

```

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#

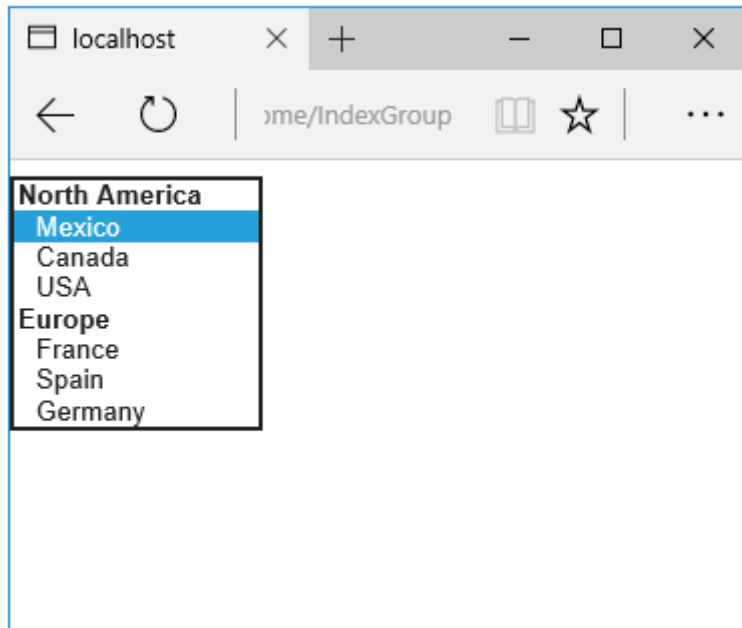
```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America" };
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

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

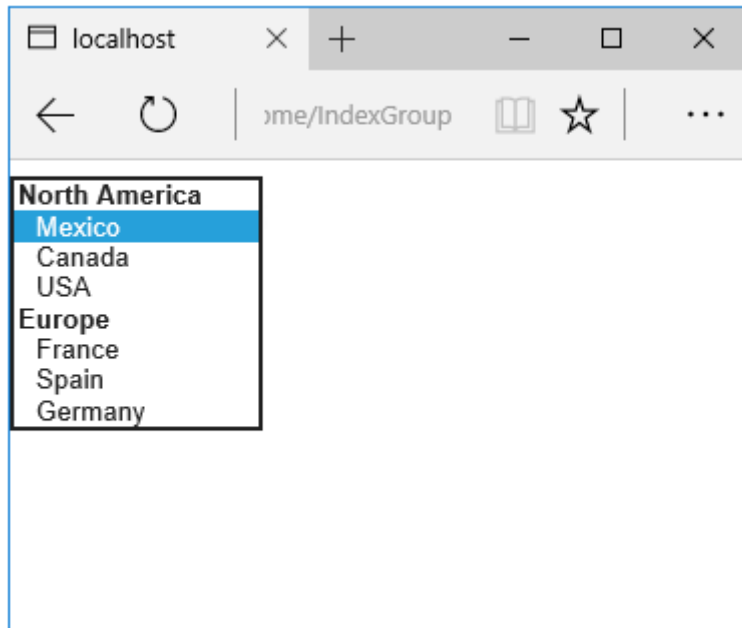
```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America" };
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

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

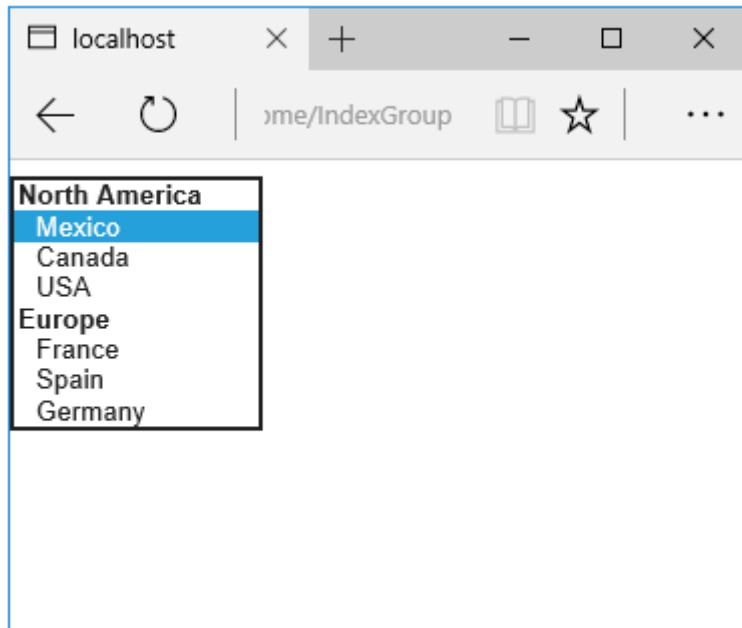
```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America" };
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        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](#) ↗

