"> Current Time Inside Cache Tag Helper: @DateTime.Now </cache></pre>

vary-by-header

[Expand table](#)

Attribute Type	Examples
String	User-Agent, User-Agent, content-encoding

`vary-by-header` accepts a comma-delimited list of header values that trigger a cache refresh when they change.

The following example monitors the header value `User-Agent`. The example caches the content for every different `User-Agent` presented to the web server:

CSHTML
<pre><cache vary-by-header="User-Agent"> Current Time Inside Cache Tag Helper: @DateTime.Now </cache></pre>

vary-by-query

[Expand table](#)

Attribute Type	Examples
String	Make, Make, Model

`vary-by-query` accepts a comma-delimited list of `Keys` in a query string (`Query`) that trigger a cache refresh when the value of any listed key changes.

The following example monitors the values of `Make` and `Model`. The example caches the content for every different `Make` and `Model` presented to the web server:

C#HTML

```
<cache vary-by-query="Make,Model">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

vary-by-route

[Expand table](#)

Attribute Type	Examples
String	<code>Make</code> , <code>Make,Model</code>

`vary-by-route` accepts a comma-delimited list of route parameter names that trigger a cache refresh when the route data parameter value changes.

Example:

`Startup.cs`:

C#

```
routes.MapRoute(
    name: "default",
    template: "{controller=Home}/{action=Index}/{Make?}/{Model?}");
```

`Index.cshtml`:

C#HTML

```
<cache vary-by-route="Make,Model">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

vary-by-cookie

[Expand table](#)

Attribute Type	Examples
String	<code>.AspNetCore.Identity.Application,</code> <code>.AspNetCore.Identity.Application, HairColor</code>

`vary-by-cookie` accepts a comma-delimited list of cookie names that trigger a cache refresh when the cookie values change.

The following example monitors the cookie associated with ASP.NET Core Identity. When a user is authenticated, a change in the Identity cookie triggers a cache refresh:

C#HTML

```
<cache vary-by-cookie=".AspNetCore.Identity.Application">  
    Current Time Inside Cache Tag Helper: @DateTime.Now  
</cache>
```

vary-by-user

[Expand table](#)

Attribute Type	Examples	Default
Boolean	<code>true</code> , <code>false</code>	<code>true</code>

`vary-by-user` specifies whether or not the cache resets when the signed-in user (or Context Principal) changes. The current user is also known as the Request Context Principal and can be viewed in a Razor view by referencing `@User.Identity.Name`.

The following example monitors the current logged in user to trigger a cache refresh:

C#HTML

```
<cache vary-by-user="true">  
    Current Time Inside Cache Tag Helper: @DateTime.Now  
</cache>
```

Using this attribute maintains the contents in cache through a sign-in and sign-out cycle. When the value is set to `true`, an authentication cycle invalidates the cache for the authenticated user. The cache is invalidated because a new unique cookie value is generated when a user is authenticated. Cache is maintained for the anonymous state

when no cookie is present or the cookie has expired. If the user is **not** authenticated, the cache is maintained.

vary-by

[Expand table](#)

Attribute Type	Example
String	@Model

`vary-by` allows for customization of what data is cached. When the object referenced by the attribute's string value changes, the content of the Cache Tag Helper is updated. Often, a string-concatenation of model values are assigned to this attribute. Effectively, this results in a scenario where an update to any of the concatenated values invalidates the cache.

The following example assumes the controller method rendering the view sums the integer value of the two route parameters, `myParam1` and `myParam2`, and returns the sum as the single model property. When this sum changes, the content of the Cache Tag Helper is rendered and cached again.

Action:

C#

```
public IActionResult Index(string myParam1, string myParam2, string myParam3)
{
    int num1;
    int num2;
    int.TryParse(myParam1, out num1);
    int.TryParse(myParam2, out num2);
    return View(viewName, num1 + num2);
}
```

Index.cshtml:

CSHTML

```
<cache vary-by="@Model">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

priority

 Expand table

Attribute Type	Examples	Default
CacheItemPriority	High, Low, NeverRemove, Normal	Normal

`priority` provides cache eviction guidance to the built-in cache provider. The web server evicts `Low` cache entries first when it's under memory pressure.

Example:

CHTML

```
<cache priority="High">  
    Current Time Inside Cache Tag Helper: @DateTime.Now  
</cache>
```

The `priority` attribute doesn't guarantee a specific level of cache retention.

`CacheItemPriority` is only a suggestion. Setting this attribute to `NeverRemove` doesn't guarantee that cached items are always retained. See the topics in the [Additional Resources](#) section for more information.

The Cache Tag Helper is dependent on the [memory cache service](#). The Cache Tag Helper adds the service if it hasn't been added.

Additional resources

- [Cache in-memory in ASP.NET Core](#)
- [Introduction to Identity on ASP.NET Core](#)

Component Tag Helper in ASP.NET Core

Article • 11/15/2024

The Component Tag Helper renders a Razor component in a Razor Pages page or MVC view.

Prerequisites

Follow the guidance in the *Use non-routable components in pages or views* section of the [Integrate ASP.NET Core Razor components with MVC or Razor Pages](#) article.


Component Tag Helper

To render a component from a page or view, use the [Component Tag Helper](#) (`<component>` tag).

`RenderMode` configures whether the component:


- Is prerendered into the page.
- Is rendered as static HTML on the page or if it includes the necessary information to bootstrap a Blazor app from the user agent.

Blazor WebAssembly app render modes are shown in the following table.

 Expand table

Render Mode	Description
<code>WebAssembly</code>	Renders a marker for a Blazor WebAssembly app for use to include an interactive component when loaded in the browser. The component isn't prerendered. This option makes it easier to render different Blazor WebAssembly components on different pages.
<code>WebAssemblyPrerendered</code>	Prerenders the component into static HTML and includes a marker for a Blazor WebAssembly app for later use to make the component interactive when loaded in the browser.

Render modes are shown in the following table.

 Expand table

Render Mode	Description
ServerPrerendered	Renders the component into static HTML and includes a marker for a server-side Blazor app. When the user-agent starts, this marker is used to bootstrap a Blazor app.
Server	Renders a marker for a server-side Blazor app. Output from the component isn't included. When the user-agent starts, this marker is used to bootstrap a Blazor app.
Static	Renders the component into static HTML.

Additional characteristics include:

- Multiple Component Tag Helpers rendering multiple Razor components is allowed.
- Components can't be dynamically rendered after the app has started.
- While pages and views can use components, the converse isn't true. Components can't use view- and page-specific features, such as partial views and sections. To use logic from a partial view in a component, factor out the partial view logic into a component.
- Rendering server components from a static HTML page isn't supported.

The following Component Tag Helper renders the `EmbeddedCounter` component in a page or view in a server-side Blazor app with `ServerPrerendered`:

C#HTML

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@using {APP ASSEMBLY}.Components

...

<component type="typeof(EmbeddedCounter)" render-mode="ServerPrerendered" />
```

The preceding example assumes that the `EmbeddedCounter` component is in the app's `Components` folder. The placeholder `{APP ASSEMBLY}` is the app's assembly name (for example, `@using BlazorSample.Components`).

The Component Tag Helper can also pass parameters to components. Consider the following `ColorfulCheckbox` component that sets the checkbox label's color and size.

`Components/ColorfulCheckbox.razor`:

razor

```

<label style="font-size:@(Size)px;color:@Color">
    <input @bind="Value"
        id="survey"
        name="blazor"
        type="checkbox" />
    Enjoying Blazor?
</label>

@code {
    [Parameter]
    public bool Value { get; set; }

    [Parameter]
    public int Size { get; set; } = 8;

    [Parameter]
    public string? Color { get; set; }

    protected override void OnInitialized()
    {
        Size += 10;
    }
}

```

The `Size` (`int`) and `Color` (`string`) [component parameters](#) can be set by the Component Tag Helper:

CSHTML

```

@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@using {APP ASSEMBLY}.Components

...

<component type="typeof(ColorfulCheckbox)" render-mode="ServerPrerendered"
    param-Size="14" param-Color="@("blue")" />

```

The preceding example assumes that the `ColorfulCheckbox` component is in the `Components` folder. The placeholder `{APP ASSEMBLY}` is the app's assembly name (for example, `@using BlazorSample.Components`).

The following HTML is rendered in the page or view:

HTML

```

<label style="font-size:24px;color:blue">
    <input id="survey" name="blazor" type="checkbox">

```

```
Enjoying Blazor?  
</label>
```

Passing a quoted string requires an [explicit Razor expression](#), as shown for `param-Color` in the preceding example. The Razor parsing behavior for a `string` type value doesn't apply to a `param-*` attribute because the attribute is an `object` type.

All types of parameters are supported, except:

- Generic parameters.
- Non-serializable parameters.
- Inheritance in collection parameters.
- Parameters whose type is defined outside of the Blazor WebAssembly app or within a lazily-loaded assembly.
- For receiving a [RenderFragment delegate for child content](#) (for example, `param-ChildContent="..."`). For this scenario, we recommend creating a Razor component (`.razor`) that references the component you want to render with the child content you want to pass and then invoke the Razor component from the page or view with the Component Tag Helper.

The parameter type must be JSON serializable, which typically means that the type must have a default constructor and settable properties. For example, you can specify a value for `Size` and `Color` in the preceding example because the types of `Size` and `Color` are primitive types (`int` and `string`), which are supported by the JSON serializer.

In the following example, a class object is passed to the component:

`MyClass.cs`:

C#

```
public class MyClass  
{  
    public MyClass()  
    {  
    }  
  
    public int MyInt { get; set; } = 999;  
    public string MyString { get; set; } = "Initial value";  
}
```

The class must have a public parameterless constructor.

`Components/ParameterComponent.razor`:

razor

```
<h2>ParameterComponent</h2>

<p>Int: @MyObject?.MyInt</p>
<p>String: @MyObject?.MyString</p>

@code
{
    [Parameter]
    public MyClass? MyObject { get; set; }
}
```

Pages/MyPage.cshtml:

CSHTML

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@using {APP ASSEMBLY}
@using {APP ASSEMBLY}.Components

...

@{
    var myObject = new MyClass();
    myObject.MyInt = 7;
    myObject.MyString = "Set by MyPage";
}

<component type="typeof(ParameterComponent)" render-mode="ServerPrerendered"
    param-MyObject="@myObject" />
```

The preceding example assumes that the `ParameterComponent` component is in the app's `Components` folder. The placeholder `{APP ASSEMBLY}` is the app's assembly name (for example, `@using BlazorSample` and `@using BlazorSample.Components`). `MyClass` is in the app's namespace.

Additional resources

- [Persist Component State Tag Helper in ASP.NET Core](#)
- [Prerender ASP.NET Core Razor components](#)
- [ComponentTagHelper](#)
- [Tag Helpers in ASP.NET Core](#)
- [ASP.NET Core Razor components](#)

Distributed Cache Tag Helper in ASP.NET Core

Article • 06/17/2024

By [Peter Kellner](#) 

The Distributed Cache Tag Helper provides the ability to dramatically improve the performance of your ASP.NET Core app by caching its content to a distributed cache source.

For an overview of Tag Helpers, see [Tag Helpers in ASP.NET Core](#).

The Distributed Cache Tag Helper inherits from the same base class as the Cache Tag Helper. All of the [Cache Tag Helper](#) attributes are available to the Distributed Tag Helper.

The Distributed Cache Tag Helper uses [constructor injection](#). The [IDistributedCache](#) interface is passed into the Distributed Cache Tag Helper's constructor. If no concrete implementation of `IDistributedCache` is created in `Startup.ConfigureServices` (`Startup.cs`), the Distributed Cache Tag Helper uses the same in-memory provider for storing cached data as the [Cache Tag Helper](#).

Distributed Cache Tag Helper Attributes

Attributes shared with the Cache Tag Helper

- `enabled`
- `expires-on`
- `expires-after`
- `expires-sliding`
- `vary-by-header`
- `vary-by-query`
- `vary-by-route`
- `vary-by-cookie`
- `vary-by-user`
- `vary-by`
- `priority`

The Distributed Cache Tag Helper inherits from the same class as Cache Tag Helper. For descriptions of these attributes, see the [Cache Tag Helper](#).

name

 Expand table

Attribute Type	Example
String	<code>my-distributed-cache-unique-key-101</code>

`name` is required. The `name` attribute is used as a key for each stored cache instance. Unlike the Cache Tag Helper that assigns a cache key to each instance based on the Razor page name and location in the Razor page, the Distributed Cache Tag Helper only bases its key on the attribute `name`.

Example:

C#HTML

```
<distributed-cache name="my-distributed-cache-unique-key-101">  
    Time Inside Cache Tag Helper: @DateTime.Now  
</distributed-cache>
```

Distributed Cache Tag Helper `IDistributedCache` implementations

There are two implementations of `IDistributedCache` built in to ASP.NET Core. One is based on SQL Server, and the other is based on Redis. Third-party implementations are also available, such as [NCache](#). Details of these implementations can be found at [Distributed caching in ASP.NET Core](#). Both implementations involve setting an instance of `IDistributedCache` in `Startup`.

There are no tag attributes specifically associated with using any specific implementation of `IDistributedCache`.

Additional resources

- [Cache Tag Helper in ASP.NET Core MVC](#)
- [Dependency injection in ASP.NET Core](#)
- [Distributed caching in ASP.NET Core](#)
- [Cache in-memory in ASP.NET Core](#)
- [Introduction to Identity on ASP.NET Core](#)

Environment Tag Helper in ASP.NET Core

Article • 06/03/2022

By [Peter Kellner](#) and [Hisham Bin Ateya](#)

The Environment Tag Helper conditionally renders its enclosed content based on the current [hosting environment](#). The Environment Tag Helper's single attribute, `names`, is a comma-separated list of environment names. If any of the provided environment names match the current environment, the enclosed content is rendered.

For an overview of Tag Helpers, see [Tag Helpers in ASP.NET Core](#).

Environment Tag Helper Attributes

`names`

`names` accepts a single hosting environment name or a comma-separated list of hosting environment names that trigger the rendering of the enclosed content.

Environment values are compared to the current value returned by [IWebHostEnvironment.EnvironmentName](#). The comparison ignores case.

The following example uses an Environment Tag Helper. The content is rendered if the hosting environment is Staging or Production:

C#HTML

```
<environment names="Staging,Production">
    <strong>IWebHostEnvironment.EnvironmentName is Staging or
    Production</strong>
</environment>
```

include and exclude attributes

`include` & `exclude` attributes control rendering the enclosed content based on the included or excluded hosting environment names.

include

The `include` property exhibits similar behavior to the `names` attribute. An environment listed in the `include` attribute value must match the app's hosting environment (`IWebHostEnvironment.EnvironmentName`) to render the content of the `<environment>` tag.

CSSHTML

```
<environment include="Staging,Production">
    <strong>IWebHostEnvironment.EnvironmentName is Staging or
Production</strong>
</environment>
```

exclude

In contrast to the `include` attribute, the content of the `<environment>` tag is rendered when the hosting environment doesn't match an environment listed in the `exclude` attribute value.

CSSHTML

```
<environment exclude="Development">
    <strong>IWebHostEnvironment.EnvironmentName is not Development</strong>
</environment>
```

Additional resources

- [Use multiple environments in ASP.NET Core](#)

Tag Helpers in forms in ASP.NET Core

Article • 09/27/2024

By [Rick Anderson](#), [N. Taylor Mullen](#), [Dave Paquette](#), and [Jerrie Pelser](#)

This document demonstrates working with Forms and the HTML elements commonly used on a Form. The HTML [Form](#) element provides the primary mechanism web apps use to post back data to the server. Most of this document describes [Tag Helpers](#) and how they can help you productively create robust HTML forms. We recommend you read [Introduction to Tag Helpers](#) before you read this document.

In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers don't replace HTML Helpers and there's not a Tag Helper for each HTML Helper. When an HTML Helper alternative exists, it's mentioned.

The Form Tag Helper

The [Form Tag Helper](#):

- Generates the HTML [<FORM>](#) `action` attribute value for a MVC controller action or named route
- Generates a hidden [Request Verification Token](#) to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method)
- Provides the `asp-route-<Parameter Name>` attribute, where `<Parameter Name>` is added to the route values. The `routeValues` parameters to `Html.BeginForm` and `Html.BeginRouteForm` provide similar functionality.
- Has an HTML Helper alternative `Html.BeginForm` and `Html.BeginRouteForm`

Sample:

C#HTML

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

The Form Tag Helper above generates the following HTML:

HTML

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

The MVC runtime generates the `action` attribute value from the Form Tag Helper attributes `asp-controller` and `asp-action`. The Form Tag Helper also generates a hidden [Request Verification Token](#) to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method). Protecting a pure HTML Form from cross-site request forgery is difficult, the Form Tag Helper provides this service for you.

Using a named route

The `asp-route` Tag Helper attribute can also generate markup for the HTML `action` attribute. An app with a [route](#) named `register` could use the following markup for the registration page:

C#HTML

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Many of the views in the *Views/Account* folder (generated when you create a new web app with *Individual User Accounts*) contain the `asp-route-returnurl` attribute:

C#HTML

```
<form asp-controller="Account" asp-action="Login"
      asp-route-returnurl="@ViewData["ReturnUrl"]"
      method="post" class="form-horizontal" role="form">
```

ⓘ Note

With the built in templates, `returnUrl` is only populated automatically when you try to access an authorized resource but are not authenticated or authorized. When you attempt an unauthorized access, the security middleware redirects you to the login page with the `returnUrl` set.

The Form Action Tag Helper

The Form Action Tag Helper generates the `formaction` attribute on the generated `<button ...>` or `<input type="image" ...>` tag. The `formaction` attribute controls where a form submits its data. It binds to `<input>` [↗](#) elements of type `image` and `<button>` [↗](#) elements. The Form Action Tag Helper enables the usage of several [AnchorTagHelper](#) `asp-` attributes to control what `formaction` link is generated for the corresponding element.

Supported [AnchorTagHelper](#) attributes to control the value of `formaction`:

[↗](#) Expand table

Attribute	Description
asp-controller	The name of the controller.
asp-action	The name of the action method.
asp-area	The name of the area.
asp-page	The name of the Razor page.
asp-page-handler	The name of the Razor page handler.
asp-route	The name of the route.
asp-route-{value}	A single URL route value. For example, <code>asp-route-id="1234"</code> .
asp-all-route-data	All route values.
asp-fragment	The URL fragment.

Submit to controller example

The following markup submits the form to the `Index` action of `HomeController` when the input or button are selected:

C#HTML

```
<form method="post">
  <button asp-controller="Home" asp-action="Index">Click Me</button>
  <input type="image" src="..." alt="Or Click Me" asp-controller="Home"
        asp-action="Index">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
  <button formaction="/Home">Click Me</button>
  <input type="image" src="..." alt="Or Click Me" formaction="/Home">
</form>
```

Submit to page example

The following markup submits the form to the `About` Razor Page:

C#HTML

```
<form method="post">
  <button asp-page="About">Click Me</button>
  <input type="image" src="..." alt="Or Click Me" asp-page="About">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
  <button formaction="/About">Click Me</button>
  <input type="image" src="..." alt="Or Click Me" formaction="/About">
</form>
```

Submit to route example

Consider the `/Home/Test` endpoint:

C#

```
public class HomeController : Controller
{
    [Route("/Home/Test", Name = "Custom")]
    public string Test()
    {
        return "This is the test page";
    }
}
```

The following markup submits the form to the `/Home/Test` endpoint.

C#HTML

```
<form method="post">
    <button asp-route="Custom">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-route="Custom">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/Home/Test">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home/Test">
</form>
```

The Input Tag Helper

The Input Tag Helper binds an HTML `<input>` [↗](#) element to a model expression in your razor view.

Syntax:

C#HTML

```
<input asp-for="<Expression Name>">
```

The Input Tag Helper:

- Generates the `id` and `name` HTML attributes for the expression name specified in the `asp-for` attribute. `asp-for="Property1.Property2"` is equivalent to `m => m.Property1.Property2`. The name of the expression is what is used for the `asp-for` attribute value. See the [Expression names](#) section for additional information.
- Sets the HTML `type` attribute value based on the model type and [data annotation](#) attributes applied to the model property
- Won't overwrite the HTML `type` attribute value when one is specified
- Generates [HTML5](#) [↗](#) validation attributes from [data annotation](#) attributes applied to model properties
- Has an HTML Helper feature overlap with `Html.TextBoxFor` and `Html.EditorFor`. See the [HTML Helper alternatives to Input Tag Helper](#) section for details.

- Provides strong typing. If the name of the property changes and you don't update the Tag Helper you'll get an error similar to the following:

```
An error occurred during the compilation of a resource required to
process
this request. Please review the following specific error details and
modify
your source code appropriately.
```

```
Type expected
'RegisterViewModel' does not contain a definition for 'Email' and no
extension method 'Email' accepting a first argument of type
'RegisterViewModel'
could be found (are you missing a using directive or an assembly
reference?)
```

The `Input` Tag Helper sets the HTML `type` attribute based on the .NET type. The following table lists some common .NET types and generated HTML type (not every .NET type is listed).

[Expand table](#)

.NET type	Input Type
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime-local" ↗
Byte	type="number"
Int	type="number"
Single, Double	type="number"

The following table shows some common [data annotations](#) attributes that the input tag helper will map to specific input types (not every validation attribute is listed):

[Expand table](#)

Attribute	Input Type
[EmailAddress]	type="email"
[Url]	type="url"

Attribute	Input Type
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

CSHTML

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <button type="submit">Register</button>
</form>
```

The code above generates the following HTML:

HTML

```
<form method="post" action="/Demo/RegisterInput">
    Email:
    <input type="email" data-val="true"
```

```

        data-val-email="The Email Address field is not a valid email
address."
        data-val-required="The Email Address field is required."
        id="Email" name="Email" value=""><br>
    Password:
    <input type="password" data-val="true"
        data-val-required="The Password field is required."
        id="Password" name="Password"><br>
    <button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>

```

The data annotations applied to the `Email` and `Password` properties generate metadata on the model. The Input Tag Helper consumes the model metadata and produces [HTML5](#) `data-val-*` attributes (see [Model Validation](#)). These attributes describe the validators to attach to the input fields. This provides unobtrusive HTML5 and [jQuery](#) validation. The unobtrusive attributes have the format `data-val-rule="Error Message"`, where rule is the name of the validation rule (such as `data-val-required`, `data-val-email`, `data-val-maxlength`, etc.) If an error message is provided in the attribute, it's displayed as the value for the `data-val-rule` attribute. There are also attributes of the form `data-val-ruleName-argumentName="argumentValue"` that provide additional details about the rule, for example, `data-val-maxlength-max="1024"`.

When binding multiple `input` controls to the same property, the generated controls share the same `id`, which makes the generated mark-up invalid. To prevent duplicates, specify the `id` attribute for each control explicitly.

Checkbox hidden input rendering

Checkboxes in HTML5 don't submit a value when they're unchecked. To enable a default value to be sent for an unchecked checkbox, the Input Tag Helper generates an additional hidden input for checkboxes.

For example, consider the following Razor markup that uses the Input Tag Helper for a boolean model property `IsChecked`:

C#HTML

```

<form method="post">
    <input asp-for="@Model.IsChecked" />
    <button type="submit">Submit</button>
</form>

```

The preceding Razor markup generates HTML markup similar to the following:

HTML

```
<form method="post">
  <input name="IsChecked" type="checkbox" value="true" />
  <button type="submit">Submit</button>

  <input name="IsChecked" type="hidden" value="false" />
</form>
```

The preceding HTML markup shows an additional hidden input with a name of `IsChecked` and a value of `false`. By default, this hidden input is rendered at the end of the form. When the form is submitted:

- If the `IsChecked` checkbox input is checked, both `true` and `false` are submitted as values.
- If the `IsChecked` checkbox input is unchecked, only the hidden input value `false` is submitted.

The ASP.NET Core model-binding process reads only the first value when binding to a `bool` value, which results in `true` for checked checkboxes and `false` for unchecked checkboxes.

To configure the behavior of the hidden input rendering, set the [CheckBoxHiddenInputRenderMode](#) property on [MvcViewOptions.HtmlHelperOptions](#). For example:

C#

```
services.Configure<MvcViewOptions>(options =>
    options.HtmlHelperOptions.CheckBoxHiddenInputRenderMode =
        CheckBoxHiddenInputRenderMode.None);
```

The preceding code disables hidden input rendering for checkboxes by setting `CheckBoxHiddenInputRenderMode` to [CheckBoxHiddenInputRenderMode.None](#). For all available rendering modes, see the [CheckBoxHiddenInputRenderMode](#) enum.

HTML Helper alternatives to Input Tag Helper

`Html.TextBox`, `Html.TextBoxFor`, `Html.Editor` and `Html.EditorFor` have overlapping features with the Input Tag Helper. The Input Tag Helper will automatically set the `type` attribute; `Html.TextBox` and `Html.TextBoxFor` won't. `Html.Editor` and `Html.EditorFor` handle collections, complex objects and templates; the Input Tag Helper doesn't. The

Input Tag Helper, `Html.EditorFor` and `Html.TextBoxFor` are strongly typed (they use lambda expressions); `Html.TextBox` and `Html.Editor` are not (they use expression names).

HtmlAttributes

`@Html.Editor()` and `@Html.EditorFor()` use a special `ViewDataDictionary` entry named `htmlAttributes` when executing their default templates. This behavior is optionally augmented using `additionalViewData` parameters. The key "htmlAttributes" is case-insensitive. The key "htmlAttributes" is handled similarly to the `htmlAttributes` object passed to input helpers like `@Html.TextBox()`.

C#HTML

```
@Html.EditorFor(model => model.YourProperty,  
    new { htmlAttributes = new { @class="myCssClass", style="width:100px" } })
```

Expression names

The `asp-for` attribute value is a `ModelExpression` and the right hand side of a lambda expression. Therefore, `asp-for="Property1"` becomes `m => m.Property1` in the generated code which is why you don't need to prefix with `Model`. You can use the "@" character to start an inline expression and move before the `m.`:

C#HTML

```
@{  
    var joe = "Joe";  
}  
  
<input asp-for="@joe">
```

Generates the following:

HTML

```
<input type="text" id="joe" name="joe" value="Joe">
```

With collection properties, `asp-for="CollectionProperty[23].Member"` generates the same name as `asp-for="CollectionProperty[i].Member"` when `i` has the value `23`.

When ASP.NET Core MVC calculates the value of `ModelExpression`, it inspects several sources, including `ModelState`. Consider `<input type="text" asp-for="Name">`. The calculated `value` attribute is the first non-null value from:

- `ModelState` entry with key "Name".
- Result of the expression `Model.Name`.

Navigating child properties

You can also navigate to child properties using the property path of the view model. Consider a more complex model class that contains a child `Address` property.

C#

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

C#

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

In the view, we bind to `Address.AddressLine1`:

CSHTML

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <label>Address: <input asp-for="Address.AddressLine1" /></label><br />
    <button type="submit">Register</button>
</form>
```

The following HTML is generated for `Address.AddressLine1`:

HTML

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1" value="">
```

Expression names and Collections

Sample, a model containing an array of `Colors`:

C#

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

The action method:

C#

```
public IActionResult Edit(int id, int colorIndex)
{
    ViewData["Index"] = colorIndex;
    return View(GetPerson(id));
}
```

The following Razor shows how you access a specific `Color` element:

CSHTML

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

The `Views/Shared/EditorTemplates/String.cshtml` template:

CSHTML


```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

Sample using `List<T>`:

```
C#

public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

The following Razor shows how to iterate over a collection:

```
C#HTML

@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

The `Views/Shared/EditorTemplates/ToDoItem.cshtml` template:

```
C#HTML

@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
```

```
</td>
```

```
@*
```

```
    This template replaces the following Razor which evaluates the indexer  
    three times.
```

```
    <td>
```

```
        <label asp-for="@Model[i].Name"></label>
```

```
        @Html.DisplayFor(model => model[i].Name)
```

```
    </td>
```

```
    <td>
```

```
        <input asp-for="@Model[i].IsDone" />
```

```
    </td>
```

```
*@
```

`foreach` should be used if possible when the value is going to be used in an `asp-for` or `Html.DisplayFor` equivalent context. In general, `for` is better than `foreach` (if the scenario allows it) because it doesn't need to allocate an enumerator; however, evaluating an indexer in a LINQ expression can be expensive and should be minimized.

⚠ Note

The commented sample code above shows how you would replace the lambda expression with the `@` operator to access each `ToDoItem` in the list.

The Textarea Tag Helper

The `Textarea Tag Helper` tag helper is similar to the `Input Tag Helper`.

- Generates the `id` and `name` attributes, and the data validation attributes from the model for a `<textarea>` [↗](#) element.
- Provides strong typing.
- HTML Helper alternative: `Html.TextAreaFor`

Sample:

```
C#
```

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

```
namespace FormsTagHelper.ViewModels
```

```
{
```

```
    public class DescriptionViewModel
```

```

{
    [MinLength(5)]
    [MaxLength(1024)]
    public string Description { get; set; }
}

```

C#HTML

```

@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>

```

The following HTML is generated:

HTML

```

<form method="post" action="/Demo/RegisterTextArea">
    <textarea data-val="true"
        data-val-maxlength="The field Description must be a string or array type
with a maximum length of &#x27;1024&#x27;."
        data-val-maxlength-max="1024"
        data-val-minlength="The field Description must be a string or array type
with a minimum length of &#x27;5&#x27;."
        data-val-minlength-min="5"
        id="Description" name="Description">
    </textarea>
    <button type="submit">Test</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>

```

The Label Tag Helper

- Generates the label caption and `for` attribute on a `<label>` [↗](#) element for an expression name
- HTML Helper alternative: `Html.LabelFor`.

The `Label Tag Helper` provides the following benefits over a pure HTML label element:

- You automatically get the descriptive label value from the `Display` attribute. The intended display name might change over time, and the combination of `Display` attribute and Label Tag Helper will apply the `Display` everywhere it's used.

- Less markup in source code
- Strong typing with the model property.

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

CSHTML

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

The following HTML is generated for the `<label>` element:

HTML

```
<label for="Email">Email Address</label>
```

The Label Tag Helper generated the `for` attribute value of "Email", which is the ID associated with the `<input>` element. The Tag Helpers generate consistent `id` and `for` elements so they can be correctly associated. The caption in this sample comes from the `Display` attribute. If the model didn't contain a `Display` attribute, the caption would be the property name of the expression. To override the default caption, add a caption inside the label tag.

The Validation Tag Helpers

There are two Validation Tag Helpers. The `Validation Message Tag Helper` (which displays a validation message for a single property on your model), and the `Validation Summary Tag Helper` (which displays a summary of validation errors). The `Input Tag Helper` adds HTML5 client side validation attributes to input elements based on data annotation attributes on your model classes. Validation is also performed on the server. The Validation Tag Helper displays these error messages when a validation error occurs.

The Validation Message Tag Helper

- Adds the [HTML5](#) `data-valmsg-for="property"` attribute to the [span](#) element, which attaches the validation error messages on the input field of the specified model property. When a client side validation error occurs, [jQuery](#) displays the error message in the `` element.
- Validation also takes place on the server. Clients may have JavaScript disabled and some validation can only be done on the server side.
- HTML Helper alternative: `Html.ValidationMessageFor`

The `Validation Message Tag Helper` is used with the `asp-validation-for` attribute on an HTML [span](#) element.

C#HTML

```
<span asp-validation-for="Email"></span>
```

The Validation Message Tag Helper will generate the following HTML:

HTML

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

You generally use the `Validation Message Tag Helper` after an `Input Tag Helper` for the same property. Doing so displays any validation error messages near the input that caused the error.

ⓘ Note

You must have a view with the correct JavaScript and [jQuery](#) script references in place for client side validation. See [Model Validation](#) for more information.

When a server side validation error occurs (for example when you have custom server side validation or client-side validation is disabled), MVC places that error message as the body of the `` element.

HTML

```
<span class="field-validation-error" data-valmsg-for="Email"
      data-valmsg-replace="true">
    The Email Address field is required.
</span>
```

The Validation Summary Tag Helper

- Targets `<div>` elements with the `asp-validation-summary` attribute
- HTML Helper alternative: `@Html.ValidationSummary`

The `Validation Summary Tag Helper` is used to display a summary of validation messages. The `asp-validation-summary` attribute value can be any of the following:

[Expand table](#)

<code>asp-validation-summary</code>	Validation messages displayed
All	Property and model level
ModelOnly	Model
None	None

Sample

In the following example, the data model has `DataAnnotation` attributes, which generates validation error messages on the `<input>` element. When a validation error occurs, the Validation Tag Helper displays the error message:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
```

```

    [EmailAddress]
    [Display(Name = "Email Address")]
    public string Email { get; set; }

    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }
}

```

CSHTML

```

@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    <label>Email: <input asp-for="Email" /></label> <br />
    <span asp-validation-for="Email"></span><br />
    <label>Password: <input asp-for="Password" /></label><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>

```

The generated HTML (when the model is valid):

HTML

```

<form action="/DemoReg/Register" method="post">
    <label>Email: <input name="Email" id="Email" type="email" value=""
        data-val-required="The Email field is required."
        data-val-email="The Email field is not a valid email address."
        data-val="true"></label><br>
    <span class="field-validation-valid" data-valmsg-replace="true"
        data-valmsg-for="Email"></span><br>
    <label>Password: <input name="Password" id="Password" type="password"
        data-val-required="The Password field is required." data-val="true">
    </label><br>
    <span class="field-validation-valid" data-valmsg-replace="true"
        data-valmsg-for="Password"></span><br>
    <button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for
    brevity>">
</form>

```

The Select Tag Helper

- Generates [select](#) and associated [option](#) elements for properties of your model.
- Has an HTML Helper alternative `Html.DropDownListFor` and `Html.ListBoxFor`

The `Select Tag Helper` `asp-for` specifies the model property name for the [select](#) element and `asp-items` specifies the [option](#) elements. For example:

C#HTML

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Sample:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new
List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA" },
        };
    }
}
```

The `Index` method initializes the `CountryViewModel`, sets the selected country and passes it to the `Index` view.

C#

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

The HTTP POST `Index` method displays the selection:

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
```



```

public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country + " selected";
        return RedirectToAction("IndexSuccess", new { message = msg });
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}

```

The `Index` view:

C#HTML

```

@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>

```

Which generates the following HTML (with "CA" selected):

HTML

```

<form method="post" action="/">
    <select id="Country" name="Country">
        <option value="MX">Mexico</option>
        <option selected="selected" value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>

```

⚠ Note

We don't recommend using `ViewBag` or `ViewData` with the Select Tag Helper. A view model is more robust at providing MVC metadata and generally less problematic.

The `asp-for` attribute value is a special case and doesn't require a `Model` prefix, the other Tag Helper attributes do (such as `asp-items`)

C#HTML

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Enum binding

It's often convenient to use `<select>` with an `enum` property and generate the `SelectListItem` elements from the `enum` values.

Sample:

C#

```
public class CountryEnumViewModel
{
    public CountryEnum EnumCountry { get; set; }
}
```

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The `GetEnumSelectList` method generates a `SelectList` object for an enum.

C#HTML

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
        asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
```

```
<br /><button type="submit">Register</button>
</form>
```

You can mark your enumerator list with the `Display` attribute to get a richer UI:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The following HTML is generated:

HTML

```
<form method="post" action="/Home/IndexEnum">
  <select data-val="true" data-val-required="The EnumCountry field is
  required."
    id="EnumCountry" name="EnumCountry">
    <option value="0">United Mexican States</option>
    <option value="1">United States of America</option>
    <option value="2">Canada</option>
    <option value="3">France</option>
    <option value="4">Germany</option>
    <option selected="selected" value="5">Spain</option>
  </select>
  <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed
  for brevity>">
</form>
```

Option Group

The HTML `<optgroup>` [↗](#) element is generated when the view model contains one or more `SelectListGroup` objects.

The `CountryViewModelGroup` groups the `SelectListItem` elements into the "North America" and "Europe" groups:

C#

```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America" };
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
        };
    }
}
```

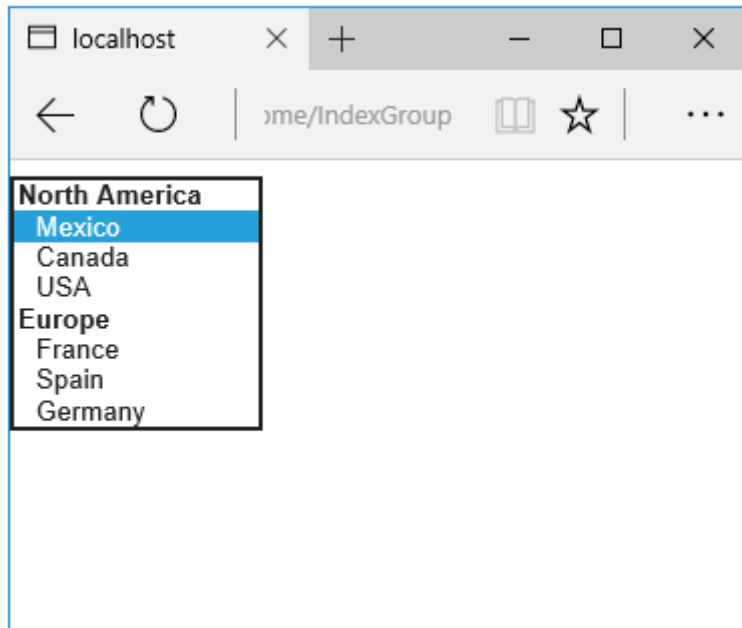
```

public string Country { get; set; }

public List<SelectListItem> Countries { get; }

```

The two groups are shown below:



The generated HTML:

HTML

```

<form method="post" action="/Home/IndexGroup">
  <select id="Country" name="Country">
    <optgroup label="North America">
      <option value="MEX">Mexico</option>
      <option value="CAN">Canada</option>
      <option value="US">USA</option>
    </optgroup>
    <optgroup label="Europe">
      <option value="FR">France</option>
      <option value="ES">Spain</option>
      <option value="DE">Germany</option>
    </optgroup>
  </select>
  <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>

```

Multiple select

The Select Tag Helper will automatically generate the `multiple = "multiple"` attribute if the property specified in the `asp-for` attribute is an `IEnumerable`. For example, given the following model:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new
List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA" },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain" },
            new SelectListItem { Value = "DE", Text = "Germany" }
        };
    }
}
```

With the following view:

C#HTML

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Generates the following HTML:

HTML

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
        multiple="multiple"
        name="CountryCodes"><option value="MX">Mexico</option>
    <option value="CA">Canada</option>
    <option value="US">USA</option>
    <option value="FR">France</option>
```

```

<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>

```

No selection

If you find yourself using the "not specified" option in multiple pages, you can create a template to eliminate repeating the HTML:

CSSHTML

```

@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>

```

The `Views/Shared/EditorTemplates/CountryViewModel.csshtml` template:

CSSHTML

```

@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>

```

Adding HTML `<option>` [↗](#) elements isn't limited to the *No selection* case. For example, the following view and action method will generate HTML similar to the code above:

C#

```

public IActionResult IndexNone()
{
    var model = new CountryViewModel();
    model.Countries.Insert(0, new SelectListItem("<none>", ""));
    return View(model);
}

```

CSSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

The correct `<option>` element will be selected (contain the `selected="selected"` attribute) depending on the current `Country` value.

C#

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

HTML

```
<form method="post" action="/Home/IndexEmpty">
    <select id="Country" name="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA" selected="selected">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

Additional resources

- [Tag Helpers in ASP.NET Core](#)
- [HTML Form element](#) [↗](#)
- [Request Verification Token](#)
- [Model Binding in ASP.NET Core](#)
- [Model validation in ASP.NET Core MVC](#)
- [IAttributeAdapter Interface](#)

- [Code snippets for this document](#) ↗

Tag Helpers in forms in ASP.NET Core

Article • 09/27/2024

By [Rick Anderson](#), [N. Taylor Mullen](#), [Dave Paquette](#), and [Jerrie Pelser](#)

This document demonstrates working with Forms and the HTML elements commonly used on a Form. The HTML [Form](#) element provides the primary mechanism web apps use to post back data to the server. Most of this document describes [Tag Helpers](#) and how they can help you productively create robust HTML forms. We recommend you read [Introduction to Tag Helpers](#) before you read this document.

In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers don't replace HTML Helpers and there's not a Tag Helper for each HTML Helper. When an HTML Helper alternative exists, it's mentioned.

The Form Tag Helper

The [Form Tag Helper](#):

- Generates the HTML [<FORM>](#) `action` attribute value for a MVC controller action or named route
- Generates a hidden [Request Verification Token](#) to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method)
- Provides the `asp-route-<Parameter Name>` attribute, where `<Parameter Name>` is added to the route values. The `routeValues` parameters to `Html.BeginForm` and `Html.BeginRouteForm` provide similar functionality.
- Has an HTML Helper alternative `Html.BeginForm` and `Html.BeginRouteForm`

Sample:

C#HTML

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

The Form Tag Helper above generates the following HTML:

HTML

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

The MVC runtime generates the `action` attribute value from the Form Tag Helper attributes `asp-controller` and `asp-action`. The Form Tag Helper also generates a hidden [Request Verification Token](#) to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method). Protecting a pure HTML Form from cross-site request forgery is difficult, the Form Tag Helper provides this service for you.

Using a named route

The `asp-route` Tag Helper attribute can also generate markup for the HTML `action` attribute. An app with a [route](#) named `register` could use the following markup for the registration page:

C#HTML

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Many of the views in the *Views/Account* folder (generated when you create a new web app with *Individual User Accounts*) contain the `asp-route-returnurl` attribute:

C#HTML

```
<form asp-controller="Account" asp-action="Login"
      asp-route-returnurl="@ViewData["ReturnUrl"]"
      method="post" class="form-horizontal" role="form">
```

ⓘ Note

With the built in templates, `returnUrl` is only populated automatically when you try to access an authorized resource but are not authenticated or authorized. When you attempt an unauthorized access, the security middleware redirects you to the login page with the `returnUrl` set.

The Form Action Tag Helper

The Form Action Tag Helper generates the `formaction` attribute on the generated `<button ...>` or `<input type="image" ...>` tag. The `formaction` attribute controls where a form submits its data. It binds to `<input>` [↗](#) elements of type `image` and `<button>` [↗](#) elements. The Form Action Tag Helper enables the usage of several [AnchorTagHelper](#) `asp-` attributes to control what `formaction` link is generated for the corresponding element.

Supported [AnchorTagHelper](#) attributes to control the value of `formaction`:

[↗](#) Expand table

Attribute	Description
asp-controller	The name of the controller.
asp-action	The name of the action method.
asp-area	The name of the area.
asp-page	The name of the Razor page.
asp-page-handler	The name of the Razor page handler.
asp-route	The name of the route.
asp-route-{value}	A single URL route value. For example, <code>asp-route-id="1234"</code> .
asp-all-route-data	All route values.
asp-fragment	The URL fragment.