# Share controllers, views, Razor Pages and more with Application Parts

Article • 06/03/2022

By [Rick Anderson](#) 

[View or download sample code](#)  ([how to download](#))

An *Application Part* is an abstraction over the resources of an app. Application Parts allow ASP.NET Core to discover controllers, view components, tag helpers, Razor Pages, razor compilation sources, and more. [AssemblyPart](#) is an Application part. `AssemblyPart` encapsulates an assembly reference and exposes types and compilation references.

[Feature providers](#) work with application parts to populate the features of an ASP.NET Core app. The main use case for application parts is to configure an app to discover (or avoid loading) ASP.NET Core features from an assembly. For example, you might want to share common functionality between multiple apps. Using Application Parts, you can share an assembly (DLL) containing controllers, views, Razor Pages, razor compilation sources, Tag Helpers, and more with multiple apps. Sharing an assembly is preferred to duplicating code in multiple projects.

ASP.NET Core apps load features from [ApplicationPart](#). The [AssemblyPart](#) class represents an application part that's backed by an assembly.

## Load ASP.NET Core features

Use the [Microsoft.AspNetCore.Mvc.ApplicationParts](#) and [AssemblyPart](#) classes to discover and load ASP.NET Core features (controllers, view components, etc.). The [ApplicationPartManager](#) tracks the application parts and feature providers available.

`ApplicationPartManager` is configured in `Startup.ConfigureServices`:

C#

```
// Requires using System.Reflection;
public void ConfigureServices(IServiceCollection services)
{
    var assembly = typeof(MySharedController).Assembly;
    services.AddControllersWithViews()
        .AddApplicationPart(assembly)
        .AddRazorRuntimeCompilation();

    services.Configure<MvcRazorRuntimeCompilationOptions>(options =>
```

```
{ options.FileProviders.Add(new EmbeddedFileProvider(assembly)); });  
}
```

The following code provides an alternative approach to configuring `ApplicationPartManager` using `AssemblyPart`:

C#

```
// Requires using System.Reflection;  
// Requires using Microsoft.AspNetCore.Mvc.ApplicationParts;  
public void ConfigureServices(IServiceCollection services)  
{  
    var assembly = typeof(MySharedController).Assembly;  
    // This creates an AssemblyPart, but does not create any related parts  
    // for items such as views.  
    var part = new AssemblyPart(assembly);  
    services.AddControllersWithViews()  
        .ConfigureApplicationPartManager(apm =>  
            apm.ApplicationParts.Add(part));  
}
```

The preceding two code samples load the `SharedController` from an assembly. The `SharedController` is not in the app's project. See the [WebAppParts solution](#) sample download.

## Include views

Use a [Razor class library](#) to include views in the assembly.

## Prevent loading resources

Application parts can be used to *avoid* loading resources in a particular assembly or location. Add or remove members of the `Microsoft.AspNetCore.Mvc.ApplicationParts` collection to hide or make available resources. The order of the entries in the `ApplicationParts` collection isn't important. Configure the `ApplicationPartManager` before using it to configure services in the container. For example, configure the `ApplicationPartManager` before invoking `AddControllersAsServices`. Call `Remove` on the `ApplicationParts` collection to remove a resource.