Submit to controller example

The following markup submits the form to the `Index` action of `HomeController` when the input or button are selected:

C#HTML

```
<form method="post">
  <button asp-controller="Home" asp-action="Index">Click Me</button>
  <input type="image" src="..." alt="Or Click Me" asp-controller="Home"
        asp-action="Index">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
  <button formaction="/Home">Click Me</button>
  <input type="image" src="..." alt="Or Click Me" formaction="/Home">
</form>
```

Submit to page example

The following markup submits the form to the `About` Razor Page:

C#HTML

```
<form method="post">
  <button asp-page="About">Click Me</button>
  <input type="image" src="..." alt="Or Click Me" asp-page="About">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
  <button formaction="/About">Click Me</button>
  <input type="image" src="..." alt="Or Click Me" formaction="/About">
</form>
```

Submit to route example

Consider the `/Home/Test` endpoint:

C#

```
public class HomeController : Controller
{
    [Route("/Home/Test", Name = "Custom")]
    public string Test()
    {
        return "This is the test page";
    }
}
```

The following markup submits the form to the `/Home/Test` endpoint.

C#HTML

```
<form method="post">
    <button asp-route="Custom">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-route="Custom">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/Home/Test">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home/Test">
</form>
```

The Input Tag Helper

The Input Tag Helper binds an HTML `<input>` [↗](#) element to a model expression in your razor view.

Syntax:

C#HTML

```
<input asp-for="<Expression Name>">
```

The Input Tag Helper:

- Generates the `id` and `name` HTML attributes for the expression name specified in the `asp-for` attribute. `asp-for="Property1.Property2"` is equivalent to `m => m.Property1.Property2`. The name of the expression is what is used for the `asp-for` attribute value. See the [Expression names](#) section for additional information.
- Sets the HTML `type` attribute value based on the model type and [data annotation](#) attributes applied to the model property
- Won't overwrite the HTML `type` attribute value when one is specified
- Generates [HTML5](#) [↗](#) validation attributes from [data annotation](#) attributes applied to model properties
- Has an HTML Helper feature overlap with `Html.TextBoxFor` and `Html.EditorFor`. See the [HTML Helper alternatives to Input Tag Helper](#) section for details.

- Provides strong typing. If the name of the property changes and you don't update the Tag Helper you'll get an error similar to the following:

```
An error occurred during the compilation of a resource required to
process
this request. Please review the following specific error details and
modify
your source code appropriately.
```

```
Type expected
'RegisterViewModel' does not contain a definition for 'Email' and no
extension method 'Email' accepting a first argument of type
'RegisterViewModel'
could be found (are you missing a using directive or an assembly
reference?)
```

The `Input` Tag Helper sets the HTML `type` attribute based on the .NET type. The following table lists some common .NET types and generated HTML type (not every .NET type is listed).

[Expand table](#)

.NET type	Input Type
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime-local" ↗
Byte	type="number"
Int	type="number"
Single, Double	type="number"

The following table shows some common [data annotations](#) attributes that the input tag helper will map to specific input types (not every validation attribute is listed):

[Expand table](#)

Attribute	Input Type
[EmailAddress]	type="email"
[Url]	type="url"

Attribute	Input Type
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

CSHTML

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <button type="submit">Register</button>
</form>
```

The code above generates the following HTML:

HTML

```
<form method="post" action="/Demo/RegisterInput">
    Email:
    <input type="email" data-val="true"
```



```

        data-val-email="The Email Address field is not a valid email
address."
        data-val-required="The Email Address field is required."
        id="Email" name="Email" value=""><br>
    Password:
    <input type="password" data-val="true"
        data-val-required="The Password field is required."
        id="Password" name="Password"><br>
    <button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>

```

The data annotations applied to the `Email` and `Password` properties generate metadata on the model. The Input Tag Helper consumes the model metadata and produces [HTML5](#) `data-val-*` attributes (see [Model Validation](#)). These attributes describe the validators to attach to the input fields. This provides unobtrusive HTML5 and [jQuery](#) validation. The unobtrusive attributes have the format `data-val-rule="Error Message"`, where rule is the name of the validation rule (such as `data-val-required`, `data-val-email`, `data-val-maxlength`, etc.) If an error message is provided in the attribute, it's displayed as the value for the `data-val-rule` attribute. There are also attributes of the form `data-val-ruleName-argumentName="argumentValue"` that provide additional details about the rule, for example, `data-val-maxlength-max="1024"`.

When binding multiple `input` controls to the same property, the generated controls share the same `id`, which makes the generated mark-up invalid. To prevent duplicates, specify the `id` attribute for each control explicitly.

Checkbox hidden input rendering

Checkboxes in HTML5 don't submit a value when they're unchecked. To enable a default value to be sent for an unchecked checkbox, the Input Tag Helper generates an additional hidden input for checkboxes.

For example, consider the following Razor markup that uses the Input Tag Helper for a boolean model property `IsChecked`:

C#HTML

```

<form method="post">
    <input asp-for="@Model.IsChecked" />
    <button type="submit">Submit</button>
</form>

```

The preceding Razor markup generates HTML markup similar to the following:

HTML

```
<form method="post">
  <input name="IsChecked" type="checkbox" value="true" />
  <button type="submit">Submit</button>

  <input name="IsChecked" type="hidden" value="false" />
</form>
```

The preceding HTML markup shows an additional hidden input with a name of `IsChecked` and a value of `false`. By default, this hidden input is rendered at the end of the form. When the form is submitted:

- If the `IsChecked` checkbox input is checked, both `true` and `false` are submitted as values.
- If the `IsChecked` checkbox input is unchecked, only the hidden input value `false` is submitted.

The ASP.NET Core model-binding process reads only the first value when binding to a `bool` value, which results in `true` for checked checkboxes and `false` for unchecked checkboxes.

To configure the behavior of the hidden input rendering, set the [CheckBoxHiddenInputRenderMode](#) property on [MvcViewOptions.HtmlHelperOptions](#). For example:

C#

```
services.Configure<MvcViewOptions>(options =>
    options.HtmlHelperOptions.CheckBoxHiddenInputRenderMode =
        CheckBoxHiddenInputRenderMode.None);
```

The preceding code disables hidden input rendering for checkboxes by setting `CheckBoxHiddenInputRenderMode` to [CheckBoxHiddenInputRenderMode.None](#). For all available rendering modes, see the [CheckBoxHiddenInputRenderMode](#) enum.

HTML Helper alternatives to Input Tag Helper

`Html.TextBox`, `Html.TextBoxFor`, `Html.Editor` and `Html.EditorFor` have overlapping features with the Input Tag Helper. The Input Tag Helper will automatically set the `type` attribute; `Html.TextBox` and `Html.TextBoxFor` won't. `Html.Editor` and `Html.EditorFor` handle collections, complex objects and templates; the Input Tag Helper doesn't. The

Input Tag Helper, `Html.EditorFor` and `Html.TextBoxFor` are strongly typed (they use lambda expressions); `Html.TextBox` and `Html.Editor` are not (they use expression names).

HtmlAttributes

`@Html.Editor()` and `@Html.EditorFor()` use a special `ViewDataDictionary` entry named `htmlAttributes` when executing their default templates. This behavior is optionally augmented using `additionalViewData` parameters. The key "htmlAttributes" is case-insensitive. The key "htmlAttributes" is handled similarly to the `htmlAttributes` object passed to input helpers like `@Html.TextBox()`.

C#HTML

```
@Html.EditorFor(model => model.YourProperty,
    new { htmlAttributes = new { @class="myCssClass", style="width:100px" } })
```

Expression names

The `asp-for` attribute value is a `ModelExpression` and the right hand side of a lambda expression. Therefore, `asp-for="Property1"` becomes `m => m.Property1` in the generated code which is why you don't need to prefix with `Model`. You can use the "@" character to start an inline expression and move before the `m.`:

C#HTML

```
@{
    var joe = "Joe";
}

<input asp-for="@joe">
```

Generates the following:

HTML

```
<input type="text" id="joe" name="joe" value="Joe">
```

With collection properties, `asp-for="CollectionProperty[23].Member"` generates the same name as `asp-for="CollectionProperty[i].Member"` when `i` has the value `23`.

When ASP.NET Core MVC calculates the value of `ModelExpression`, it inspects several sources, including `ModelState`. Consider `<input type="text" asp-for="Name">`. The calculated `value` attribute is the first non-null value from:

- `ModelState` entry with key "Name".
- Result of the expression `Model.Name`.

Navigating child properties

You can also navigate to child properties using the property path of the view model. Consider a more complex model class that contains a child `Address` property.

C#

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

C#

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

In the view, we bind to `Address.AddressLine1`:

CSHTML

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <label>Address: <input asp-for="Address.AddressLine1" /></label><br />
    <button type="submit">Register</button>
</form>
```

The following HTML is generated for `Address.AddressLine1`:

HTML

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1" value="">
```

Expression names and Collections

Sample, a model containing an array of `Colors`:

C#

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

The action method:

C#

```
public IActionResult Edit(int id, int colorIndex)
{
    ViewData["Index"] = colorIndex;
    return View(GetPerson(id));
}
```

The following Razor shows how you access a specific `color` element:

CSHTML

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

The `Views/Shared/EditorTemplates/String.cshtml` template:

CSHTML

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

Sample using `List<T>`:

```
C#

public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

The following Razor shows how to iterate over a collection:

```
C#HTML

@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

The `Views/Shared/EditorTemplates/ToDoItem.cshtml` template:

```
C#HTML

@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
```

```
</td>
```

```
@*
```

```
    This template replaces the following Razor which evaluates the indexer  
    three times.
```

```
    <td>
```

```
        <label asp-for="@Model[i].Name"></label>
```

```
        @Html.DisplayFor(model => model[i].Name)
```

```
    </td>
```

```
    <td>
```

```
        <input asp-for="@Model[i].IsDone" />
```

```
    </td>
```

```
*@
```

`foreach` should be used if possible when the value is going to be used in an `asp-for` or `Html.DisplayFor` equivalent context. In general, `for` is better than `foreach` (if the scenario allows it) because it doesn't need to allocate an enumerator; however, evaluating an indexer in a LINQ expression can be expensive and should be minimized.

❗ Note

The commented sample code above shows how you would replace the lambda expression with the `@` operator to access each `ToDoItem` in the list.

The Textarea Tag Helper

The `Textarea Tag Helper` tag helper is similar to the `Input Tag Helper`.

- Generates the `id` and `name` attributes, and the data validation attributes from the model for a `<textarea>` [↗](#) element.
- Provides strong typing.
- HTML Helper alternative: `Html.TextAreaFor`

Sample:

```
C#
```

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

```
namespace FormsTagHelper.ViewModels
```

```
{
```

```
    public class DescriptionViewModel
```

```
{
    [MinLength(5)]
    [MaxLength(1024)]
    public string Description { get; set; }
}
```

C#HTML

```
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>
```

The following HTML is generated:

HTML

```
<form method="post" action="/Demo/RegisterTextArea">
    <textarea data-val="true"
        data-val-maxlength="The field Description must be a string or array type
with a maximum length of &#x27;1024&#x27;."
        data-val-maxlength-max="1024"
        data-val-minlength="The field Description must be a string or array type
with a minimum length of &#x27;5&#x27;."
        data-val-minlength-min="5"
        id="Description" name="Description">
    </textarea>
    <button type="submit">Test</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>
```

The Label Tag Helper

- Generates the label caption and `for` attribute on a `<label>` [↗](#) element for an expression name
- HTML Helper alternative: `Html.LabelFor`.

The `Label Tag Helper` provides the following benefits over a pure HTML label element:

- You automatically get the descriptive label value from the `Display` attribute. The intended display name might change over time, and the combination of `Display` attribute and Label Tag Helper will apply the `Display` everywhere it's used.

- Less markup in source code
- Strong typing with the model property.

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

CSHTML

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

The following HTML is generated for the `<label>` element:

HTML

```
<label for="Email">Email Address</label>
```

The Label Tag Helper generated the `for` attribute value of "Email", which is the ID associated with the `<input>` element. The Tag Helpers generate consistent `id` and `for` elements so they can be correctly associated. The caption in this sample comes from the `Display` attribute. If the model didn't contain a `Display` attribute, the caption would be the property name of the expression. To override the default caption, add a caption inside the label tag.

The Validation Tag Helpers

There are two Validation Tag Helpers. The `Validation Message Tag Helper` (which displays a validation message for a single property on your model), and the `Validation Summary Tag Helper` (which displays a summary of validation errors). The `Input Tag Helper` adds HTML5 client side validation attributes to input elements based on data annotation attributes on your model classes. Validation is also performed on the server. The Validation Tag Helper displays these error messages when a validation error occurs.

The Validation Message Tag Helper

- Adds the [HTML5](#) `data-valmsg-for="property"` attribute to the [span](#) element, which attaches the validation error messages on the input field of the specified model property. When a client side validation error occurs, [jQuery](#) displays the error message in the `` element.
- Validation also takes place on the server. Clients may have JavaScript disabled and some validation can only be done on the server side.
- HTML Helper alternative: `Html.ValidationMessageFor`

The `Validation Message Tag Helper` is used with the `asp-validation-for` attribute on an HTML [span](#) element.

C#HTML

```
<span asp-validation-for="Email"></span>
```

The Validation Message Tag Helper will generate the following HTML:

HTML

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

You generally use the `Validation Message Tag Helper` after an `Input Tag Helper` for the same property. Doing so displays any validation error messages near the input that caused the error.

❗ Note

You must have a view with the correct JavaScript and [jQuery](#) script references in place for client side validation. See [Model Validation](#) for more information.

When a server side validation error occurs (for example when you have custom server side validation or client-side validation is disabled), MVC places that error message as the body of the `` element.

HTML

```
<span class="field-validation-error" data-valmsg-for="Email"
      data-valmsg-replace="true">
    The Email Address field is required.
</span>
```

The Validation Summary Tag Helper

- Targets `<div>` elements with the `asp-validation-summary` attribute
- HTML Helper alternative: `@Html.ValidationSummary`

The `Validation Summary Tag Helper` is used to display a summary of validation messages. The `asp-validation-summary` attribute value can be any of the following:

[Expand table](#)

<code>asp-validation-summary</code>	Validation messages displayed
<code>All</code>	Property and model level
<code>ModelOnly</code>	Model
<code>None</code>	None

Sample

In the following example, the data model has `DataAnnotation` attributes, which generates validation error messages on the `<input>` element. When a validation error occurs, the Validation Tag Helper displays the error message:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
```

```

    [EmailAddress]
    [Display(Name = "Email Address")]
    public string Email { get; set; }

    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }
}

```

CSHTML

```

@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    <label>Email: <input asp-for="Email" /></label> <br />
    <span asp-validation-for="Email"></span><br />
    <label>Password: <input asp-for="Password" /></label><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>

```

The generated HTML (when the model is valid):

HTML

```

<form action="/DemoReg/Register" method="post">
    <label>Email: <input name="Email" id="Email" type="email" value=""
        data-val-required="The Email field is required."
        data-val-email="The Email field is not a valid email address."
        data-val="true"></label><br>
    <span class="field-validation-valid" data-valmsg-replace="true"
        data-valmsg-for="Email"></span><br>
    <label>Password: <input name="Password" id="Password" type="password"
        data-val-required="The Password field is required." data-val="true">
    </label><br>
    <span class="field-validation-valid" data-valmsg-replace="true"
        data-valmsg-for="Password"></span><br>
    <button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for
    brevity">">
</form>

```

The Select Tag Helper

- Generates [select](#) and associated [option](#) elements for properties of your model.
- Has an HTML Helper alternative `Html.DropDownListFor` and `Html.ListBoxFor`

The `Select Tag Helper` `asp-for` specifies the model property name for the [select](#) element and `asp-items` specifies the [option](#) elements. For example:

C#HTML

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Sample:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new
List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA" },
        };
    }
}
```

The `Index` method initializes the `CountryViewModel`, sets the selected country and passes it to the `Index` view.

C#

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

The HTTP POST `Index` method displays the selection:

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
```

```

public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country + " selected";
        return RedirectToAction("IndexSuccess", new { message = msg });
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}

```

The `Index` view:

C#HTML

```

@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>

```

Which generates the following HTML (with "CA" selected):

HTML

```

<form method="post" action="/">
    <select id="Country" name="Country">
        <option value="MX">Mexico</option>
        <option selected="selected" value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>

```

⚠ Note

We don't recommend using `ViewBag` or `ViewData` with the Select Tag Helper. A view model is more robust at providing MVC metadata and generally less problematic.

The `asp-for` attribute value is a special case and doesn't require a `Model` prefix, the other Tag Helper attributes do (such as `asp-items`)

C#HTML

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Enum binding

It's often convenient to use `<select>` with an `enum` property and generate the `SelectListItem` elements from the `enum` values.

Sample:

C#

```
public class CountryEnumViewModel
{
    public CountryEnum EnumCountry { get; set; }
}
```

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The `GetEnumSelectList` method generates a `SelectList` object for an enum.

C#HTML

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
        asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
```

```
<br /><button type="submit">Register</button>
</form>
```

You can mark your enumerator list with the `Display` attribute to get a richer UI:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The following HTML is generated:

HTML

```
<form method="post" action="/Home/IndexEnum">
    <select data-val="true" data-val-required="The EnumCountry field is
    required."
        id="EnumCountry" name="EnumCountry">
        <option value="0">United Mexican States</option>
        <option value="1">United States of America</option>
        <option value="2">Canada</option>
        <option value="3">France</option>
        <option value="4">Germany</option>
        <option selected="selected" value="5">Spain</option>
    </select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
    for brevity>">
</form>
```

Option Group

The HTML `<optgroup>` [↗](#) element is generated when the view model contains one or more `SelectListGroup` objects.

The `CountryViewModelGroup` groups the `SelectListItem` elements into the "North America" and "Europe" groups:

C#

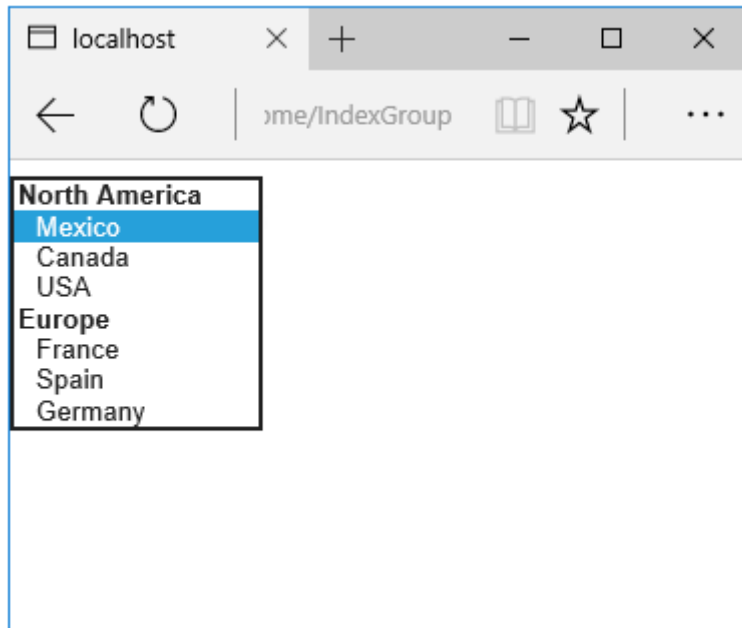
```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America" };
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
        };
    }
}
```

```
public string Country { get; set; }

public List<SelectListItem> Countries { get; }
```

The two groups are shown below:



The generated HTML:

HTML

```
<form method="post" action="/Home/IndexGroup">
  <select id="Country" name="Country">
    <optgroup label="North America">
      <option value="MEX">Mexico</option>
      <option value="CAN">Canada</option>
      <option value="US">USA</option>
    </optgroup>
    <optgroup label="Europe">
      <option value="FR">France</option>
      <option value="ES">Spain</option>
      <option value="DE">Germany</option>
    </optgroup>
  </select>
  <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

Multiple select

The Select Tag Helper will automatically generate the `multiple = "multiple"` attribute if the property specified in the `asp-for` attribute is an `IEnumerable`. For example, given the following model:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new
        List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA" },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain" },
            new SelectListItem { Value = "DE", Text = "Germany" }
        };
    }
}
```

With the following view:

C#HTML

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Generates the following HTML:

HTML

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
        multiple="multiple"
        name="CountryCodes"><option value="MX">Mexico</option>
    <option value="CA">Canada</option>
    <option value="US">USA</option>
    <option value="FR">France</option>
```

```

<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>

```

No selection

If you find yourself using the "not specified" option in multiple pages, you can create a template to eliminate repeating the HTML:

CSSHTML

```

@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>

```

The `Views/Shared/EditorTemplates/CountryViewModel.csshtml` template:

CSSHTML

```

@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>

```

Adding HTML `<option>` [↗](#) elements isn't limited to the *No selection* case. For example, the following view and action method will generate HTML similar to the code above:

C#

```

public IActionResult IndexNone()
{
    var model = new CountryViewModel();
    model.Countries.Insert(0, new SelectListItem("<none>", ""));
    return View(model);
}

```

CSSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

The correct `<option>` element will be selected (contain the `selected="selected"` attribute) depending on the current `Country` value.

C#

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

HTML

```
<form method="post" action="/Home/IndexEmpty">
    <select id="Country" name="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA" selected="selected">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

Additional resources

- [Tag Helpers in ASP.NET Core](#)
- [HTML Form element](#) ↗
- [Request Verification Token](#)
- [Model Binding in ASP.NET Core](#)
- [Model validation in ASP.NET Core MVC](#)
- [IAttributeAdapter Interface](#)

- [Code snippets for this document](#) ↗

Image Tag Helper in ASP.NET Core

Article • 06/03/2022

By [Peter Kellner](#) 

The Image Tag Helper enhances the `` tag to provide cache-busting behavior for static image files.

A cache-busting string is a unique value representing the hash of the static image file appended to the asset's URL. The unique string prompts clients (and some proxies) to reload the image from the host web server and not from the client's cache.

If the image source (`src`) is a static file on the host web server:

- A unique cache-busting string is appended as a query parameter to the image source.
- If the file on the host web server changes, a unique request URL is generated that includes the updated request parameter.

For an overview of Tag Helpers, see [Tag Helpers in ASP.NET Core](#).

Image Tag Helper Attributes

`src`

To activate the Image Tag Helper, the `src` attribute is required on the `` element.

The image source (`src`) must point to a physical static file on the server. If the `src` is a remote URI, the cache-busting query string parameter isn't generated.

`asp-append-version`

When `asp-append-version` is specified with a `true` value along with a `src` attribute, the Image Tag Helper is invoked.

The following example uses an Image Tag Helper:

CSHTML

```

```

If the static file exists in the directory `/wwwroot/images/`, the generated HTML is similar to the following (the hash will be different):

HTML

```

```

The value assigned to the parameter `v` is the hash value of the `asplogo.png` file on disk. If the web server is unable to obtain read access to the static file, no `v` parameter is added to the `src` attribute in the rendered markup.

For a Tag Helper to generate a version for a static file outside `wwwroot`, see [Serve files from multiple locations](#)

Hash caching behavior

The Image Tag Helper uses the cache provider on the local web server to store the calculated `Sha512` hash of a given file. If the file is requested multiple times, the hash isn't recalculated. The cache is invalidated by a file watcher that's attached to the file when the file's `Sha512` hash is calculated. When the file changes on disk, a new hash is calculated and cached.

Additional resources

- [Cache in-memory in ASP.NET Core](#)

Tag Helpers in forms in ASP.NET Core

Article • 09/27/2024

By [Rick Anderson](#), [N. Taylor Mullen](#), [Dave Paquette](#), and [Jerrie Pelser](#)

This document demonstrates working with Forms and the HTML elements commonly used on a Form. The HTML [Form](#) element provides the primary mechanism web apps use to post back data to the server. Most of this document describes [Tag Helpers](#) and how they can help you productively create robust HTML forms. We recommend you read [Introduction to Tag Helpers](#) before you read this document.

In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers don't replace HTML Helpers and there's not a Tag Helper for each HTML Helper. When an HTML Helper alternative exists, it's mentioned.

The Form Tag Helper

The [Form Tag Helper](#):

- Generates the HTML [<FORM>](#) `action` attribute value for a MVC controller action or named route
- Generates a hidden [Request Verification Token](#) to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method)
- Provides the `asp-route-<Parameter Name>` attribute, where `<Parameter Name>` is added to the route values. The `routeValues` parameters to `Html.BeginForm` and `Html.BeginRouteForm` provide similar functionality.
- Has an HTML Helper alternative `Html.BeginForm` and `Html.BeginRouteForm`

Sample:

C#HTML

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

The Form Tag Helper above generates the following HTML:

HTML

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

The MVC runtime generates the `action` attribute value from the Form Tag Helper attributes `asp-controller` and `asp-action`. The Form Tag Helper also generates a hidden [Request Verification Token](#) to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method). Protecting a pure HTML Form from cross-site request forgery is difficult, the Form Tag Helper provides this service for you.

Using a named route

The `asp-route` Tag Helper attribute can also generate markup for the HTML `action` attribute. An app with a [route](#) named `register` could use the following markup for the registration page:

C#HTML

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Many of the views in the *Views/Account* folder (generated when you create a new web app with *Individual User Accounts*) contain the `asp-route-returnurl` attribute:

C#HTML

```
<form asp-controller="Account" asp-action="Login"
      asp-route-returnurl="@ViewData["ReturnUrl"]"
      method="post" class="form-horizontal" role="form">
```

ⓘ Note

With the built in templates, `returnUrl` is only populated automatically when you try to access an authorized resource but are not authenticated or authorized. When you attempt an unauthorized access, the security middleware redirects you to the login page with the `returnUrl` set.

The Form Action Tag Helper

The Form Action Tag Helper generates the `formaction` attribute on the generated `<button ...>` or `<input type="image" ...>` tag. The `formaction` attribute controls where a form submits its data. It binds to `<input>` [elements](#) of type `image` and `<button>` [elements](#). The Form Action Tag Helper enables the usage of several [AnchorTagHelper](#) `asp-` attributes to control what `formaction` link is generated for the corresponding element.

Supported [AnchorTagHelper](#) attributes to control the value of `formaction`:

[Expand table](#)

Attribute	Description
asp-controller	The name of the controller.
asp-action	The name of the action method.
asp-area	The name of the area.
asp-page	The name of the Razor page.
asp-page-handler	The name of the Razor page handler.
asp-route	The name of the route.
asp-route-{value}	A single URL route value. For example, <code>asp-route-id="1234"</code> .
asp-all-route-data	All route values.
asp-fragment	The URL fragment.

Submit to controller example

The following markup submits the form to the `Index` action of `HomeController` when the input or button are selected:

C#HTML

```
<form method="post">
  <button asp-controller="Home" asp-action="Index">Click Me</button>
  <input type="image" src="..." alt="Or Click Me" asp-controller="Home"
        asp-action="Index">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
  <button formaction="/Home">Click Me</button>
  <input type="image" src="..." alt="Or Click Me" formaction="/Home">
</form>
```

Submit to page example

The following markup submits the form to the `About` Razor Page:

C#HTML

```
<form method="post">
  <button asp-page="About">Click Me</button>
  <input type="image" src="..." alt="Or Click Me" asp-page="About">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
  <button formaction="/About">Click Me</button>
  <input type="image" src="..." alt="Or Click Me" formaction="/About">
</form>
```

Submit to route example

Consider the `/Home/Test` endpoint:

C#

```
public class HomeController : Controller
{
    [Route("/Home/Test", Name = "Custom")]
    public string Test()
    {
        return "This is the test page";
    }
}
```

The following markup submits the form to the `/Home/Test` endpoint.

C#HTML

```
<form method="post">
  <button asp-route="Custom">Click Me</button>
  <input type="image" src="..." alt="Or Click Me" asp-route="Custom">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
  <button formaction="/Home/Test">Click Me</button>
  <input type="image" src="..." alt="Or Click Me" formaction="/Home/Test">
</form>
```

The Input Tag Helper

The Input Tag Helper binds an HTML `<input>` [↗](#) element to a model expression in your razor view.

Syntax:

C#HTML

```
<input asp-for="<Expression Name>">
```

The Input Tag Helper:

- Generates the `id` and `name` HTML attributes for the expression name specified in the `asp-for` attribute. `asp-for="Property1.Property2"` is equivalent to `m => m.Property1.Property2`. The name of the expression is what is used for the `asp-for` attribute value. See the [Expression names](#) section for additional information.
- Sets the HTML `type` attribute value based on the model type and [data annotation](#) attributes applied to the model property
- Won't overwrite the HTML `type` attribute value when one is specified
- Generates [HTML5](#) [↗](#) validation attributes from [data annotation](#) attributes applied to model properties
- Has an HTML Helper feature overlap with `Html.TextBoxFor` and `Html.EditorFor`. See the [HTML Helper alternatives to Input Tag Helper](#) section for details.

- Provides strong typing. If the name of the property changes and you don't update the Tag Helper you'll get an error similar to the following:

```
An error occurred during the compilation of a resource required to
process
this request. Please review the following specific error details and
modify
your source code appropriately.
```

```
Type expected
'RegisterViewModel' does not contain a definition for 'Email' and no
extension method 'Email' accepting a first argument of type
'RegisterViewModel'
could be found (are you missing a using directive or an assembly
reference?)
```

The `Input` Tag Helper sets the HTML `type` attribute based on the .NET type. The following table lists some common .NET types and generated HTML type (not every .NET type is listed).

[Expand table](#)

.NET type	Input Type
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime-local" ↗
Byte	type="number"
Int	type="number"
Single, Double	type="number"

The following table shows some common [data annotations](#) attributes that the input tag helper will map to specific input types (not every validation attribute is listed):

[Expand table](#)

Attribute	Input Type
[EmailAddress]	type="email"
[Url]	type="url"

Attribute	Input Type
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

CSHTML

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <button type="submit">Register</button>
</form>
```

The code above generates the following HTML:

HTML

```
<form method="post" action="/Demo/RegisterInput">
    Email:
    <input type="email" data-val="true"
```

```

        data-val-email="The Email Address field is not a valid email
address."
        data-val-required="The Email Address field is required."
        id="Email" name="Email" value=""><br>
    Password:
    <input type="password" data-val="true"
        data-val-required="The Password field is required."
        id="Password" name="Password"><br>
    <button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>

```

The data annotations applied to the `Email` and `Password` properties generate metadata on the model. The Input Tag Helper consumes the model metadata and produces [HTML5](#) `data-val-*` attributes (see [Model Validation](#)). These attributes describe the validators to attach to the input fields. This provides unobtrusive HTML5 and [jQuery](#) validation. The unobtrusive attributes have the format `data-val-rule="Error Message"`, where rule is the name of the validation rule (such as `data-val-required`, `data-val-email`, `data-val-maxlength`, etc.) If an error message is provided in the attribute, it's displayed as the value for the `data-val-rule` attribute. There are also attributes of the form `data-val-ruleName-argumentName="argumentValue"` that provide additional details about the rule, for example, `data-val-maxlength-max="1024"`.

When binding multiple `input` controls to the same property, the generated controls share the same `id`, which makes the generated mark-up invalid. To prevent duplicates, specify the `id` attribute for each control explicitly.

Checkbox hidden input rendering

Checkboxes in HTML5 don't submit a value when they're unchecked. To enable a default value to be sent for an unchecked checkbox, the Input Tag Helper generates an additional hidden input for checkboxes.

For example, consider the following Razor markup that uses the Input Tag Helper for a boolean model property `IsChecked`:

C#HTML

```

<form method="post">
    <input asp-for="@Model.IsChecked" />
    <button type="submit">Submit</button>
</form>

```


The preceding Razor markup generates HTML markup similar to the following:

HTML

```
<form method="post">
  <input name="IsChecked" type="checkbox" value="true" />
  <button type="submit">Submit</button>

  <input name="IsChecked" type="hidden" value="false" />
</form>
```

The preceding HTML markup shows an additional hidden input with a name of `IsChecked` and a value of `false`. By default, this hidden input is rendered at the end of the form. When the form is submitted:

- If the `IsChecked` checkbox input is checked, both `true` and `false` are submitted as values.
- If the `IsChecked` checkbox input is unchecked, only the hidden input value `false` is submitted.

The ASP.NET Core model-binding process reads only the first value when binding to a `bool` value, which results in `true` for checked checkboxes and `false` for unchecked checkboxes.

To configure the behavior of the hidden input rendering, set the [CheckBoxHiddenInputRenderMode](#) property on [MvcViewOptions.HtmlHelperOptions](#). For example:

C#

```
services.Configure<MvcViewOptions>(options =>
    options.HtmlHelperOptions.CheckBoxHiddenInputRenderMode =
        CheckBoxHiddenInputRenderMode.None);
```

The preceding code disables hidden input rendering for checkboxes by setting `CheckBoxHiddenInputRenderMode` to [CheckBoxHiddenInputRenderMode.None](#). For all available rendering modes, see the [CheckBoxHiddenInputRenderMode](#) enum.

HTML Helper alternatives to Input Tag Helper

`Html.TextBox`, `Html.TextBoxFor`, `Html.Editor` and `Html.EditorFor` have overlapping features with the Input Tag Helper. The Input Tag Helper will automatically set the `type` attribute; `Html.TextBox` and `Html.TextBoxFor` won't. `Html.Editor` and `Html.EditorFor` handle collections, complex objects and templates; the Input Tag Helper doesn't. The

Input Tag Helper, `Html.EditorFor` and `Html.TextBoxFor` are strongly typed (they use lambda expressions); `Html.TextBox` and `Html.Editor` are not (they use expression names).

HtmlAttributes

`@Html.Editor()` and `@Html.EditorFor()` use a special `ViewDataDictionary` entry named `htmlAttributes` when executing their default templates. This behavior is optionally augmented using `additionalViewData` parameters. The key "htmlAttributes" is case-insensitive. The key "htmlAttributes" is handled similarly to the `htmlAttributes` object passed to input helpers like `@Html.TextBox()`.

C#HTML

```
@Html.EditorFor(model => model.YourProperty,  
    new { htmlAttributes = new { @class="myCssClass", style="width:100px" } })
```

Expression names

The `asp-for` attribute value is a `ModelExpression` and the right hand side of a lambda expression. Therefore, `asp-for="Property1"` becomes `m => m.Property1` in the generated code which is why you don't need to prefix with `Model`. You can use the "@" character to start an inline expression and move before the `m.`:

C#HTML

```
@{  
    var joe = "Joe";  
}  
  
<input asp-for="@joe">
```

Generates the following:

HTML

```
<input type="text" id="joe" name="joe" value="Joe">
```

With collection properties, `asp-for="CollectionProperty[23].Member"` generates the same name as `asp-for="CollectionProperty[i].Member"` when `i` has the value `23`.

When ASP.NET Core MVC calculates the value of `ModelExpression`, it inspects several sources, including `ModelState`. Consider `<input type="text" asp-for="Name">`. The calculated `value` attribute is the first non-null value from:

- `ModelState` entry with key "Name".
- Result of the expression `Model.Name`.

Navigating child properties

You can also navigate to child properties using the property path of the view model. Consider a more complex model class that contains a child `Address` property.

C#

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

C#

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

In the view, we bind to `Address.AddressLine1`:

CSHTML

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <label>Address: <input asp-for="Address.AddressLine1" /></label><br />
    <button type="submit">Register</button>
</form>
```

The following HTML is generated for `Address.AddressLine1`:

HTML

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1" value="">
```

Expression names and Collections

Sample, a model containing an array of `Colors`:

C#

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

The action method:

C#

```
public IActionResult Edit(int id, int colorIndex)
{
    ViewData["Index"] = colorIndex;
    return View(GetPerson(id));
}
```

The following Razor shows how you access a specific `Color` element:

CSHTML

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

The `Views/Shared/EditorTemplates/String.cshtml` template:

CSHTML

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

Sample using `List<T>`:

```
C#

public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

The following Razor shows how to iterate over a collection:

```
C#HTML

@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

The `Views/Shared/EditorTemplates/ToDoItem.cshtml` template:

```
C#HTML

@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
```

```
</td>
```

```
@*
```

```
    This template replaces the following Razor which evaluates the indexer  
    three times.
```

```
    <td>
```

```
        <label asp-for="@Model[i].Name"></label>
```

```
        @Html.DisplayFor(model => model[i].Name)
```

```
    </td>
```

```
    <td>
```

```
        <input asp-for="@Model[i].IsDone" />
```

```
    </td>
```

```
*@
```

`foreach` should be used if possible when the value is going to be used in an `asp-for` or `Html.DisplayFor` equivalent context. In general, `for` is better than `foreach` (if the scenario allows it) because it doesn't need to allocate an enumerator; however, evaluating an indexer in a LINQ expression can be expensive and should be minimized.

❗ Note

The commented sample code above shows how you would replace the lambda expression with the `@` operator to access each `ToDoItem` in the list.

The Textarea Tag Helper

The `Textarea Tag Helper` tag helper is similar to the `Input Tag Helper`.

- Generates the `id` and `name` attributes, and the data validation attributes from the model for a `<textarea>` [↗](#) element.
- Provides strong typing.
- HTML Helper alternative: `Html.TextAreaFor`

Sample:

```
C#
```

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

```
namespace FormsTagHelper.ViewModels
```

```
{
```

```
    public class DescriptionViewModel
```

```

{
    [MinLength(5)]
    [MaxLength(1024)]
    public string Description { get; set; }
}

```

C#HTML

```

@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>

```

The following HTML is generated:

HTML

```

<form method="post" action="/Demo/RegisterTextArea">
    <textarea data-val="true"
        data-val-maxlength="The field Description must be a string or array type
with a maximum length of &#x27;1024&#x27;."
        data-val-maxlength-max="1024"
        data-val-minlength="The field Description must be a string or array type
with a minimum length of &#x27;5&#x27;."
        data-val-minlength-min="5"
        id="Description" name="Description">
    </textarea>
    <button type="submit">Test</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>

```

The Label Tag Helper

- Generates the label caption and `for` attribute on a `<label>` [↗](#) element for an expression name
- HTML Helper alternative: `Html.LabelFor`.

The `Label Tag Helper` provides the following benefits over a pure HTML label element:

- You automatically get the descriptive label value from the `Display` attribute. The intended display name might change over time, and the combination of `Display` attribute and Label Tag Helper will apply the `Display` everywhere it's used.

- Less markup in source code
- Strong typing with the model property.

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

CSHTML

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

The following HTML is generated for the `<label>` element:

HTML

```
<label for="Email">Email Address</label>
```

The Label Tag Helper generated the `for` attribute value of "Email", which is the ID associated with the `<input>` element. The Tag Helpers generate consistent `id` and `for` elements so they can be correctly associated. The caption in this sample comes from the `Display` attribute. If the model didn't contain a `Display` attribute, the caption would be the property name of the expression. To override the default caption, add a caption inside the label tag.

The Validation Tag Helpers

There are two Validation Tag Helpers. The `Validation Message Tag Helper` (which displays a validation message for a single property on your model), and the `Validation Summary Tag Helper` (which displays a summary of validation errors). The `Input Tag Helper` adds HTML5 client side validation attributes to input elements based on data annotation attributes on your model classes. Validation is also performed on the server. The Validation Tag Helper displays these error messages when a validation error occurs.

The Validation Message Tag Helper

- Adds the [HTML5](#) `data-valmsg-for="property"` attribute to the [span](#) element, which attaches the validation error messages on the input field of the specified model property. When a client side validation error occurs, [jQuery](#) displays the error message in the `` element.
- Validation also takes place on the server. Clients may have JavaScript disabled and some validation can only be done on the server side.
- HTML Helper alternative: `Html.ValidationMessageFor`

The `Validation Message Tag Helper` is used with the `asp-validation-for` attribute on an HTML [span](#) element.

C#HTML

```
<span asp-validation-for="Email"></span>
```

The Validation Message Tag Helper will generate the following HTML:

HTML

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

You generally use the `Validation Message Tag Helper` after an `Input Tag Helper` for the same property. Doing so displays any validation error messages near the input that caused the error.

❗ Note

You must have a view with the correct JavaScript and [jQuery](#) script references in place for client side validation. See [Model Validation](#) for more information.

When a server side validation error occurs (for example when you have custom server side validation or client-side validation is disabled), MVC places that error message as the body of the `` element.

HTML

```
<span class="field-validation-error" data-valmsg-for="Email"
      data-valmsg-replace="true">
    The Email Address field is required.
</span>
```

The Validation Summary Tag Helper

- Targets `<div>` elements with the `asp-validation-summary` attribute
- HTML Helper alternative: `@Html.ValidationSummary`

The `Validation Summary Tag Helper` is used to display a summary of validation messages. The `asp-validation-summary` attribute value can be any of the following:

[Expand table](#)

<code>asp-validation-summary</code>	Validation messages displayed
<code>All</code>	Property and model level
<code>ModelOnly</code>	Model
<code>None</code>	None

Sample

In the following example, the data model has `DataAnnotation` attributes, which generates validation error messages on the `<input>` element. When a validation error occurs, the Validation Tag Helper displays the error message:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
```

```

    [EmailAddress]
    [Display(Name = "Email Address")]
    public string Email { get; set; }

    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }
}

```

CSHTML

```

@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    <label>Email: <input asp-for="Email" /></label> <br />
    <span asp-validation-for="Email"></span><br />
    <label>Password: <input asp-for="Password" /></label><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>

```

The generated HTML (when the model is valid):

HTML

```

<form action="/DemoReg/Register" method="post">
    <label>Email: <input name="Email" id="Email" type="email" value=""
        data-val-required="The Email field is required."
        data-val-email="The Email field is not a valid email address."
        data-val="true"></label><br>
    <span class="field-validation-valid" data-valmsg-replace="true"
        data-valmsg-for="Email"></span><br>
    <label>Password: <input name="Password" id="Password" type="password"
        data-val-required="The Password field is required." data-val="true">
    </label><br>
    <span class="field-validation-valid" data-valmsg-replace="true"
        data-valmsg-for="Password"></span><br>
    <button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for
    brevity>">
</form>

```

The Select Tag Helper

- Generates [select](#) and associated [option](#) elements for properties of your model.
- Has an HTML Helper alternative `Html.DropDownListFor` and `Html.ListBoxFor`

The `Select Tag Helper` `asp-for` specifies the model property name for the [select](#) element and `asp-items` specifies the [option](#) elements. For example:

C#HTML

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Sample:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new
List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA" },
        };
    }
}
```

The `Index` method initializes the `CountryViewModel`, sets the selected country and passes it to the `Index` view.

C#

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

The HTTP POST `Index` method displays the selection:

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
```

```
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country + " selected";
        return RedirectToAction("IndexSuccess", new { message = msg });
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

The `Index` view:

C#HTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Which generates the following HTML (with "CA" selected):

HTML

```
<form method="post" action="/">
    <select id="Country" name="Country">
        <option value="MX">Mexico</option>
        <option selected="selected" value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

⚠ Note

We don't recommend using `ViewBag` or `ViewData` with the Select Tag Helper. A view model is more robust at providing MVC metadata and generally less problematic.

The `asp-for` attribute value is a special case and doesn't require a `Model` prefix, the other Tag Helper attributes do (such as `asp-items`)

C#HTML

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Enum binding

It's often convenient to use `<select>` with an `enum` property and generate the `SelectListItem` elements from the `enum` values.

Sample:

C#

```
public class CountryEnumViewModel
{
    public CountryEnum EnumCountry { get; set; }
}
```

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The `GetEnumSelectList` method generates a `SelectList` object for an enum.

C#HTML

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
        asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
```

```
<br /><button type="submit">Register</button>
</form>
```

You can mark your enumerator list with the `Display` attribute to get a richer UI:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The following HTML is generated:

HTML

```
<form method="post" action="/Home/IndexEnum">
    <select data-val="true" data-val-required="The EnumCountry field is
    required."
        id="EnumCountry" name="EnumCountry">
        <option value="0">United Mexican States</option>
        <option value="1">United States of America</option>
        <option value="2">Canada</option>
        <option value="3">France</option>
        <option value="4">Germany</option>
        <option selected="selected" value="5">Spain</option>
    </select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
    for brevity>">
</form>
```

Option Group

The HTML `<optgroup>` [↗](#) element is generated when the view model contains one or more `SelectListGroup` objects.