The `ApplicationPartManager` includes parts for:

- The app's assembly and dependent assemblies.
- `Microsoft.AspNetCore.Mvc.ApplicationParts.CompiledRazorAssemblyPart`
- `Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation`

- `Microsoft.AspNetCore.Mvc.TagHelpers`.
- `Microsoft.AspNetCore.Mvc.Razor`.

## Feature providers

Application feature providers examine application parts and provide features for those parts. There are built-in feature providers for the following ASP.NET Core features:

- [ControllerFeatureProvider](#)
- [TagHelperFeatureProvider](#)
- [MetadataReferenceFeatureProvider](#)
- [ViewsFeatureProvider](#)
- `internal class` [RazorCompiledItemFeatureProvider](#) [↗](#)

Feature providers inherit from [IApplicationFeatureProvider<TFeature>](#), where `T` is the type of the feature. Feature providers can be implemented for any of the previously listed feature types. The order of feature providers in the `ApplicationPartManager.FeatureProviders` can impact run time behavior. Later added providers can react to actions taken by earlier added providers.

## Display available features

The features available to an app can be enumerated by requesting an `ApplicationPartManager` through [dependency injection](#):

C#

```
using AppPartsSample.ViewModels;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.ApplicationParts;
using Microsoft.AspNetCore.Mvc.Controllers;
using System.Linq;
using Microsoft.AspNetCore.Mvc.Razor.Compilation;
using Microsoft.AspNetCore.Mvc.Razor.TagHelpers;
using Microsoft.AspNetCore.Mvc.ViewComponents;

namespace AppPartsSample.Controllers
{
    public class FeaturesController : Controller
    {
        private readonly ApplicationPartManager _partManager;

        public FeaturesController(ApplicationPartManager partManager)
        {
            _partManager = partManager;
        }
    }
}
```

```

public IActionResult Index()
{
    var viewModel = new FeaturesViewModel();

    var controllerFeature = new ControllerFeature();
    _partManager.PopulateFeature(controllerFeature);
    viewModel.Controllers = controllerFeature.Controllers.ToList();

    var tagHelperFeature = new TagHelperFeature();
    _partManager.PopulateFeature(tagHelperFeature);
    viewModel.TagHelpers = tagHelperFeature.TagHelpers.ToList();

    var viewComponentFeature = new ViewComponentFeature();
    _partManager.PopulateFeature(viewComponentFeature);
    viewModel.ViewComponents =
viewComponentFeature.ViewComponents.ToList();

    return View(viewModel);
}
}
}

```

The [download sample](#) uses the preceding code to display the app features:

text

Controllers:

- FeaturesController
- HomeController
- HelloController
- GenericController`1
- GenericController`1

Tag Helpers:

- PrerenderTagHelper
- AnchorTagHelper
- CacheTagHelper
- DistributedCacheTagHelper
- EnvironmentTagHelper
- Additional Tag Helpers omitted for brevity.

View Components:

- SampleViewComponent

## Discovery in application parts

HTTP 404 errors are not uncommon when developing with application parts. These errors are typically caused by missing an essential requirement for how applications parts are discovered. If your app returns an HTTP 404 error, verify the following requirements have been met:

- The `applicationName` setting needs to be set to the root assembly used for discovery. The root assembly used for discovery is normally the entry point assembly.
- The root assembly needs to have a reference to the parts used for discovery. The reference can be direct or transitive.
- The root assembly needs to reference the Web SDK. The framework has logic that stamps attributes into the root assembly that are used for discovery.



# Work with the application model in ASP.NET Core

Article • 06/17/2024

By [Steve Smith](#) 

ASP.NET Core MVC defines an *application model* representing the components of an MVC app. Read and manipulate this model to modify how MVC elements behave. By default, MVC follows certain conventions to determine which classes are considered controllers, which methods on those classes are actions, and how parameters and routing behave. Customize this behavior to suit an app's needs by creating custom conventions and applying them globally or as attributes.