The `CountryViewModelGroup` groups the `SelectListItem` elements into the "North America" and "Europe" groups:

C#

```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America" };
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

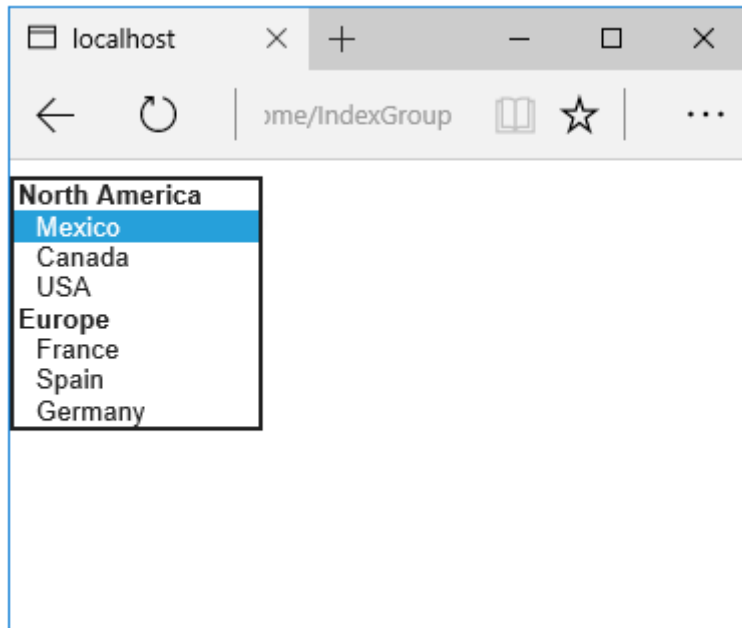
        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
        };
    }
}
```



```
public string Country { get; set; }

public List<SelectListItem> Countries { get; }
```

The two groups are shown below:



The generated HTML:

HTML

```
<form method="post" action="/Home/IndexGroup">
  <select id="Country" name="Country">
    <optgroup label="North America">
      <option value="MEX">Mexico</option>
      <option value="CAN">Canada</option>
      <option value="US">USA</option>
    </optgroup>
    <optgroup label="Europe">
      <option value="FR">France</option>
      <option value="ES">Spain</option>
      <option value="DE">Germany</option>
    </optgroup>
  </select>
  <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

Multiple select

The Select Tag Helper will automatically generate the `multiple = "multiple"` attribute if the property specified in the `asp-for` attribute is an `IEnumerable`. For example, given the following model:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new
List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA" },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain" },
            new SelectListItem { Value = "DE", Text = "Germany" }
        };
    }
}
```

With the following view:

C#HTML

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Generates the following HTML:

HTML

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
        multiple="multiple"
        name="CountryCodes"><option value="MX">Mexico</option>
    <option value="CA">Canada</option>
    <option value="US">USA</option>
    <option value="FR">France</option>
```

```

<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>

```

No selection

If you find yourself using the "not specified" option in multiple pages, you can create a template to eliminate repeating the HTML:

CSSHTML

```

@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>

```

The `Views/Shared/EditorTemplates/CountryViewModel.csshtml` template:

CSSHTML

```

@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>

```

Adding HTML `<option>` [↗](#) elements isn't limited to the *No selection* case. For example, the following view and action method will generate HTML similar to the code above:

C#

```

public IActionResult IndexNone()
{
    var model = new CountryViewModel();
    model.Countries.Insert(0, new SelectListItem("<none>", ""));
    return View(model);
}

```

CSSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

The correct `<option>` element will be selected (contain the `selected="selected"` attribute) depending on the current `Country` value.

C#

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

HTML

```
<form method="post" action="/Home/IndexEmpty">
    <select id="Country" name="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA" selected="selected">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

Additional resources

- [Tag Helpers in ASP.NET Core](#)
- [HTML Form element](#) ↗
- [Request Verification Token](#)
- [Model Binding in ASP.NET Core](#)
- [Model validation in ASP.NET Core MVC](#)
- [IAttributeAdapter Interface](#)

- [Code snippets for this document](#) ↗

Tag Helpers in forms in ASP.NET Core

Article • 09/27/2024

By [Rick Anderson](#), [N. Taylor Mullen](#), [Dave Paquette](#), and [Jerrie Pelser](#)

This document demonstrates working with Forms and the HTML elements commonly used on a Form. The HTML [Form](#) element provides the primary mechanism web apps use to post back data to the server. Most of this document describes [Tag Helpers](#) and how they can help you productively create robust HTML forms. We recommend you read [Introduction to Tag Helpers](#) before you read this document.

In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers don't replace HTML Helpers and there's not a Tag Helper for each HTML Helper. When an HTML Helper alternative exists, it's mentioned.

The Form Tag Helper

The [Form Tag Helper](#):

- Generates the HTML [<FORM>](#) `action` attribute value for a MVC controller action or named route
- Generates a hidden [Request Verification Token](#) to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method)
- Provides the `asp-route-<Parameter Name>` attribute, where `<Parameter Name>` is added to the route values. The `routeValues` parameters to `Html.BeginForm` and `Html.BeginRouteForm` provide similar functionality.
- Has an HTML Helper alternative `Html.BeginForm` and `Html.BeginRouteForm`

Sample:

C#HTML

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

The Form Tag Helper above generates the following HTML:

HTML

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

The MVC runtime generates the `action` attribute value from the Form Tag Helper attributes `asp-controller` and `asp-action`. The Form Tag Helper also generates a hidden [Request Verification Token](#) to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method). Protecting a pure HTML Form from cross-site request forgery is difficult, the Form Tag Helper provides this service for you.

Using a named route

The `asp-route` Tag Helper attribute can also generate markup for the HTML `action` attribute. An app with a [route](#) named `register` could use the following markup for the registration page:

C#HTML

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Many of the views in the *Views/Account* folder (generated when you create a new web app with *Individual User Accounts*) contain the `asp-route-returnurl` attribute:

C#HTML

```
<form asp-controller="Account" asp-action="Login"
      asp-route-returnurl="@ViewData["ReturnUrl"]"
      method="post" class="form-horizontal" role="form">
```

ⓘ Note

With the built in templates, `returnUrl` is only populated automatically when you try to access an authorized resource but are not authenticated or authorized. When you attempt an unauthorized access, the security middleware redirects you to the login page with the `returnUrl` set.

The Form Action Tag Helper

The Form Action Tag Helper generates the `formaction` attribute on the generated `<button ...>` or `<input type="image" ...>` tag. The `formaction` attribute controls where a form submits its data. It binds to `<input>` [↗](#) elements of type `image` and `<button>` [↗](#) elements. The Form Action Tag Helper enables the usage of several [AnchorTagHelper](#) `asp-` attributes to control what `formaction` link is generated for the corresponding element.

Supported [AnchorTagHelper](#) attributes to control the value of `formaction`:

[↗](#) Expand table

Attribute	Description
asp-controller	The name of the controller.
asp-action	The name of the action method.
asp-area	The name of the area.
asp-page	The name of the Razor page.
asp-page-handler	The name of the Razor page handler.
asp-route	The name of the route.
asp-route-{value}	A single URL route value. For example, <code>asp-route-id="1234"</code> .
asp-all-route-data	All route values.
asp-fragment	The URL fragment.

Submit to controller example

The following markup submits the form to the `Index` action of `HomeController` when the input or button are selected:

C#HTML

```
<form method="post">
  <button asp-controller="Home" asp-action="Index">Click Me</button>
  <input type="image" src="..." alt="Or Click Me" asp-controller="Home"
        asp-action="Index">
</form>
```


The previous markup generates following HTML:

HTML

```
<form method="post">
  <button formaction="/Home">Click Me</button>
  <input type="image" src="..." alt="Or Click Me" formaction="/Home">
</form>
```

Submit to page example

The following markup submits the form to the `About` Razor Page:

C#HTML

```
<form method="post">
  <button asp-page="About">Click Me</button>
  <input type="image" src="..." alt="Or Click Me" asp-page="About">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
  <button formaction="/About">Click Me</button>
  <input type="image" src="..." alt="Or Click Me" formaction="/About">
</form>
```

Submit to route example

Consider the `/Home/Test` endpoint:

C#

```
public class HomeController : Controller
{
    [Route("/Home/Test", Name = "Custom")]
    public string Test()
    {
        return "This is the test page";
    }
}
```

The following markup submits the form to the `/Home/Test` endpoint.

C#HTML

```
<form method="post">
    <button asp-route="Custom">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-route="Custom">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/Home/Test">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home/Test">
</form>
```

The Input Tag Helper

The Input Tag Helper binds an HTML `<input>` [↗](#) element to a model expression in your razor view.

Syntax:

C#HTML

```
<input asp-for="<Expression Name>">
```

The Input Tag Helper:

- Generates the `id` and `name` HTML attributes for the expression name specified in the `asp-for` attribute. `asp-for="Property1.Property2"` is equivalent to `m => m.Property1.Property2`. The name of the expression is what is used for the `asp-for` attribute value. See the [Expression names](#) section for additional information.
- Sets the HTML `type` attribute value based on the model type and [data annotation](#) attributes applied to the model property
- Won't overwrite the HTML `type` attribute value when one is specified
- Generates [HTML5](#) [↗](#) validation attributes from [data annotation](#) attributes applied to model properties
- Has an HTML Helper feature overlap with `Html.TextBoxFor` and `Html.EditorFor`. See the [HTML Helper alternatives to Input Tag Helper](#) section for details.

- Provides strong typing. If the name of the property changes and you don't update the Tag Helper you'll get an error similar to the following:

```
An error occurred during the compilation of a resource required to
process
this request. Please review the following specific error details and
modify
your source code appropriately.
```

```
Type expected
'RegisterViewModel' does not contain a definition for 'Email' and no
extension method 'Email' accepting a first argument of type
'RegisterViewModel'
could be found (are you missing a using directive or an assembly
reference?)
```

The `Input` Tag Helper sets the HTML `type` attribute based on the .NET type. The following table lists some common .NET types and generated HTML type (not every .NET type is listed).

[Expand table](#)

.NET type	Input Type
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime-local" ↗
Byte	type="number"
Int	type="number"
Single, Double	type="number"

The following table shows some common [data annotations](#) attributes that the input tag helper will map to specific input types (not every validation attribute is listed):

[Expand table](#)

Attribute	Input Type
[EmailAddress]	type="email"
[Url]	type="url"

Attribute	Input Type
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

CSHTML

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <button type="submit">Register</button>
</form>
```

The code above generates the following HTML:

HTML

```
<form method="post" action="/Demo/RegisterInput">
    Email:
    <input type="email" data-val="true"
```

```

        data-val-email="The Email Address field is not a valid email
address."
        data-val-required="The Email Address field is required."
        id="Email" name="Email" value=""><br>
    Password:
    <input type="password" data-val="true"
        data-val-required="The Password field is required."
        id="Password" name="Password"><br>
    <button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>

```

The data annotations applied to the `Email` and `Password` properties generate metadata on the model. The Input Tag Helper consumes the model metadata and produces [HTML5](#) `data-val-*` attributes (see [Model Validation](#)). These attributes describe the validators to attach to the input fields. This provides unobtrusive HTML5 and [jQuery](#) validation. The unobtrusive attributes have the format `data-val-rule="Error Message"`, where rule is the name of the validation rule (such as `data-val-required`, `data-val-email`, `data-val-maxlength`, etc.) If an error message is provided in the attribute, it's displayed as the value for the `data-val-rule` attribute. There are also attributes of the form `data-val-ruleName-argumentName="argumentValue"` that provide additional details about the rule, for example, `data-val-maxlength-max="1024"`.

When binding multiple `input` controls to the same property, the generated controls share the same `id`, which makes the generated mark-up invalid. To prevent duplicates, specify the `id` attribute for each control explicitly.

Checkbox hidden input rendering

Checkboxes in HTML5 don't submit a value when they're unchecked. To enable a default value to be sent for an unchecked checkbox, the Input Tag Helper generates an additional hidden input for checkboxes.

For example, consider the following Razor markup that uses the Input Tag Helper for a boolean model property `IsChecked`:

C#HTML

```

<form method="post">
    <input asp-for="@Model.IsChecked" />
    <button type="submit">Submit</button>
</form>

```

The preceding Razor markup generates HTML markup similar to the following:

HTML

```
<form method="post">
  <input name="IsChecked" type="checkbox" value="true" />
  <button type="submit">Submit</button>

  <input name="IsChecked" type="hidden" value="false" />
</form>
```

The preceding HTML markup shows an additional hidden input with a name of `IsChecked` and a value of `false`. By default, this hidden input is rendered at the end of the form. When the form is submitted:

- If the `IsChecked` checkbox input is checked, both `true` and `false` are submitted as values.
- If the `IsChecked` checkbox input is unchecked, only the hidden input value `false` is submitted.

The ASP.NET Core model-binding process reads only the first value when binding to a `bool` value, which results in `true` for checked checkboxes and `false` for unchecked checkboxes.

To configure the behavior of the hidden input rendering, set the [CheckBoxHiddenInputRenderMode](#) property on [MvcViewOptions.HtmlHelperOptions](#). For example:

C#

```
services.Configure<MvcViewOptions>(options =>
    options.HtmlHelperOptions.CheckBoxHiddenInputRenderMode =
        CheckBoxHiddenInputRenderMode.None);
```

The preceding code disables hidden input rendering for checkboxes by setting `CheckBoxHiddenInputRenderMode` to [CheckBoxHiddenInputRenderMode.None](#). For all available rendering modes, see the [CheckBoxHiddenInputRenderMode](#) enum.

HTML Helper alternatives to Input Tag Helper

`Html.TextBox`, `Html.TextBoxFor`, `Html.Editor` and `Html.EditorFor` have overlapping features with the Input Tag Helper. The Input Tag Helper will automatically set the `type` attribute; `Html.TextBox` and `Html.TextBoxFor` won't. `Html.Editor` and `Html.EditorFor` handle collections, complex objects and templates; the Input Tag Helper doesn't. The

Input Tag Helper, `Html.EditorFor` and `Html.TextBoxFor` are strongly typed (they use lambda expressions); `Html.TextBox` and `Html.Editor` are not (they use expression names).

HtmlAttributes

`@Html.Editor()` and `@Html.EditorFor()` use a special `ViewDataDictionary` entry named `htmlAttributes` when executing their default templates. This behavior is optionally augmented using `additionalViewData` parameters. The key "htmlAttributes" is case-insensitive. The key "htmlAttributes" is handled similarly to the `htmlAttributes` object passed to input helpers like `@Html.TextBox()`.

C#HTML

```
@Html.EditorFor(model => model.YourProperty,  
    new { htmlAttributes = new { @class="myCssClass", style="width:100px" } })
```

Expression names

The `asp-for` attribute value is a `ModelExpression` and the right hand side of a lambda expression. Therefore, `asp-for="Property1"` becomes `m => m.Property1` in the generated code which is why you don't need to prefix with `Model`. You can use the "@" character to start an inline expression and move before the `m.`:

C#HTML

```
@{  
    var joe = "Joe";  
}  
  
<input asp-for="@joe">
```

Generates the following:

HTML

```
<input type="text" id="joe" name="joe" value="Joe">
```

With collection properties, `asp-for="CollectionProperty[23].Member"` generates the same name as `asp-for="CollectionProperty[i].Member"` when `i` has the value `23`.

When ASP.NET Core MVC calculates the value of `ModelExpression`, it inspects several sources, including `ModelState`. Consider `<input type="text" asp-for="Name">`. The calculated `value` attribute is the first non-null value from:

- `ModelState` entry with key "Name".
- Result of the expression `Model.Name`.

Navigating child properties

You can also navigate to child properties using the property path of the view model. Consider a more complex model class that contains a child `Address` property.

C#

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

C#

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

In the view, we bind to `Address.AddressLine1`:

CSHTML

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <label>Address: <input asp-for="Address.AddressLine1" /></label><br />
    <button type="submit">Register</button>
</form>
```

The following HTML is generated for `Address.AddressLine1`:

HTML

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1" value="">
```

Expression names and Collections

Sample, a model containing an array of `Colors`:

C#

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

The action method:

C#

```
public IActionResult Edit(int id, int colorIndex)
{
    ViewData["Index"] = colorIndex;
    return View(GetPerson(id));
}
```

The following Razor shows how you access a specific `Color` element:

CSHTML

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

The `Views/Shared/EditorTemplates/String.cshtml` template:

CSHTML

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

Sample using `List<T>`:

```
C#

public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

The following Razor shows how to iterate over a collection:

```
CSSHTML

@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

The `Views/Shared/EditorTemplates/ToDoItem.cshtml` template:

```
CSSHTML

@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
```

```
</td>
```

```
@*
```

```
    This template replaces the following Razor which evaluates the indexer  
    three times.
```

```
    <td>
```

```
        <label asp-for="@Model[i].Name"></label>
```

```
        @Html.DisplayFor(model => model[i].Name)
```

```
    </td>
```

```
    <td>
```

```
        <input asp-for="@Model[i].IsDone" />
```

```
    </td>
```

```
*@
```

`foreach` should be used if possible when the value is going to be used in an `asp-for` or `Html.DisplayFor` equivalent context. In general, `for` is better than `foreach` (if the scenario allows it) because it doesn't need to allocate an enumerator; however, evaluating an indexer in a LINQ expression can be expensive and should be minimized.

⚠ Note

The commented sample code above shows how you would replace the lambda expression with the `@` operator to access each `ToDoItem` in the list.

The Textarea Tag Helper

The `Textarea Tag Helper` tag helper is similar to the `Input Tag Helper`.

- Generates the `id` and `name` attributes, and the data validation attributes from the model for a `<textarea>` [↗](#) element.
- Provides strong typing.
- HTML Helper alternative: `Html.TextAreaFor`

Sample:

```
C#
```

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

```
namespace FormsTagHelper.ViewModels
```

```
{
```

```
    public class DescriptionViewModel
```

```

{
    [MinLength(5)]
    [MaxLength(1024)]
    public string Description { get; set; }
}

```

C#HTML

```

@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>

```

The following HTML is generated:

HTML

```

<form method="post" action="/Demo/RegisterTextArea">
    <textarea data-val="true"
        data-val-maxlength="The field Description must be a string or array type
with a maximum length of &#x27;1024&#x27;."
        data-val-maxlength-max="1024"
        data-val-minlength="The field Description must be a string or array type
with a minimum length of &#x27;5&#x27;."
        data-val-minlength-min="5"
        id="Description" name="Description">
    </textarea>
    <button type="submit">Test</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>

```

The Label Tag Helper

- Generates the label caption and `for` attribute on a `<label>` [↗](#) element for an expression name
- HTML Helper alternative: `Html.LabelFor`.

The `Label Tag Helper` provides the following benefits over a pure HTML label element:

- You automatically get the descriptive label value from the `Display` attribute. The intended display name might change over time, and the combination of `Display` attribute and Label Tag Helper will apply the `Display` everywhere it's used.

- Less markup in source code
- Strong typing with the model property.

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

CSHTML

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

The following HTML is generated for the `<label>` element:

HTML

```
<label for="Email">Email Address</label>
```

The Label Tag Helper generated the `for` attribute value of "Email", which is the ID associated with the `<input>` element. The Tag Helpers generate consistent `id` and `for` elements so they can be correctly associated. The caption in this sample comes from the `Display` attribute. If the model didn't contain a `Display` attribute, the caption would be the property name of the expression. To override the default caption, add a caption inside the label tag.

The Validation Tag Helpers

There are two Validation Tag Helpers. The `Validation Message Tag Helper` (which displays a validation message for a single property on your model), and the `Validation Summary Tag Helper` (which displays a summary of validation errors). The `Input Tag Helper` adds HTML5 client side validation attributes to input elements based on data annotation attributes on your model classes. Validation is also performed on the server. The Validation Tag Helper displays these error messages when a validation error occurs.

The Validation Message Tag Helper

- Adds the [HTML5](#) `data-valmsg-for="property"` attribute to the [span](#) element, which attaches the validation error messages on the input field of the specified model property. When a client side validation error occurs, [jQuery](#) displays the error message in the `` element.
- Validation also takes place on the server. Clients may have JavaScript disabled and some validation can only be done on the server side.
- HTML Helper alternative: `Html.ValidationMessageFor`

The `Validation Message Tag Helper` is used with the `asp-validation-for` attribute on an HTML [span](#) element.

C#HTML

```
<span asp-validation-for="Email"></span>
```

The Validation Message Tag Helper will generate the following HTML:

HTML

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

You generally use the `Validation Message Tag Helper` after an `Input Tag Helper` for the same property. Doing so displays any validation error messages near the input that caused the error.

❗ Note

You must have a view with the correct JavaScript and [jQuery](#) script references in place for client side validation. See [Model Validation](#) for more information.

When a server side validation error occurs (for example when you have custom server side validation or client-side validation is disabled), MVC places that error message as the body of the `` element.

HTML

```
<span class="field-validation-error" data-valmsg-for="Email"
      data-valmsg-replace="true">
    The Email Address field is required.
</span>
```

The Validation Summary Tag Helper

- Targets `<div>` elements with the `asp-validation-summary` attribute
- HTML Helper alternative: `@Html.ValidationSummary`

The `Validation Summary Tag Helper` is used to display a summary of validation messages. The `asp-validation-summary` attribute value can be any of the following:

[Expand table](#)

<code>asp-validation-summary</code>	Validation messages displayed
All	Property and model level
ModelOnly	Model
None	None

Sample

In the following example, the data model has `DataAnnotation` attributes, which generates validation error messages on the `<input>` element. When a validation error occurs, the Validation Tag Helper displays the error message:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
```

```

    [EmailAddress]
    [Display(Name = "Email Address")]
    public string Email { get; set; }

    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }
}

```

CSHTML

```

@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    <label>Email: <input asp-for="Email" /></label> <br />
    <span asp-validation-for="Email"></span><br />
    <label>Password: <input asp-for="Password" /></label><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>

```

The generated HTML (when the model is valid):

HTML

```

<form action="/DemoReg/Register" method="post">
    <label>Email: <input name="Email" id="Email" type="email" value=""
        data-val-required="The Email field is required."
        data-val-email="The Email field is not a valid email address."
        data-val="true"></label><br>
    <span class="field-validation-valid" data-valmsg-replace="true"
        data-valmsg-for="Email"></span><br>
    <label>Password: <input name="Password" id="Password" type="password"
        data-val-required="The Password field is required." data-val="true">
    </label><br>
    <span class="field-validation-valid" data-valmsg-replace="true"
        data-valmsg-for="Password"></span><br>
    <button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for
    brevity>">
</form>

```

The Select Tag Helper

- Generates [select](#) and associated [option](#) elements for properties of your model.
- Has an HTML Helper alternative `Html.DropDownListFor` and `Html.ListBoxFor`

The `Select Tag Helper` `asp-for` specifies the model property name for the [select](#) element and `asp-items` specifies the [option](#) elements. For example:

C#HTML

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Sample:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new
List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA" },
        };
    }
}
```

The `Index` method initializes the `CountryViewModel`, sets the selected country and passes it to the `Index` view.

C#

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

The HTTP POST `Index` method displays the selection:

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
```

```
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country + " selected";
        return RedirectToAction("IndexSuccess", new { message = msg });
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

The `Index` view:

C#HTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Which generates the following HTML (with "CA" selected):

HTML

```
<form method="post" action="/">
    <select id="Country" name="Country">
        <option value="MX">Mexico</option>
        <option selected="selected" value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

⚠ Note

We don't recommend using `ViewBag` or `ViewData` with the Select Tag Helper. A view model is more robust at providing MVC metadata and generally less problematic.

The `asp-for` attribute value is a special case and doesn't require a `Model` prefix, the other Tag Helper attributes do (such as `asp-items`)

C#HTML

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Enum binding

It's often convenient to use `<select>` with an `enum` property and generate the `SelectListItem` elements from the `enum` values.

Sample:

C#

```
public class CountryEnumViewModel
{
    public CountryEnum EnumCountry { get; set; }
}
```

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The `GetEnumSelectList` method generates a `SelectList` object for an enum.

C#HTML

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
        asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
```

```
<br /><button type="submit">Register</button>
</form>
```

You can mark your enumerator list with the `Display` attribute to get a richer UI:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The following HTML is generated:

HTML

```
<form method="post" action="/Home/IndexEnum">
    <select data-val="true" data-val-required="The EnumCountry field is
    required."
        id="EnumCountry" name="EnumCountry">
        <option value="0">United Mexican States</option>
        <option value="1">United States of America</option>
        <option value="2">Canada</option>
        <option value="3">France</option>
        <option value="4">Germany</option>
        <option selected="selected" value="5">Spain</option>
    </select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
    for brevity>">
</form>
```

Option Group

The HTML `<optgroup>` [↗](#) element is generated when the view model contains one or more `SelectListGroup` objects.

The `CountryViewModelGroup` groups the `SelectListItem` elements into the "North America" and "Europe" groups:

C#

```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America" };
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
        };
    }
}
```

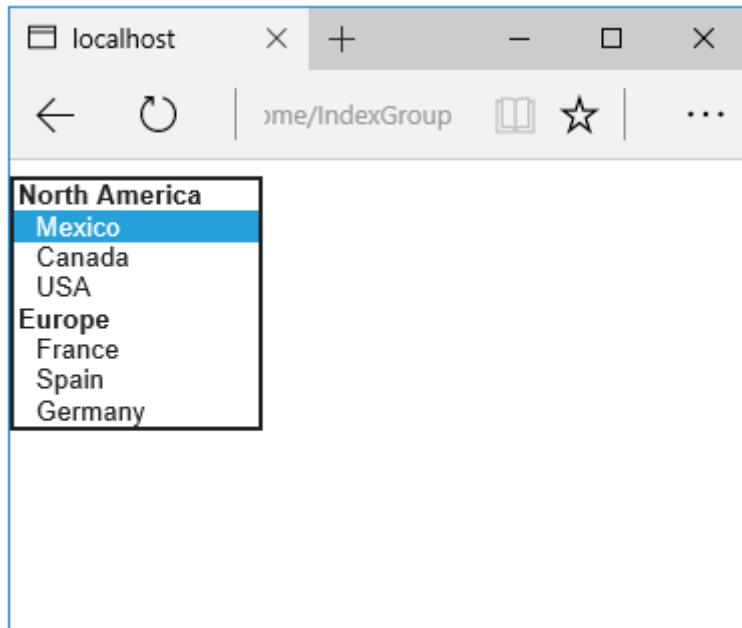
```

public string Country { get; set; }

public List<SelectListItem> Countries { get; }

```

The two groups are shown below:



The generated HTML:

HTML

```

<form method="post" action="/Home/IndexGroup">
  <select id="Country" name="Country">
    <optgroup label="North America">
      <option value="MEX">Mexico</option>
      <option value="CAN">Canada</option>
      <option value="US">USA</option>
    </optgroup>
    <optgroup label="Europe">
      <option value="FR">France</option>
      <option value="ES">Spain</option>
      <option value="DE">Germany</option>
    </optgroup>
  </select>
  <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>

```

Multiple select

The Select Tag Helper will automatically generate the `multiple = "multiple"` attribute if the property specified in the `asp-for` attribute is an `IEnumerable`. For example, given the following model:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new
List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA" },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain" },
            new SelectListItem { Value = "DE", Text = "Germany" }
        };
    }
}
```

With the following view:

C#HTML

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Generates the following HTML:

HTML

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
        multiple="multiple"
        name="CountryCodes"><option value="MX">Mexico</option>
    <option value="CA">Canada</option>
    <option value="US">USA</option>
    <option value="FR">France</option>
```

```

<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>

```

No selection

If you find yourself using the "not specified" option in multiple pages, you can create a template to eliminate repeating the HTML:

CSSHTML

```

@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>

```

The `Views/Shared/EditorTemplates/CountryViewModel.csshtml` template:

CSSHTML

```

@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>

```

Adding HTML `<option>` [↗](#) elements isn't limited to the *No selection* case. For example, the following view and action method will generate HTML similar to the code above:

C#

```

public IActionResult IndexNone()
{
    var model = new CountryViewModel();
    model.Countries.Insert(0, new SelectListItem("<none>", ""));
    return View(model);
}

```

CSSHTML


```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

The correct `<option>` element will be selected (contain the `selected="selected"` attribute) depending on the current `Country` value.

C#

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

HTML

```
<form method="post" action="/Home/IndexEmpty">
    <select id="Country" name="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA" selected="selected">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

Additional resources

- [Tag Helpers in ASP.NET Core](#)
- [HTML Form element](#) ↗
- [Request Verification Token](#)
- [Model Binding in ASP.NET Core](#)
- [Model validation in ASP.NET Core MVC](#)
- [IAttributeAdapter Interface](#)

- [Code snippets for this document](#) ↗

Link Tag Helper in ASP.NET Core

Article • 06/27/2022

By [Rick Anderson](#) 

The [Link Tag Helper](#) generates a link to a primary or fall back CSS file. Typically the primary CSS file is on a [Content Delivery Network](#) (CDN).

A CDN:

- Provides several [performance advantages](#) vs hosting the asset with the web app.
- Should not be relied on as the only source for the asset. CDNs are not always available, therefore a reliable fallback should be used. Typically the fallback is the site hosting the web app.

The Link Tag Helper allows you to specify a CDN for the CSS file and a fallback when the CDN is not available. The Link Tag Helper provides the performance advantage of a CDN with the robustness of local hosting.

The following Razor markup shows the `head` element of a layout file created with the ASP.NET Core web app template:

CSSHTML

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>@ViewData["Title"] - WebLinkTH</title>

  <environment include="Development">
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css"
  />
</environment>
<environment exclude="Development">
  <link rel="stylesheet"
    href="https://cdnjs.cloudflare.com/ajax/libs/twitter-
bootstrap/4.1.3/css/bootstrap.css"
    asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.css"
    asp-fallback-test-class="sr-only" asp-fallback-test-
property="position"
    asp-fallback-test-value="absolute"
    crossorigin="anonymous"
    integrity="sha256-
eSi1q2PG6J7g7ib17yAawMcrr5GrtohYChqibrV7PBE=" />
</environment>
```

```
<link rel="stylesheet" href="~/css/site.css" />
</head>
```

The following is rendered HTML from the preceding code (in a non-Development environment):

HTML

```
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Home page - WebLinkTH</title>
  <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/twitter-
bootstrap/4.1.3/css/bootstrap.css"
  crossorigin="anonymous" integrity="sha256-eS<snip>BE=" />
  <meta name="x-stylesheet-fallback-test" content="" class="sr-only" />
  <script>
    !function (a, b, c, d) {
      var e, f = document,
        g = f.getElementsByTagName("SCRIPT"),
        h = g[g.length - 1].previousElementSibling,
        i = f.defaultView && f.defaultView.getComputedStyle ?
f.defaultView.getComputedStyle(h) : h.currentStyle;
      if (i && i[a] !== b) for (e = 0; e < c.length; e++)
        f.write('<link href="' + c[e] + '" ' + d + "/>")
    }
    ("position", "absolute",
["~/lib/bootstrap/dist/css/bootstrap.css"],
    "rel=\u0022stylesheet\u0022
crossorigin=\u0022anonymous\u0022 integrity=\abc<snip>BE=\u0022 ");
  </script>

  <link rel="stylesheet" href="/css/site.css" />
</head>
```

In the preceding code, the Link Tag Helper generated the `<meta name="x-stylesheet-fallback-test" content="" class="sr-only" />` element and the following JavaScript which is used to verify the requested `bootstrap.css` file is available on the CDN. In this case, the CSS file was available so the Tag Helper generated the `<link />` element with the CDN CSS file.

Commonly used Link Tag Helper attributes

See [Link Tag Helper](#) for all the Link Tag Helper attributes, properties, and methods.

href

Preferred address of the linked resource. The address is passed thought to the generated HTML in all cases.

asp-fallback-href

The URL of a CSS stylesheet to fallback to in the case the primary URL fails.

asp-fallback-test-class

The class name defined in the stylesheet to use for the fallback test. For more information, see [FallbackTestClass](#).

asp-fallback-test-property

The CSS property name to use for the fallback test. For more information, see [FallbackTestProperty](#).

asp-fallback-test-value

The CSS property value to use for the fallback test. For more information, see [FallbackTestValue](#).

Additional resources

- [Tag Helpers in ASP.NET Core](#)
- [Areas in ASP.NET Core](#)
- [Introduction to Razor Pages in ASP.NET Core](#)
- [Compatibility version for ASP.NET Core MVC](#)

Partial Tag Helper in ASP.NET Core

Article • 06/18/2024

By [Scott Addie](#) 

For an overview of Tag Helpers, see [Tag Helpers in ASP.NET Core](#).

[View or download sample code](#)  ([how to download](#))

Overview

The Partial Tag Helper is used for rendering a [partial view](#) in Razor Pages and MVC apps. Consider that it:

- Requires ASP.NET Core 2.1 or later.
- Is an alternative to [HTML Helper syntax](#).
- Renders the partial view asynchronously.

The HTML Helper options for rendering a partial view include:

- [@await Html.PartialAsync](#)
- [@await Html.RenderPartialAsync](#)
- [@Html.Partial](#)
- [@Html.RenderPartial](#)

The *Product* model is used in samples throughout this document:

C#

```
namespace TagHelpersBuiltIn.Models
{
    public class Product
    {
        public int Number { get; set; }

        public string Name { get; set; }

        public string Description { get; set; }
    }
}
```

An inventory of the Partial Tag Helper attributes follows.

name

The `name` attribute is required. It indicates the name or the path of the partial view to be rendered. When a partial view name is provided, the [view discovery](#) process is initiated. That process is bypassed when an explicit path is provided. For all acceptable `name` values, see [Partial view discovery](#).

The following markup uses an explicit path, indicating that `_ProductPartial.cshtml` is to be loaded from the *Shared* folder. Using the `for` attribute, a model is passed to the partial view for binding.

C#HTML

```
<partial name="Shared/_ProductPartial.cshtml" for="Product">
```

for

The `for` attribute assigns a [ModelExpression](#) to be evaluated against the current model. A `ModelExpression` infers the `@Model.` syntax. For example, `for="Product"` can be used instead of `for="@Model.Product"`. This default inference behavior is overridden by using the `@` symbol to define an inline expression.

The following markup loads `_ProductPartial.cshtml`:

C#HTML

```
<partial name="_ProductPartial" for="Product">
```

The partial view is bound to the associated page model's `Product` property:

C#

```
using Microsoft.AspNetCore.Mvc.RazorPages;
using TagHelpersBuiltIn.Models;

namespace TagHelpersBuiltIn.Pages
{
    public class ProductModel : PageModel
    {
        public Product Product { get; set; }

        public void OnGet()
        {
            Product = new Product
            {
                Number = 1,
                Name = "Test product",
            }
        }
    }
}
```

```

        Description = "This is a test product"
    };
}
}
}

```

model

The `model` attribute assigns a model instance to pass to the partial view. The `model` attribute can't be used with the `for` attribute.

In the following markup, a new `Product` object is instantiated and passed to the `model` attribute for binding:

CSSHTML

```

<partial name="_ProductPartial"
    model='new Product { Number = 1, Name = "Test product", Description
= "This is a test" }'>

```

view-data

The `view-data` attribute assigns a `ViewDataDictionary` to pass to the partial view. The following markup makes the entire `ViewData` collection accessible to the partial view:

CSSHTML

```

@{
    ViewData["IsNumberReadOnly"] = true;
}

<partial name="_ProductViewDataPartial" for="Product" view-data="ViewData">

```

In the preceding code, the `IsNumberReadOnly` key value is set to `true` and added to the `ViewData` collection. Consequently, `ViewData["IsNumberReadOnly"]` is made accessible within the following partial view:

CSSHTML

```

@model TagHelpersBuiltIn.Models.Product

<div class="form-group">
    <label asp-for="Number"></label>
    @if ((bool)ViewData["IsNumberReadOnly"])

```



```

    {
        <input asp-for="Number" type="number" class="form-control" readonly
    />
    }
    else
    {
        <input asp-for="Number" type="number" class="form-control" />
    }
</div>
<div class="form-group">
    <label asp-for="Name"></label>
    <input asp-for="Name" type="text" class="form-control" />
</div>
<div class="form-group">
    <label asp-for="Description"></label>
    <textarea asp-for="Description" rows="4" cols="50" class="form-control">
</textarea>
</div>

```

In this example, the value of `ViewData["IsNumberReadOnly"]` determines whether the *Number* field is displayed as read only.

Migrate from an HTML Helper

Consider the following asynchronous HTML Helper example. A collection of products is iterated and displayed. Per the `PartialAsync` method's first parameter, the `_ProductPartial.cshtml` partial view is loaded. An instance of the `Product` model is passed to the partial view for binding.

CSHTML

```

@foreach (var product in Model.Products)
{
    @await Html.PartialAsync("_ProductPartial", product)
}

```

The following Partial Tag Helper achieves the same asynchronous rendering behavior as the `PartialAsync` HTML Helper. The `model` attribute is assigned a `Product` model instance for binding to the partial view.

CSHTML

```

@foreach (var product in Model.Products)
{
    <partial name="_ProductPartial" model="@product" />
}

```

Additional resources

- [Partial views in ASP.NET Core](#)
- [Views in ASP.NET Core MVC](#)

Persist Component State Tag Helper in ASP.NET Core

Article • 11/15/2024

The Persist Component State Tag Helper saves the state of non-routable Razor components rendered in a page or view of a Razor Pages or MVC app.

Prerequisites

Follow the guidance in the *Use non-routable components in pages or views* section of the [Integrate ASP.NET Core Razor components with MVC or Razor Pages](#) article.

Persist state for prerendered components

To persist state for prerendered components, use the [Persist Component State Tag Helper](#) ([reference source](#) [↗](#)). Add the Tag Helper's tag, `<persist-component-state />`, inside the closing `</body>` tag of the layout in an app that prerenders components.

❗ Note

Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see [How to select a version tag of ASP.NET Core source code \(dotnet/AspNetCore.Docs #26205\)](#) [↗](#).

In `Pages/Shared/_Layout.cshtml` for embedded components that are either WebAssembly prerendered (`WebAssemblyPrerendered`) or server prerendered (`ServerPrerendered`):

C#HTML

```
<body>
    ...

    <persist-component-state />
</body>
```

Decide what state to persist using the [PersistentComponentState](#) service.

[PersistentComponentState.RegisterOnPersisting](#) registers a callback to persist the component state before the app is paused. The state is retrieved when the application resumes.

For more information and examples, see [Prerender ASP.NET Core Razor components](#).

Additional resources

- [Component Tag Helper in ASP.NET Core](#)
- [Prerender ASP.NET Core Razor components](#)
- [ComponentTagHelper](#)
- [Tag Helpers in ASP.NET Core](#)
- [ASP.NET Core Razor components](#)

Script Tag Helper in ASP.NET Core

Article • 09/23/2024

By [Rick Anderson](#) 

The [Script Tag Helper](#) generates a link to a primary or fall back script file. Typically the primary script file is on a [Content Delivery Network](#) (CDN).

A CDN:


- Provides several [performance advantages](#) vs hosting the asset with the web app.
- Should not be relied on as the only source for the asset. CDNs are not always available, therefore a reliable fallback should be used. Typically the fallback is the site hosting the web app.

The Script Tag Helper allows you to specify a CDN for the script file and a fallback when the CDN is not available. The Script Tag Helper provides the performance advantage of a CDN with the robustness of local hosting.

The following Razor markup shows a `<script>` element with a fallback:

HTML

```
<script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-3.3.1.js"
    asp-fallback-src="~/lib/jquery/dist/jquery.js"
    asp-fallback-test="window.jQuery"
    crossorigin="anonymous"
    integrity="sha384-
tsQFqpEReu7ZLhBV2VZ1Au7zcOV+rXbYlF2cqB8txI/8aZajjp4Bqd+V6D5IgvKT" >
</script>
```

Don't use the `<script>` element's [defer](#)  attribute to defer loading the CDN script. The Script Tag Helper renders JavaScript that immediately executes the [asp-fallback-test](#) expression. The expression fails if loading the CDN script is deferred.

Commonly used Script Tag Helper attributes

See [Script Tag Helper](#) for all the Script Tag Helper attributes, properties, and methods.

src

Address of the external script to use.

asp-append-version

When `asp-append-version` is specified with a `true` value along with a `src` [↗](#) attribute, a unique version is generated.

For a Tag Helper to generate a version for a static file outside `wwwroot`, see [Serve files from multiple locations](#)

asp-fallback-src

The URL of a Script tag to fallback to in the case the primary one fails.

asp-fallback-src-exclude

A comma-separated list of globbed file patterns of JavaScript scripts to exclude from the fallback list, in the case the primary one fails. The glob patterns are assessed relative to the application's `webroot` setting. Must be used in conjunction with `asp-fallback-src-include`.

asp-fallback-src-include

A comma-separated list of globbed file patterns of JavaScript scripts to fallback to in the case the primary one fails. The glob patterns are assessed relative to the application's `webroot` setting.

asp-fallback-test

The script method defined in the primary script to use for the fallback test. For more information, see [FallbackTestExpression](#).

asp-order

When a set of `ITagHelper` instances are executed, their `Init(TagHelperContext)` methods are first invoked in the specified order; then their `ProcessAsync(TagHelperContext, TagHelperOutput)` methods are invoked in the specified order. Lower values are executed first.

asp-src-exclude

A comma-separated list of globbed file patterns of JavaScript scripts to exclude from loading. The glob patterns are assessed relative to the application's `webroot` setting. Must be used in conjunction with `asp-src-include`.

asp-src-include

A comma-separated list of globbed file patterns of JavaScript scripts to load. The glob patterns are assessed relative to the application's `webroot` setting.

asp-suppress-fallback-integrity

Boolean value that determines if an integrity hash will be compared with the `asp-fallback-src` value.

Additional resources

- [Tag Helpers in ASP.NET Core](#)
- [Areas in ASP.NET Core](#)
- [Introduction to Razor Pages in ASP.NET Core](#)
- [Compatibility version for ASP.NET Core MVC](#)

Tag Helpers in forms in ASP.NET Core

Article • 09/27/2024

By [Rick Anderson](#), [N. Taylor Mullen](#), [Dave Paquette](#), and [Jerrie Pelser](#)

This document demonstrates working with Forms and the HTML elements commonly used on a Form. The HTML [Form](#) element provides the primary mechanism web apps use to post back data to the server. Most of this document describes [Tag Helpers](#) and how they can help you productively create robust HTML forms. We recommend you read [Introduction to Tag Helpers](#) before you read this document.