## Models and Providers ( `IApplicationModelProvider` )

The ASP.NET Core MVC application model includes both abstract interfaces and concrete implementation classes that describe an MVC application. This model is the result of MVC discovering the app's controllers, actions, action parameters, routes, and filters according to default conventions. By working with the application model, modify an app to follow different conventions from the default MVC behavior. The parameters, names, routes, and filters are all used as configuration data for actions and controllers.

The ASP.NET Core MVC Application Model has the following structure:

- ApplicationModel
  - Controllers (ControllerModel)
  - Actions (ActionModel)
    - Parameters (ParameterModel)

Each level of the model has access to a common `Properties` collection, and lower levels can access and overwrite property values set by higher levels in the hierarchy. The properties are persisted to the `ActionDescriptor.Properties` when the actions are created. Then when a request is being handled, any properties a convention added or modified can be accessed through `ActionContext.ActionDescriptor`. Using properties is a great way to configure filters, model binders, and other app model aspects on a per-action basis.

 **Note**

The [ActionDescriptor.Properties](#) collection isn't thread safe (for writes) after app startup. Conventions are the best way to safely add data to this collection.

ASP.NET Core MVC loads the application model using a provider pattern, defined by the [IApplicationModelProvider](#) interface. This section covers some of the internal implementation details of how this provider functions. Use of the provider pattern is an advanced subject, primarily for framework use. Most apps should use conventions, not the provider pattern.

Implementations of the [IApplicationModelProvider](#) interface "wrap" one another, where each implementation calls [OnProvidersExecuting](#) in ascending order based on its [Order](#) property. The [OnProvidersExecuted](#) method is then called in reverse order. The framework defines several providers:

First (`Order=-1000`):

- `DefaultApplicationModelProvider`

Then (`Order=-990`):

- `AuthorizationApplicationModelProvider`
- `CorsApplicationModelProvider`

#### ⓘ Note

The order in which two providers with the same value for `order` are called is undefined and shouldn't be relied upon.

#### ⓘ Note

[IApplicationModelProvider](#) is an advanced concept for framework authors to extend. In general, apps should use conventions, and frameworks should use providers. The key distinction is that providers always run before conventions.

The `DefaultApplicationModelProvider` establishes many of the default behaviors used by ASP.NET Core MVC. Its responsibilities include:

- Adding global filters to the context
- Adding controllers to the context
- Adding public controller methods as actions
- Adding action method parameters to the context

- Applying route and other attributes

Some built-in behaviors are implemented by the `DefaultApplicationModelProvider`. This provider is responsible for constructing the `ControllerModel`, which in turn references `ActionModel`, `PropertyModel`, and `ParameterModel` instances. The `DefaultApplicationModelProvider` class is an internal framework implementation detail that may change in the future.

The `AuthorizationApplicationModelProvider` is responsible for applying the behavior associated with the `AuthorizeFilter` and `AllowAnonymousFilter` attributes. For more information, see [Simple authorization in ASP.NET Core](#).

The `CorsApplicationModelProvider` implements behavior associated with `EnableCorsAttribute` and `IDisableCorsAttribute`. For more information, see [Enable Cross-Origin Requests \(CORS\) in ASP.NET Core](#).

Information on the framework's internal providers described in this section aren't available via the [.NET API browser](#). However, the providers may be inspected in the [ASP.NET Core reference source \(dotnet/aspnetcore GitHub repository\)](#) [↗](#). Use GitHub search to find the providers by name and select the version of the source with the **Switch branches/tags** dropdown list.

## Conventions

The application model defines convention abstractions that provide a simpler way to customize the behavior of the models than overriding the entire model or provider. These abstractions are the recommended way to modify an app's behavior. Conventions provide a way to write code that dynamically applies customizations. While [filters](#) provide a means of modifying the framework's behavior, customizations permit control over how the whole app works together.