In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers don't replace HTML Helpers and there's not a Tag Helper for each HTML Helper. When an HTML Helper alternative exists, it's mentioned.

The Form Tag Helper

The [Form Tag Helper](#):

- Generates the HTML [<FORM>](#) `action` attribute value for a MVC controller action or named route
- Generates a hidden [Request Verification Token](#) to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method)
- Provides the `asp-route-<Parameter Name>` attribute, where `<Parameter Name>` is added to the route values. The `routeValues` parameters to `Html.BeginForm` and `Html.BeginRouteForm` provide similar functionality.
- Has an HTML Helper alternative `Html.BeginForm` and `Html.BeginRouteForm`

Sample:

C#HTML

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

The Form Tag Helper above generates the following HTML:

HTML

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

The MVC runtime generates the `action` attribute value from the Form Tag Helper attributes `asp-controller` and `asp-action`. The Form Tag Helper also generates a hidden [Request Verification Token](#) to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method). Protecting a pure HTML Form from cross-site request forgery is difficult, the Form Tag Helper provides this service for you.

Using a named route

The `asp-route` Tag Helper attribute can also generate markup for the HTML `action` attribute. An app with a [route](#) named `register` could use the following markup for the registration page:

C#HTML

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Many of the views in the *Views/Account* folder (generated when you create a new web app with *Individual User Accounts*) contain the `asp-route-returnurl` attribute:

C#HTML

```
<form asp-controller="Account" asp-action="Login"
      asp-route-returnurl="@ViewData["ReturnUrl"]"
      method="post" class="form-horizontal" role="form">
```

ⓘ Note

With the built in templates, `returnUrl` is only populated automatically when you try to access an authorized resource but are not authenticated or authorized. When you attempt an unauthorized access, the security middleware redirects you to the login page with the `returnUrl` set.

The Form Action Tag Helper

The Form Action Tag Helper generates the `formaction` attribute on the generated `<button ...>` or `<input type="image" ...>` tag. The `formaction` attribute controls where a form submits its data. It binds to `<input>` [↗](#) elements of type `image` and `<button>` [↗](#) elements. The Form Action Tag Helper enables the usage of several [AnchorTagHelper](#) `asp-` attributes to control what `formaction` link is generated for the corresponding element.

Supported [AnchorTagHelper](#) attributes to control the value of `formaction`:

[↗](#) Expand table

Attribute	Description
asp-controller	The name of the controller.
asp-action	The name of the action method.
asp-area	The name of the area.
asp-page	The name of the Razor page.
asp-page-handler	The name of the Razor page handler.
asp-route	The name of the route.
asp-route-{value}	A single URL route value. For example, <code>asp-route-id="1234"</code> .
asp-all-route-data	All route values.
asp-fragment	The URL fragment.

Submit to controller example

The following markup submits the form to the `Index` action of `HomeController` when the input or button are selected:

C#HTML

```
<form method="post">
  <button asp-controller="Home" asp-action="Index">Click Me</button>
  <input type="image" src="..." alt="Or Click Me" asp-controller="Home"
        asp-action="Index">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
  <button formaction="/Home">Click Me</button>
  <input type="image" src="..." alt="Or Click Me" formaction="/Home">
</form>
```

Submit to page example

The following markup submits the form to the `About` Razor Page:

C#HTML

```
<form method="post">
  <button asp-page="About">Click Me</button>
  <input type="image" src="..." alt="Or Click Me" asp-page="About">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
  <button formaction="/About">Click Me</button>
  <input type="image" src="..." alt="Or Click Me" formaction="/About">
</form>
```

Submit to route example

Consider the `/Home/Test` endpoint:

C#

```
public class HomeController : Controller
{
    [Route("/Home/Test", Name = "Custom")]
    public string Test()
    {
        return "This is the test page";
    }
}
```

The following markup submits the form to the `/Home/Test` endpoint.

C#HTML

```
<form method="post">
    <button asp-route="Custom">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-route="Custom">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/Home/Test">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home/Test">
</form>
```

The Input Tag Helper

The Input Tag Helper binds an HTML `<input>` [↗](#) element to a model expression in your razor view.

Syntax:

C#HTML

```
<input asp-for="<Expression Name>">
```

The Input Tag Helper:

- Generates the `id` and `name` HTML attributes for the expression name specified in the `asp-for` attribute. `asp-for="Property1.Property2"` is equivalent to `m => m.Property1.Property2`. The name of the expression is what is used for the `asp-for` attribute value. See the [Expression names](#) section for additional information.
- Sets the HTML `type` attribute value based on the model type and [data annotation](#) attributes applied to the model property
- Won't overwrite the HTML `type` attribute value when one is specified
- Generates [HTML5](#) [↗](#) validation attributes from [data annotation](#) attributes applied to model properties
- Has an HTML Helper feature overlap with `Html.TextBoxFor` and `Html.EditorFor`. See the [HTML Helper alternatives to Input Tag Helper](#) section for details.

- Provides strong typing. If the name of the property changes and you don't update the Tag Helper you'll get an error similar to the following:

```
An error occurred during the compilation of a resource required to
process
this request. Please review the following specific error details and
modify
your source code appropriately.
```

```
Type expected
'RegisterViewModel' does not contain a definition for 'Email' and no
extension method 'Email' accepting a first argument of type
'RegisterViewModel'
could be found (are you missing a using directive or an assembly
reference?)
```

The `Input` Tag Helper sets the HTML `type` attribute based on the .NET type. The following table lists some common .NET types and generated HTML type (not every .NET type is listed).

[Expand table](#)

.NET type	Input Type
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime-local" ↗
Byte	type="number"
Int	type="number"
Single, Double	type="number"

The following table shows some common [data annotations](#) attributes that the input tag helper will map to specific input types (not every validation attribute is listed):

[Expand table](#)

Attribute	Input Type
[EmailAddress]	type="email"
[Url]	type="url"

Attribute	Input Type
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

CSHTML

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <button type="submit">Register</button>
</form>
```

The code above generates the following HTML:

HTML

```
<form method="post" action="/Demo/RegisterInput">
    Email:
    <input type="email" data-val="true"
```

```

        data-val-email="The Email Address field is not a valid email
address."
        data-val-required="The Email Address field is required."
        id="Email" name="Email" value=""><br>
    Password:
    <input type="password" data-val="true"
        data-val-required="The Password field is required."
        id="Password" name="Password"><br>
    <button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>

```

The data annotations applied to the `Email` and `Password` properties generate metadata on the model. The Input Tag Helper consumes the model metadata and produces [HTML5](#) `data-val-*` attributes (see [Model Validation](#)). These attributes describe the validators to attach to the input fields. This provides unobtrusive HTML5 and [jQuery](#) validation. The unobtrusive attributes have the format `data-val-rule="Error Message"`, where rule is the name of the validation rule (such as `data-val-required`, `data-val-email`, `data-val-maxlength`, etc.) If an error message is provided in the attribute, it's displayed as the value for the `data-val-rule` attribute. There are also attributes of the form `data-val-ruleName-argumentName="argumentValue"` that provide additional details about the rule, for example, `data-val-maxlength-max="1024"`.

When binding multiple `input` controls to the same property, the generated controls share the same `id`, which makes the generated mark-up invalid. To prevent duplicates, specify the `id` attribute for each control explicitly.

Checkbox hidden input rendering

Checkboxes in HTML5 don't submit a value when they're unchecked. To enable a default value to be sent for an unchecked checkbox, the Input Tag Helper generates an additional hidden input for checkboxes.

For example, consider the following Razor markup that uses the Input Tag Helper for a boolean model property `IsChecked`:

C#HTML

```

<form method="post">
    <input asp-for="@Model.IsChecked" />
    <button type="submit">Submit</button>
</form>

```

The preceding Razor markup generates HTML markup similar to the following:

HTML

```
<form method="post">
  <input name="IsChecked" type="checkbox" value="true" />
  <button type="submit">Submit</button>

  <input name="IsChecked" type="hidden" value="false" />
</form>
```

The preceding HTML markup shows an additional hidden input with a name of `IsChecked` and a value of `false`. By default, this hidden input is rendered at the end of the form. When the form is submitted:

- If the `IsChecked` checkbox input is checked, both `true` and `false` are submitted as values.
- If the `IsChecked` checkbox input is unchecked, only the hidden input value `false` is submitted.

The ASP.NET Core model-binding process reads only the first value when binding to a `bool` value, which results in `true` for checked checkboxes and `false` for unchecked checkboxes.

To configure the behavior of the hidden input rendering, set the [CheckBoxHiddenInputRenderMode](#) property on [MvcViewOptions.HtmlHelperOptions](#). For example:

C#

```
services.Configure<MvcViewOptions>(options =>
    options.HtmlHelperOptions.CheckBoxHiddenInputRenderMode =
        CheckBoxHiddenInputRenderMode.None);
```

The preceding code disables hidden input rendering for checkboxes by setting `CheckBoxHiddenInputRenderMode` to [CheckBoxHiddenInputRenderMode.None](#). For all available rendering modes, see the [CheckBoxHiddenInputRenderMode](#) enum.

HTML Helper alternatives to Input Tag Helper

`Html.TextBox`, `Html.TextBoxFor`, `Html.Editor` and `Html.EditorFor` have overlapping features with the Input Tag Helper. The Input Tag Helper will automatically set the `type` attribute; `Html.TextBox` and `Html.TextBoxFor` won't. `Html.Editor` and `Html.EditorFor` handle collections, complex objects and templates; the Input Tag Helper doesn't. The

Input Tag Helper, `Html.EditorFor` and `Html.TextBoxFor` are strongly typed (they use lambda expressions); `Html.TextBox` and `Html.Editor` are not (they use expression names).

HtmlAttributes

`@Html.Editor()` and `@Html.EditorFor()` use a special `ViewDataDictionary` entry named `htmlAttributes` when executing their default templates. This behavior is optionally augmented using `additionalViewData` parameters. The key "htmlAttributes" is case-insensitive. The key "htmlAttributes" is handled similarly to the `htmlAttributes` object passed to input helpers like `@Html.TextBox()`.

C#HTML

```
@Html.EditorFor(model => model.YourProperty,  
    new { htmlAttributes = new { @class="myCssClass", style="width:100px" } })
```

Expression names

The `asp-for` attribute value is a `ModelExpression` and the right hand side of a lambda expression. Therefore, `asp-for="Property1"` becomes `m => m.Property1` in the generated code which is why you don't need to prefix with `Model`. You can use the "@" character to start an inline expression and move before the `m.`:

C#HTML

```
@{  
    var joe = "Joe";  
}  
  
<input asp-for="@joe">
```

Generates the following:

HTML

```
<input type="text" id="joe" name="joe" value="Joe">
```

With collection properties, `asp-for="CollectionProperty[23].Member"` generates the same name as `asp-for="CollectionProperty[i].Member"` when `i` has the value `23`.

When ASP.NET Core MVC calculates the value of `ModelExpression`, it inspects several sources, including `ModelState`. Consider `<input type="text" asp-for="Name">`. The calculated `value` attribute is the first non-null value from:

- `ModelState` entry with key "Name".
- Result of the expression `Model.Name`.

Navigating child properties

You can also navigate to child properties using the property path of the view model. Consider a more complex model class that contains a child `Address` property.

C#

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

C#

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

In the view, we bind to `Address.AddressLine1`:

CSHTML

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <label>Address: <input asp-for="Address.AddressLine1" /></label><br />
    <button type="submit">Register</button>
</form>
```

The following HTML is generated for `Address.AddressLine1`:

HTML

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1" value="">
```

Expression names and Collections

Sample, a model containing an array of `Colors`:

C#

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

The action method:

C#

```
public IActionResult Edit(int id, int colorIndex)
{
    ViewData["Index"] = colorIndex;
    return View(GetPerson(id));
}
```

The following Razor shows how you access a specific `Color` element:

CSHTML

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

The `Views/Shared/EditorTemplates/String.cshtml` template:

CSHTML

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

Sample using `List<T>`:

```
C#

public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

The following Razor shows how to iterate over a collection:

```
C#HTML

@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

The `Views/Shared/EditorTemplates/ToDoItem.cshtml` template:

```
C#HTML

@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
```

```
</td>
```

```
@*
```

```
    This template replaces the following Razor which evaluates the indexer  
    three times.
```

```
    <td>
```

```
        <label asp-for="@Model[i].Name"></label>
```

```
        @Html.DisplayFor(model => model[i].Name)
```

```
    </td>
```

```
    <td>
```

```
        <input asp-for="@Model[i].IsDone" />
```

```
    </td>
```

```
*@
```

`foreach` should be used if possible when the value is going to be used in an `asp-for` or `Html.DisplayFor` equivalent context. In general, `for` is better than `foreach` (if the scenario allows it) because it doesn't need to allocate an enumerator; however, evaluating an indexer in a LINQ expression can be expensive and should be minimized.

⚠ Note

The commented sample code above shows how you would replace the lambda expression with the `@` operator to access each `ToDoItem` in the list.

The Textarea Tag Helper

The `Textarea Tag Helper` tag helper is similar to the `Input Tag Helper`.

- Generates the `id` and `name` attributes, and the data validation attributes from the model for a `<textarea>` [↗](#) element.
- Provides strong typing.
- HTML Helper alternative: `Html.TextAreaFor`

Sample:

```
C#
```

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

```
namespace FormsTagHelper.ViewModels
```

```
{
```

```
    public class DescriptionViewModel
```

```

{
    [MinLength(5)]
    [MaxLength(1024)]
    public string Description { get; set; }
}

```

C#HTML

```

@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>

```

The following HTML is generated:

HTML

```

<form method="post" action="/Demo/RegisterTextArea">
    <textarea data-val="true"
        data-val-maxlength="The field Description must be a string or array type
with a maximum length of &#x27;1024&#x27;."
        data-val-maxlength-max="1024"
        data-val-minlength="The field Description must be a string or array type
with a minimum length of &#x27;5&#x27;."
        data-val-minlength-min="5"
        id="Description" name="Description">
    </textarea>
    <button type="submit">Test</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>

```

The Label Tag Helper

- Generates the label caption and `for` attribute on a `<label>` [↗](#) element for an expression name
- HTML Helper alternative: `Html.LabelFor`.

The `Label Tag Helper` provides the following benefits over a pure HTML label element:

- You automatically get the descriptive label value from the `Display` attribute. The intended display name might change over time, and the combination of `Display` attribute and Label Tag Helper will apply the `Display` everywhere it's used.

- Less markup in source code
- Strong typing with the model property.

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

CSHTML

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

The following HTML is generated for the `<label>` element:

HTML

```
<label for="Email">Email Address</label>
```

The Label Tag Helper generated the `for` attribute value of "Email", which is the ID associated with the `<input>` element. The Tag Helpers generate consistent `id` and `for` elements so they can be correctly associated. The caption in this sample comes from the `Display` attribute. If the model didn't contain a `Display` attribute, the caption would be the property name of the expression. To override the default caption, add a caption inside the label tag.

The Validation Tag Helpers

There are two Validation Tag Helpers. The `Validation Message Tag Helper` (which displays a validation message for a single property on your model), and the `Validation Summary Tag Helper` (which displays a summary of validation errors). The `Input Tag Helper` adds HTML5 client side validation attributes to input elements based on data annotation attributes on your model classes. Validation is also performed on the server. The Validation Tag Helper displays these error messages when a validation error occurs.

The Validation Message Tag Helper

- Adds the [HTML5](#) `data-valmsg-for="property"` attribute to the [span](#) element, which attaches the validation error messages on the input field of the specified model property. When a client side validation error occurs, [jQuery](#) displays the error message in the `` element.
- Validation also takes place on the server. Clients may have JavaScript disabled and some validation can only be done on the server side.
- HTML Helper alternative: `Html.ValidationMessageFor`

The `Validation Message Tag Helper` is used with the `asp-validation-for` attribute on an HTML [span](#) element.

C#HTML

```
<span asp-validation-for="Email"></span>
```

The Validation Message Tag Helper will generate the following HTML:

HTML

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

You generally use the `Validation Message Tag Helper` after an `Input Tag Helper` for the same property. Doing so displays any validation error messages near the input that caused the error.

❗ Note

You must have a view with the correct JavaScript and [jQuery](#) script references in place for client side validation. See [Model Validation](#) for more information.

When a server side validation error occurs (for example when you have custom server side validation or client-side validation is disabled), MVC places that error message as the body of the `` element.

HTML

```
<span class="field-validation-error" data-valmsg-for="Email"
      data-valmsg-replace="true">
    The Email Address field is required.
</span>
```

The Validation Summary Tag Helper

- Targets `<div>` elements with the `asp-validation-summary` attribute
- HTML Helper alternative: `@Html.ValidationSummary`

The `Validation Summary Tag Helper` is used to display a summary of validation messages. The `asp-validation-summary` attribute value can be any of the following:

[Expand table](#)

<code>asp-validation-summary</code>	Validation messages displayed
<code>All</code>	Property and model level
<code>ModelOnly</code>	Model
<code>None</code>	None

Sample

In the following example, the data model has `DataAnnotation` attributes, which generates validation error messages on the `<input>` element. When a validation error occurs, the Validation Tag Helper displays the error message:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
```

```

    [EmailAddress]
    [Display(Name = "Email Address")]
    public string Email { get; set; }

    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }
}

```

CSHTML

```

@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    <label>Email: <input asp-for="Email" /></label> <br />
    <span asp-validation-for="Email"></span><br />
    <label>Password: <input asp-for="Password" /></label><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>

```

The generated HTML (when the model is valid):

HTML

```

<form action="/DemoReg/Register" method="post">
    <label>Email: <input name="Email" id="Email" type="email" value=""
        data-val-required="The Email field is required."
        data-val-email="The Email field is not a valid email address."
        data-val="true"></label><br>
    <span class="field-validation-valid" data-valmsg-replace="true"
        data-valmsg-for="Email"></span><br>
    <label>Password: <input name="Password" id="Password" type="password"
        data-val-required="The Password field is required." data-val="true">
    </label><br>
    <span class="field-validation-valid" data-valmsg-replace="true"
        data-valmsg-for="Password"></span><br>
    <button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for
    brevity>">
</form>

```

The Select Tag Helper

- Generates [select](#) and associated [option](#) elements for properties of your model.
- Has an HTML Helper alternative `Html.DropDownListFor` and `Html.ListBoxFor`

The `Select Tag Helper` `asp-for` specifies the model property name for the [select](#) element and `asp-items` specifies the [option](#) elements. For example:

C#HTML

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Sample:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new
List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA" },
        };
    }
}
```

The `Index` method initializes the `CountryViewModel`, sets the selected country and passes it to the `Index` view.

C#

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

The HTTP POST `Index` method displays the selection:

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
```

```

public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country + " selected";
        return RedirectToAction("IndexSuccess", new { message = msg });
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}

```

The `Index` view:

C#HTML

```

@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>

```

Which generates the following HTML (with "CA" selected):

HTML

```

<form method="post" action="/">
    <select id="Country" name="Country">
        <option value="MX">Mexico</option>
        <option selected="selected" value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>

```

⚠ Note

We don't recommend using `ViewBag` or `ViewData` with the Select Tag Helper. A view model is more robust at providing MVC metadata and generally less problematic.

The `asp-for` attribute value is a special case and doesn't require a `Model` prefix, the other Tag Helper attributes do (such as `asp-items`)

C#HTML

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Enum binding

It's often convenient to use `<select>` with an `enum` property and generate the `SelectListItem` elements from the `enum` values.

Sample:

C#

```
public class CountryEnumViewModel
{
    public CountryEnum EnumCountry { get; set; }
}
```

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The `GetEnumSelectList` method generates a `SelectList` object for an enum.