The following conventions are available:

- [IApplicationModelConvention](#)
- [IControllerModelConvention](#)
- [IActionModelConvention](#)
- [IParameterModelConvention](#)

Conventions are applied by adding them to MVC options or by implementing attributes and applying them to controllers, actions, or action parameters (similar to [filters](#)). Unlike filters, conventions are only executed when the app is starting, not as part of each request.

### ⓘ Note

For information on Razor Pages route and application model provider conventions, see [Razor Pages route and app conventions in ASP.NET Core](#).

## Modify the `ApplicationModel`

The following convention is used to add a property to the application model:

```
C#

using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class ApplicationDescription : IApplicationModelConvention
    {
        private readonly string _description;

        public ApplicationDescription(string description)
        {
            _description = description;
        }

        public void Apply(ApplicationModel application)
        {
            application.Properties["description"] = _description;
        }
    }
}
```

Application model conventions are applied as options when MVC is added in `Startup.ConfigureServices`:

```
C#

public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.Conventions.Add(new ApplicationDescription("My Application Description"));
        options.Conventions.Add(new NamespaceRoutingConvention());
    });
}
```

Properties are accessible from the [ActionDescriptor.Properties](#) collection within controller actions:

C#

```
public class AppModelController : Controller
{
    public string Description()
    {
        return "Description: " +
ControllerContext.ActionDescriptor.Properties["description"];
    }
}
```

## Modify the **ControllerModel** description

The controller model can also include custom properties. Custom properties override existing properties with the same name specified in the application model. The following convention attribute adds a description at the controller level:

C#

```
using System;
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class ControllerDescriptionAttribute : Attribute,
IControllerModelConvention
    {
        private readonly string _description;

        public ControllerDescriptionAttribute(string description)
        {
            _description = description;
        }

        public void Apply(ControllerModel controllerModel)
        {
            controllerModel.Properties["description"] = _description;
        }
    }
}
```

This convention is applied as an attribute on a controller:

C#

```
[ControllerDescription("Controller Description")]
public class DescriptionAttributesController : Controller
{
    public string Index()
    {
        return "Description: " +
ControllerContext.ActionDescriptor.Properties["description"];
    }
}
```

## Modify the **ActionModel** description

A separate attribute convention can be applied to individual actions, overriding behavior already applied at the application or controller level:

```
C#

using System;
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class ActionDescriptionAttribute : Attribute,
IActionModelConvention
    {
        private readonly string _description;

        public ActionDescriptionAttribute(string description)
        {
            _description = description;
        }

        public void Apply(ActionModel actionModel)
        {
            actionModel.Properties["description"] = _description;
        }
    }
}
```

Applying this to an action within the controller demonstrates how it overrides the controller-level convention:

```
C#

[ControllerDescription("Controller Description")]
public class DescriptionAttributesController : Controller
{
    public string Index()
    {
```

```

        return "Description: " +
ControllerContext.ActionDescriptor.Properties["description"];
    }

    [ActionDescription("Action Description")]
    public string UseActionDescriptionAttribute()
    {
        return "Description: " +
ControllerContext.ActionDescriptor.Properties["description"];
    }
}

```

## Modify the ParameterModel

The following convention can be applied to action parameters to modify their [BindingInfo](#). The following convention requires that the parameter be a route parameter. Other potential binding sources, such as query string values, are ignored:

C#

```

using System;
using Microsoft.AspNetCore.Mvc.ApplicationModels;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace AppModelSample.Conventions
{
    public class MustBeInRouteParameterModelConvention : Attribute,
IParacterModelConvention
    {
        public void Apply(ParameterModel model)
        {
            if (model.BindingInfo == null)
            {
                model.BindingInfo = new BindingInfo();
            }
            model.BindingInfo.BindingSource = BindingSource.Path;
        }
    }
}

```

The attribute may be applied to any action parameter:

C#

```

public class ParameterModelController : Controller
{
    // Will bind: /ParameterModel/GetById/123
    // WON'T bind: /ParameterModel/GetById?id=123
    public string GetById([MustBeInRouteParameterModelConvention]int id)