C#HTML

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
        asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
```

```
<br /><button type="submit">Register</button>
</form>
```

You can mark your enumerator list with the `Display` attribute to get a richer UI:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The following HTML is generated:

HTML

```
<form method="post" action="/Home/IndexEnum">
    <select data-val="true" data-val-required="The EnumCountry field is
    required."
        id="EnumCountry" name="EnumCountry">
        <option value="0">United Mexican States</option>
        <option value="1">United States of America</option>
        <option value="2">Canada</option>
        <option value="3">France</option>
        <option value="4">Germany</option>
        <option selected="selected" value="5">Spain</option>
    </select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
    for brevity>">
</form>
```

Option Group

The HTML `<optgroup>` [↗](#) element is generated when the view model contains one or more `SelectListGroup` objects.

The `CountryViewModelGroup` groups the `SelectListItem` elements into the "North America" and "Europe" groups:

C#

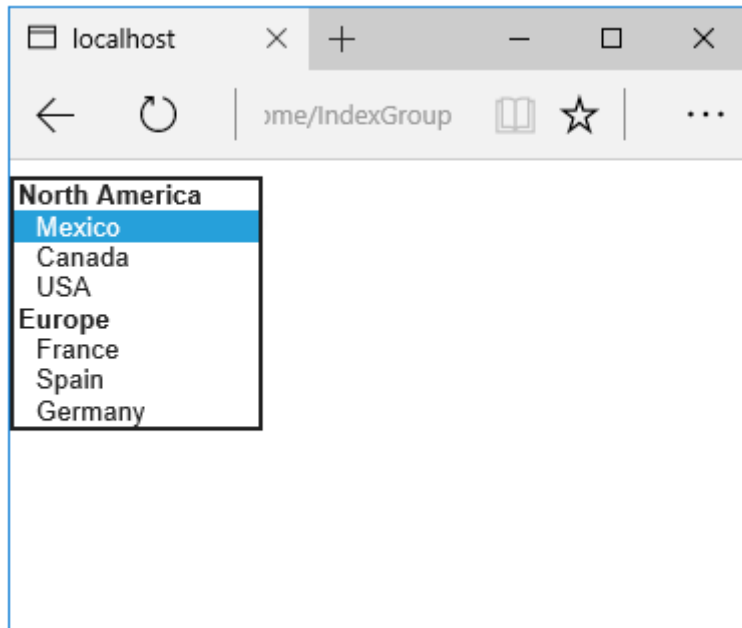
```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America" };
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
        };
    }
}
```

```
public string Country { get; set; }

public List<SelectListItem> Countries { get; }
```

The two groups are shown below:



The generated HTML:

HTML

```
<form method="post" action="/Home/IndexGroup">
  <select id="Country" name="Country">
    <optgroup label="North America">
      <option value="MEX">Mexico</option>
      <option value="CAN">Canada</option>
      <option value="US">USA</option>
    </optgroup>
    <optgroup label="Europe">
      <option value="FR">France</option>
      <option value="ES">Spain</option>
      <option value="DE">Germany</option>
    </optgroup>
  </select>
  <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

Multiple select

The Select Tag Helper will automatically generate the `multiple = "multiple"` attribute if the property specified in the `asp-for` attribute is an `IEnumerable`. For example, given the following model:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new
List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA" },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain" },
            new SelectListItem { Value = "DE", Text = "Germany" }
        };
    }
}
```

With the following view:

C#HTML

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Generates the following HTML:

HTML

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
        multiple="multiple"
        name="CountryCodes"><option value="MX">Mexico</option>
    <option value="CA">Canada</option>
    <option value="US">USA</option>
    <option value="FR">France</option>
```

```

<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>

```

No selection

If you find yourself using the "not specified" option in multiple pages, you can create a template to eliminate repeating the HTML:

CSSHTML

```

@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>

```

The `Views/Shared/EditorTemplates/CountryViewModel.csshtml` template:

CSSHTML

```

@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>

```

Adding HTML `<option>` [↗](#) elements isn't limited to the *No selection* case. For example, the following view and action method will generate HTML similar to the code above:

C#

```

public IActionResult IndexNone()
{
    var model = new CountryViewModel();
    model.Countries.Insert(0, new SelectListItem("<none>", ""));
    return View(model);
}

```

CSSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

The correct `<option>` element will be selected (contain the `selected="selected"` attribute) depending on the current `Country` value.

C#

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

HTML

```
<form method="post" action="/Home/IndexEmpty">
    <select id="Country" name="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA" selected="selected">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

Additional resources

- [Tag Helpers in ASP.NET Core](#)
- [HTML Form element](#) ↗
- [Request Verification Token](#)
- [Model Binding in ASP.NET Core](#)
- [Model validation in ASP.NET Core MVC](#)
- [IAttributeAdapter Interface](#)

- [Code snippets for this document](#) ↗

Tag Helpers in forms in ASP.NET Core

Article • 09/27/2024

By [Rick Anderson](#), [N. Taylor Mullen](#), [Dave Paquette](#), and [Jerrie Pelser](#)

This document demonstrates working with Forms and the HTML elements commonly used on a Form. The HTML [Form](#) element provides the primary mechanism web apps use to post back data to the server. Most of this document describes [Tag Helpers](#) and how they can help you productively create robust HTML forms. We recommend you read [Introduction to Tag Helpers](#) before you read this document.

In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers don't replace HTML Helpers and there's not a Tag Helper for each HTML Helper. When an HTML Helper alternative exists, it's mentioned.

The Form Tag Helper

The [Form Tag Helper](#):

- Generates the HTML [<FORM>](#) `action` attribute value for a MVC controller action or named route
- Generates a hidden [Request Verification Token](#) to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method)
- Provides the `asp-route-<Parameter Name>` attribute, where `<Parameter Name>` is added to the route values. The `routeValues` parameters to `Html.BeginForm` and `Html.BeginRouteForm` provide similar functionality.
- Has an HTML Helper alternative `Html.BeginForm` and `Html.BeginRouteForm`

Sample:

C#HTML

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

The Form Tag Helper above generates the following HTML:

HTML

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

The MVC runtime generates the `action` attribute value from the Form Tag Helper attributes `asp-controller` and `asp-action`. The Form Tag Helper also generates a hidden [Request Verification Token](#) to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method). Protecting a pure HTML Form from cross-site request forgery is difficult, the Form Tag Helper provides this service for you.

Using a named route

The `asp-route` Tag Helper attribute can also generate markup for the HTML `action` attribute. An app with a [route](#) named `register` could use the following markup for the registration page:

C#HTML

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Many of the views in the *Views/Account* folder (generated when you create a new web app with *Individual User Accounts*) contain the `asp-route-returnurl` attribute:

C#HTML

```
<form asp-controller="Account" asp-action="Login"
      asp-route-returnurl="@ViewData["ReturnUrl"]"
      method="post" class="form-horizontal" role="form">
```

ⓘ Note

With the built in templates, `returnUrl` is only populated automatically when you try to access an authorized resource but are not authenticated or authorized. When you attempt an unauthorized access, the security middleware redirects you to the login page with the `returnUrl` set.

The Form Action Tag Helper

The Form Action Tag Helper generates the `formaction` attribute on the generated `<button ...>` or `<input type="image" ...>` tag. The `formaction` attribute controls where a form submits its data. It binds to `<input>` [↗](#) elements of type `image` and `<button>` [↗](#) elements. The Form Action Tag Helper enables the usage of several [AnchorTagHelper](#) `asp-` attributes to control what `formaction` link is generated for the corresponding element.

Supported [AnchorTagHelper](#) attributes to control the value of `formaction`:

[↗](#) Expand table

Attribute	Description
asp-controller	The name of the controller.
asp-action	The name of the action method.
asp-area	The name of the area.
asp-page	The name of the Razor page.
asp-page-handler	The name of the Razor page handler.
asp-route	The name of the route.
asp-route-{value}	A single URL route value. For example, <code>asp-route-id="1234"</code> .
asp-all-route-data	All route values.
asp-fragment	The URL fragment.

Submit to controller example

The following markup submits the form to the `Index` action of `HomeController` when the input or button are selected:

C#HTML

```
<form method="post">
  <button asp-controller="Home" asp-action="Index">Click Me</button>
  <input type="image" src="..." alt="Or Click Me" asp-controller="Home"
        asp-action="Index">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
  <button formaction="/Home">Click Me</button>
  <input type="image" src="..." alt="Or Click Me" formaction="/Home">
</form>
```

Submit to page example

The following markup submits the form to the `About` Razor Page:

C#HTML

```
<form method="post">
  <button asp-page="About">Click Me</button>
  <input type="image" src="..." alt="Or Click Me" asp-page="About">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
  <button formaction="/About">Click Me</button>
  <input type="image" src="..." alt="Or Click Me" formaction="/About">
</form>
```

Submit to route example

Consider the `/Home/Test` endpoint:

C#

```
public class HomeController : Controller
{
    [Route("/Home/Test", Name = "Custom")]
    public string Test()
    {
        return "This is the test page";
    }
}
```

The following markup submits the form to the `/Home/Test` endpoint.

C#HTML

```
<form method="post">
    <button asp-route="Custom">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-route="Custom">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/Home/Test">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home/Test">
</form>
```

The Input Tag Helper

The Input Tag Helper binds an HTML `<input>` [↗](#) element to a model expression in your razor view.

Syntax:

C#HTML

```
<input asp-for="<Expression Name>">
```

The Input Tag Helper:

- Generates the `id` and `name` HTML attributes for the expression name specified in the `asp-for` attribute. `asp-for="Property1.Property2"` is equivalent to `m => m.Property1.Property2`. The name of the expression is what is used for the `asp-for` attribute value. See the [Expression names](#) section for additional information.
- Sets the HTML `type` attribute value based on the model type and [data annotation](#) attributes applied to the model property
- Won't overwrite the HTML `type` attribute value when one is specified
- Generates [HTML5](#) [↗](#) validation attributes from [data annotation](#) attributes applied to model properties
- Has an HTML Helper feature overlap with `Html.TextBoxFor` and `Html.EditorFor`. See the [HTML Helper alternatives to Input Tag Helper](#) section for details.

- Provides strong typing. If the name of the property changes and you don't update the Tag Helper you'll get an error similar to the following:

```
An error occurred during the compilation of a resource required to
process
this request. Please review the following specific error details and
modify
your source code appropriately.
```

```
Type expected
'RegisterViewModel' does not contain a definition for 'Email' and no
extension method 'Email' accepting a first argument of type
'RegisterViewModel'
could be found (are you missing a using directive or an assembly
reference?)
```

The `Input` Tag Helper sets the HTML `type` attribute based on the .NET type. The following table lists some common .NET types and generated HTML type (not every .NET type is listed).

[Expand table](#)

.NET type	Input Type
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime-local" ↗
Byte	type="number"
Int	type="number"
Single, Double	type="number"

The following table shows some common [data annotations](#) attributes that the input tag helper will map to specific input types (not every validation attribute is listed):

[Expand table](#)

Attribute	Input Type
[EmailAddress]	type="email"
[Url]	type="url"

Attribute	Input Type
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

CSHTML

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <button type="submit">Register</button>
</form>
```

The code above generates the following HTML:

HTML

```
<form method="post" action="/Demo/RegisterInput">
    Email:
    <input type="email" data-val="true"
```

```

        data-val-email="The Email Address field is not a valid email
address."
        data-val-required="The Email Address field is required."
        id="Email" name="Email" value=""><br>
    Password:
    <input type="password" data-val="true"
        data-val-required="The Password field is required."
        id="Password" name="Password"><br>
    <button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>

```

The data annotations applied to the `Email` and `Password` properties generate metadata on the model. The Input Tag Helper consumes the model metadata and produces [HTML5](#) `data-val-*` attributes (see [Model Validation](#)). These attributes describe the validators to attach to the input fields. This provides unobtrusive HTML5 and [jQuery](#) validation. The unobtrusive attributes have the format `data-val-rule="Error Message"`, where rule is the name of the validation rule (such as `data-val-required`, `data-val-email`, `data-val-maxlength`, etc.) If an error message is provided in the attribute, it's displayed as the value for the `data-val-rule` attribute. There are also attributes of the form `data-val-ruleName-argumentName="argumentValue"` that provide additional details about the rule, for example, `data-val-maxlength-max="1024"`.

When binding multiple `input` controls to the same property, the generated controls share the same `id`, which makes the generated mark-up invalid. To prevent duplicates, specify the `id` attribute for each control explicitly.

Checkbox hidden input rendering

Checkboxes in HTML5 don't submit a value when they're unchecked. To enable a default value to be sent for an unchecked checkbox, the Input Tag Helper generates an additional hidden input for checkboxes.

For example, consider the following Razor markup that uses the Input Tag Helper for a boolean model property `IsChecked`:

C#HTML

```

<form method="post">
    <input asp-for="@Model.IsChecked" />
    <button type="submit">Submit</button>
</form>

```

The preceding Razor markup generates HTML markup similar to the following:

HTML

```
<form method="post">
  <input name="IsChecked" type="checkbox" value="true" />
  <button type="submit">Submit</button>

  <input name="IsChecked" type="hidden" value="false" />
</form>
```

The preceding HTML markup shows an additional hidden input with a name of `IsChecked` and a value of `false`. By default, this hidden input is rendered at the end of the form. When the form is submitted:

- If the `IsChecked` checkbox input is checked, both `true` and `false` are submitted as values.
- If the `IsChecked` checkbox input is unchecked, only the hidden input value `false` is submitted.

The ASP.NET Core model-binding process reads only the first value when binding to a `bool` value, which results in `true` for checked checkboxes and `false` for unchecked checkboxes.

To configure the behavior of the hidden input rendering, set the [CheckBoxHiddenInputRenderMode](#) property on [MvcViewOptions.HtmlHelperOptions](#). For example:

C#

```
services.Configure<MvcViewOptions>(options =>
    options.HtmlHelperOptions.CheckBoxHiddenInputRenderMode =
        CheckBoxHiddenInputRenderMode.None);
```

The preceding code disables hidden input rendering for checkboxes by setting `CheckBoxHiddenInputRenderMode` to [CheckBoxHiddenInputRenderMode.None](#). For all available rendering modes, see the [CheckBoxHiddenInputRenderMode](#) enum.

HTML Helper alternatives to Input Tag Helper

`Html.TextBox`, `Html.TextBoxFor`, `Html.Editor` and `Html.EditorFor` have overlapping features with the Input Tag Helper. The Input Tag Helper will automatically set the `type` attribute; `Html.TextBox` and `Html.TextBoxFor` won't. `Html.Editor` and `Html.EditorFor` handle collections, complex objects and templates; the Input Tag Helper doesn't. The

Input Tag Helper, `Html.EditorFor` and `Html.TextBoxFor` are strongly typed (they use lambda expressions); `Html.TextBox` and `Html.Editor` are not (they use expression names).

HtmlAttributes

`@Html.Editor()` and `@Html.EditorFor()` use a special `ViewDataDictionary` entry named `htmlAttributes` when executing their default templates. This behavior is optionally augmented using `additionalViewData` parameters. The key "htmlAttributes" is case-insensitive. The key "htmlAttributes" is handled similarly to the `htmlAttributes` object passed to input helpers like `@Html.TextBox()`.

C#HTML

```
@Html.EditorFor(model => model.YourProperty,  
    new { htmlAttributes = new { @class="myCssClass", style="width:100px" } })
```

Expression names

The `asp-for` attribute value is a `ModelExpression` and the right hand side of a lambda expression. Therefore, `asp-for="Property1"` becomes `m => m.Property1` in the generated code which is why you don't need to prefix with `Model`. You can use the "@" character to start an inline expression and move before the `m.`:

C#HTML

```
@{  
    var joe = "Joe";  
}  
  
<input asp-for="@joe">
```

Generates the following:

HTML

```
<input type="text" id="joe" name="joe" value="Joe">
```

With collection properties, `asp-for="CollectionProperty[23].Member"` generates the same name as `asp-for="CollectionProperty[i].Member"` when `i` has the value `23`.

When ASP.NET Core MVC calculates the value of `ModelExpression`, it inspects several sources, including `ModelState`. Consider `<input type="text" asp-for="Name">`. The calculated `value` attribute is the first non-null value from:

- `ModelState` entry with key "Name".
- Result of the expression `Model.Name`.

Navigating child properties

You can also navigate to child properties using the property path of the view model. Consider a more complex model class that contains a child `Address` property.

C#

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

C#

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

In the view, we bind to `Address.AddressLine1`:

CSHTML

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <label>Address: <input asp-for="Address.AddressLine1" /></label><br />
    <button type="submit">Register</button>
</form>
```

The following HTML is generated for `Address.AddressLine1`:

HTML

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1" value="">
```

Expression names and Collections

Sample, a model containing an array of `Colors`:

C#

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

The action method:

C#

```
public IActionResult Edit(int id, int colorIndex)
{
    ViewData["Index"] = colorIndex;
    return View(GetPerson(id));
}
```

The following Razor shows how you access a specific `Color` element:

CSHTML

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

The `Views/Shared/EditorTemplates/String.cshtml` template:

CSHTML


```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

Sample using `List<T>`:

C#

```
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

The following Razor shows how to iterate over a collection:

CSSHTML

```
@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

The `Views/Shared/EditorTemplates/ToDoItem.cshtml` template:

CSSHTML

```
@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
```

```
</td>
```

```
@*
```

```
    This template replaces the following Razor which evaluates the indexer  
    three times.
```

```
    <td>
```

```
        <label asp-for="@Model[i].Name"></label>
```

```
        @Html.DisplayFor(model => model[i].Name)
```

```
    </td>
```

```
    <td>
```

```
        <input asp-for="@Model[i].IsDone" />
```

```
    </td>
```

```
*@
```

`foreach` should be used if possible when the value is going to be used in an `asp-for` or `Html.DisplayFor` equivalent context. In general, `for` is better than `foreach` (if the scenario allows it) because it doesn't need to allocate an enumerator; however, evaluating an indexer in a LINQ expression can be expensive and should be minimized.

❗ Note

The commented sample code above shows how you would replace the lambda expression with the `@` operator to access each `ToDoItem` in the list.

The Textarea Tag Helper

The `Textarea Tag Helper` tag helper is similar to the `Input Tag Helper`.

- Generates the `id` and `name` attributes, and the data validation attributes from the model for a `<textarea>` [↗](#) element.
- Provides strong typing.
- HTML Helper alternative: `Html.TextAreaFor`

Sample:

```
C#
```

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

```
namespace FormsTagHelper.ViewModels
```

```
{
```

```
    public class DescriptionViewModel
```

```

{
    [MinLength(5)]
    [MaxLength(1024)]
    public string Description { get; set; }
}

```

C#HTML

```

@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>

```

The following HTML is generated:

HTML

```

<form method="post" action="/Demo/RegisterTextArea">
    <textarea data-val="true"
        data-val-maxlength="The field Description must be a string or array type
with a maximum length of &#x27;1024&#x27;."
        data-val-maxlength-max="1024"
        data-val-minlength="The field Description must be a string or array type
with a minimum length of &#x27;5&#x27;."
        data-val-minlength-min="5"
        id="Description" name="Description">
    </textarea>
    <button type="submit">Test</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>

```

The Label Tag Helper

- Generates the label caption and `for` attribute on a `<label>` [↗](#) element for an expression name
- HTML Helper alternative: `Html.LabelFor`.

The `Label Tag Helper` provides the following benefits over a pure HTML label element:

- You automatically get the descriptive label value from the `Display` attribute. The intended display name might change over time, and the combination of `Display` attribute and Label Tag Helper will apply the `Display` everywhere it's used.

- Less markup in source code
- Strong typing with the model property.

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

CSHTML

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

The following HTML is generated for the `<label>` element:

HTML

```
<label for="Email">Email Address</label>
```

The Label Tag Helper generated the `for` attribute value of "Email", which is the ID associated with the `<input>` element. The Tag Helpers generate consistent `id` and `for` elements so they can be correctly associated. The caption in this sample comes from the `Display` attribute. If the model didn't contain a `Display` attribute, the caption would be the property name of the expression. To override the default caption, add a caption inside the label tag.

The Validation Tag Helpers

There are two Validation Tag Helpers. The `Validation Message Tag Helper` (which displays a validation message for a single property on your model), and the `Validation Summary Tag Helper` (which displays a summary of validation errors). The `Input Tag Helper` adds HTML5 client side validation attributes to input elements based on data annotation attributes on your model classes. Validation is also performed on the server. The Validation Tag Helper displays these error messages when a validation error occurs.

The Validation Message Tag Helper

- Adds the [HTML5](#) `data-valmsg-for="property"` attribute to the [span](#) element, which attaches the validation error messages on the input field of the specified model property. When a client side validation error occurs, [jQuery](#) displays the error message in the `` element.
- Validation also takes place on the server. Clients may have JavaScript disabled and some validation can only be done on the server side.
- HTML Helper alternative: `Html.ValidationMessageFor`

The `Validation Message Tag Helper` is used with the `asp-validation-for` attribute on an HTML [span](#) element.

C#HTML

```
<span asp-validation-for="Email"></span>
```

The Validation Message Tag Helper will generate the following HTML:

HTML

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

You generally use the `Validation Message Tag Helper` after an `Input Tag Helper` for the same property. Doing so displays any validation error messages near the input that caused the error.

❗ Note

You must have a view with the correct JavaScript and [jQuery](#) script references in place for client side validation. See [Model Validation](#) for more information.

When a server side validation error occurs (for example when you have custom server side validation or client-side validation is disabled), MVC places that error message as the body of the `` element.

HTML

```
<span class="field-validation-error" data-valmsg-for="Email"
      data-valmsg-replace="true">
    The Email Address field is required.
</span>
```

The Validation Summary Tag Helper

- Targets `<div>` elements with the `asp-validation-summary` attribute
- HTML Helper alternative: `@Html.ValidationSummary`

The `Validation Summary Tag Helper` is used to display a summary of validation messages. The `asp-validation-summary` attribute value can be any of the following:

[Expand table](#)

<code>asp-validation-summary</code>	Validation messages displayed
<code>All</code>	Property and model level
<code>ModelOnly</code>	Model
<code>None</code>	None

Sample

In the following example, the data model has `DataAnnotation` attributes, which generates validation error messages on the `<input>` element. When a validation error occurs, the Validation Tag Helper displays the error message:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
```

```

    [EmailAddress]
    [Display(Name = "Email Address")]
    public string Email { get; set; }

    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }
}

```

CSHTML

```

@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    <label>Email: <input asp-for="Email" /></label> <br />
    <span asp-validation-for="Email"></span><br />
    <label>Password: <input asp-for="Password" /></label><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>

```

The generated HTML (when the model is valid):

HTML

```

<form action="/DemoReg/Register" method="post">
    <label>Email: <input name="Email" id="Email" type="email" value=""
        data-val-required="The Email field is required."
        data-val-email="The Email field is not a valid email address."
        data-val="true"></label><br>
    <span class="field-validation-valid" data-valmsg-replace="true"
        data-valmsg-for="Email"></span><br>
    <label>Password: <input name="Password" id="Password" type="password"
        data-val-required="The Password field is required." data-val="true">
</label><br>
    <span class="field-validation-valid" data-valmsg-replace="true"
        data-valmsg-for="Password"></span><br>
    <button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>

```

The Select Tag Helper

- Generates [select](#) and associated [option](#) elements for properties of your model.
- Has an HTML Helper alternative `Html.DropDownListFor` and `Html.ListBoxFor`

The `Select Tag Helper` `asp-for` specifies the model property name for the [select](#) element and `asp-items` specifies the [option](#) elements. For example:

C#HTML

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Sample:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new
List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA" },
        };
    }
}
```

The `Index` method initializes the `CountryViewModel`, sets the selected country and passes it to the `Index` view.

C#

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

The HTTP POST `Index` method displays the selection:

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
```