```

```

    {
        return $"Bound to id: {id}";
    }
}

```

To apply the convention to all action parameters, add the

`MustBeInRouteParameterModelConvention` to `MvcOptions` in `Startup.ConfigureServices`:

C#

```
options.Conventions.Add(new MustBeInRouteParameterModelConvention());
```

## Modify the `ActionModel` name

The following convention modifies the `ActionModel` to update the *name* of the action to which it's applied. The new name is provided as a parameter to the attribute. This new name is used by routing, so it affects the route used to reach this action method:

C#

```

using System;
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class CustomActionNameAttribute : Attribute,
        IActionModelConvention
    {
        private readonly string _actionName;

        public CustomActionNameAttribute(string actionName)
        {
            _actionName = actionName;
        }

        public void Apply(ActionModel actionModel)
        {
            // this name will be used by routing
            actionModel.ActionName = _actionName;
        }
    }
}

```

This attribute is applied to an action method in the `HomeController`:

C#



```
// Route: /Home/MyCoolAction
[CustomActionName("MyCoolAction")]
public string SomeName()
{
    return ControllerContext.ActionDescriptor.ActionName;
}
```

Even though the method name is `SomeName`, the attribute overrides the MVC convention of using the method name and replaces the action name with `MyCoolAction`. Thus, the route used to reach this action is `/Home/MyCoolAction`.

### ❗ Note

This example in this section is essentially the same as using the built-in [ActionNameAttribute](#).

## Custom routing convention

Use an [IApplicationModelConvention](#) to customize how routing works. For example, the following convention incorporates controllers' namespaces into their routes, replacing `.` in the namespace with `/` in the route:

```
C#

using Microsoft.AspNetCore.Mvc.ApplicationModels;
using System.Linq;

namespace AppModelSample.Conventions
{
    public class NamespaceRoutingConvention : IApplicationModelConvention
    {
        public void Apply(ApplicationModel application)
        {
            foreach (var controller in application.Controllers)
            {
                var hasAttributeRouteModels = controller.Selectors
                    .Any(selector => selector.AttributeRouteModel != null);

                if (!hasAttributeRouteModels
                    && controller.ControllerName.Contains("Namespace")) //
                // affect one controller in this sample
                {
                    // Replace the . in the namespace with a / to create the
                    // attribute route
                    // Ex: MySite.Admin namespace will correspond to
                    // MySite/Admin attribute route
                }
            }
        }
    }
}
```

```

        // Then attach [controller], [action] and optional {id?}
        token.

        // [Controller] and [action] is replaced with the
        controller and action
        // name to generate the final template
        controller.Selectors[0].AttributeRouteModel = new
AttributeRouteModel()
        {
            Template =
controller.ControllerType.Namespace.Replace('.', '/') +
"/[controller]/[action]/{id?}"
        };
    }
}

// You can continue to put attribute route templates for the
controller actions depending on the way you want them to behave
}
}
}

```

The convention is added as an option in `Startup.ConfigureServices`:

C#

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.Conventions.Add(new ApplicationDescription("My Application
Description"));
        options.Conventions.Add(new NamespaceRoutingConvention());
    });
}

```

### Tip

Add conventions to middleware via MvcOptions using the following approach. The `{CONVENTION}` placeholder is the convention to add:

C#

```

services.Configure<MvcOptions>(c => c.Conventions.Add({CONVENTION}));

```

The following example applies a convention to routes that aren't using attribute routing where the controller has `Namespace` in its name:

C#

```
using Microsoft.AspNetCore.Mvc;

namespace AppModelSample.Controllers
{
    public class NamespaceRoutingController : Controller
    {
        // using NamespaceRoutingConvention
        // route: /AppModelSample/Controllers/NamespaceRouting/Index
        public string Index()
        {
            return "This demonstrates namespace routing.";
        }
    }
}
```

## Use **ApiExplorer** to document an app

The application model exposes an [ApiExplorerModel](#) property at each level that can be used to traverse the app's structure. This can be used to [generate help pages for web APIs using tools like Swagger](#). The **ApiExplorer** property exposes an [IsVisible](#) property that can be set to specify which parts of the app's model should be exposed. Configure this setting using a convention:

C#

```
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class EnableApiExplorerApplicationConvention :
        IApplicationModelConvention
    {
        public void Apply(ApplicationModel application)
        {
            application.ApiExplorer.IsVisible = true;
        }
    }
}
```

Using this approach (and additional conventions if required), API visibility is enabled or disabled at any level within an app.

# Areas in ASP.NET Core

Article • 06/17/2024

By [Dhananjay Kumar](#) and [Rick Anderson](#)

Areas are an ASP.NET feature used to organize related functionality into a group as a separate:

- Namespace for routing.
- Folder structure for views and Razor Pages.

Using areas creates a hierarchy for the purpose of routing by adding another route parameter, `area`, to `controller` and `action` or a Razor Page `page`.

Areas provide a way to partition an ASP.NET Core Web app into smaller functional groups, each with its own set of Razor Pages, controllers, views, and models. An area is effectively a structure inside an app. In an ASP.NET Core web project, logical components like Pages, Model, Controller, and View are kept in different folders. The ASP.NET Core runtime uses naming conventions to create the relationship between these components. For a large app, it may be advantageous to partition the app into separate high level areas of functionality. For instance, an e-commerce app with multiple business units, such as checkout, billing, and search. Each of these units have their own area to contain views, controllers, Razor Pages, and models.

Consider using Areas in a project when:

- The app is made of multiple high-level functional components that can be logically separated.
- You want to partition the app so that each functional area can be worked on independently.

If you're using Razor Pages, see [Areas with Razor Pages](#) in this document.

## Areas for controllers with views

A typical ASP.NET Core web app using areas, controllers, and views contains the following:

- An [Area folder structure](#).
- Controllers with the `[Area]` attribute to associate the controller with the area:

C#

```
[Area("Products")]
public class ManageController : Controller
{
```

- The [area route](#) added to `Program.cs`:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(
    name: "MyArea",
    pattern: "{area:exists}/{controller=Home}/{action=Index}/{id?}");

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

app.Run();
```

## Area folder structure

Consider an app that has two logical groups, *Products* and *Services*. Using areas, the folder structure would be similar to the following:

- Project name
  - Areas
    - Products
      - Controllers
        - HomeController.cs
        - ManageController.cs

- Views
  - Home
    - Index.cshtml
  - Manage
    - Index.cshtml
    - About.cshtml
- Services
  - Controllers
    - HomeController.cs
  - Views
    - Home
      - Index.cshtml

While the preceding layout is typical when using Areas, only the view files are required to use this folder structure. View discovery searches for a matching area view file in the following order:

text

```
/Areas/<Area-Name>/Views/<Controller-Name>/<Action-Name>.cshtml  
/Areas/<Area-Name>/Views/Shared/<Action-Name>.cshtml  
/Views/Shared/<Action-Name>.cshtml  
/Pages/Shared/<Action-Name>.cshtml
```

## Associate the controller with an Area

Area controllers are designated with the [\[Area\]](#) attribute:

C#

```
using Microsoft.AspNetCore.Mvc;  
using Microsoft.Docs.Samples;  
  
namespace MVCareas.Areas.Products.Controllers;  
  
[Area("Products")]  
public class ManageController : Controller  
{  
    public IActionResult Index()  
    {  
        ViewData["routeInfo"] = ControllerContext.MyDisplayRouteInfo();  
        return View();  
    }  
  
    public IActionResult About()  
    {
```

```
        ViewData["routeInfo"] = ControllerContext.MyDisplayRouteInfo();  
        return View();  
    }  
}
```

## Add Area route

Area routes typically use [conventional routing](#) rather than [attribute routing](#).

Conventional routing is order-dependent. In general, routes with areas should be placed earlier in the route table as they're more specific than routes without an area.

`{area:...}` can be used as a token in route templates if url space is uniform across all areas:

C#

```
var builder = WebApplication.CreateBuilder(args);  
  
builder.Services.AddControllersWithViews();  
  
var app = builder.Build();  
  
if (!app.Environment.IsDevelopment())  
{  
    app.UseExceptionHandler("/Home/Error");  
    app.UseHsts();  
}  
  
app.UseHttpsRedirection();  
app.UseStaticFiles();  
  
app.UseRouting();  
  
app.UseAuthorization();  
  
app.MapControllerRoute(  
    name: "MyArea",  
    pattern: "{area:exists}/{controller=Home}/{action=Index}/{id?}");  
  
app.MapControllerRoute(  
    name: "default",  
    pattern: "{controller=Home}/{action=Index}/{id?}");  
  
app.Run();
```

In the preceding code, `exists` applies a constraint that the route must match an area.

Using `{area:...}` with `MapControllerRoute`:

- Is the least complicated mechanism to adding routing to areas.

- Matches all controllers with the `[Area("Area name")]` attribute.

The following code uses [MapAreaControllerRoute](#) to create two named area routes:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapAreaControllerRoute(
    name: "MyAreaProducts",
    areaName: "Products",
    pattern: "Products/{controller=Home}/{action=Index}/{id?}");

app.MapAreaControllerRoute(
    name: "MyAreaServices",
    areaName: "Services",
    pattern: "Services/{controller=Home}/{action=Index}/{id?}");

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

app.Run();
```

For more information, see [Area routing](#).

## Link generation with MVC areas

The following code from the [sample download](#) shows link generation with the area specified:

CSHTML



```

<li>Anchor Tag Helper links</li>
<ul>
  <li>
    <a asp-area="Products" asp-controller="Home" asp-action="About">
      Products/Home/About
    </a>
  </li>
  <li>
    <a asp-area="Services" asp-controller="Home" asp-action="About">
      Services About
    </a>
  </li>
  <li>
    <a asp-area="" asp-controller="Home" asp-action="About">
      /Home/About
    </a>
  </li>
</ul>
<li>Html.ActionLink generated links</li>
<ul>
  <li>
    @Html.ActionLink("Product/Manage/About", "About", "Manage",
                      new { area = "Products" })
  </li>
</ul>
<li>Url.Action generated links</li>
<ul>
  <li>
    <a href=@Url.Action("About", "Manage", new { area = "Products" })>
      Products/Manage/About
    </a>
  </li>
</ul>
</ul>

```

The sample download includes a [partial view](#) that contains:

- The preceding links.
- Links similar to the preceding except `area` is not specified.

The partial view is referenced in the [layout file](#), so every page in the app displays the generated links. The links generated without specifying the area are only valid when referenced from a page in the same area and controller.

When the area or controller is not specified, routing depends on the [ambient](#) values. The current route values of the current request are considered ambient values for link generation. In many cases for the sample app, using the ambient values generates incorrect links with the markup that doesn't specify the area.

For more information, see [Routing to controller actions](#).

# Shared layout for Areas using the `_ViewStart.cshtml` file

To share a common layout for the entire app, keep the `_ViewStart.cshtml` in the [application root folder](#). For more information, see [Layout in ASP.NET Core](#)

## Application root folder

The application root folder is the folder containing the `Program.cs` file in a web app created with the ASP.NET Core templates.

## `_ViewImports.cshtml`

`/Views/_ViewImports.cshtml`, for MVC, and `/Pages/_ViewImports.cshtml` for Razor Pages, is not imported to views in areas. Use one of the following approaches to provide view imports to all views:

- Add `_ViewImports.cshtml` to the [application root folder](#). A `_ViewImports.cshtml` in the application root folder will apply to all views in the app.
- Copy the `_ViewImports.cshtml` file to the appropriate view folder under areas. For example, a Razor Pages app created with individual user accounts has a `_ViewImports.cshtml` file in the following folders:
  - `/Areas/Identity/Pages/_ViewImports.cshtml`
  - `/Pages/_ViewImports.cshtml`

The `_ViewImports.cshtml` file typically contains [Tag Helpers](#) imports, `@using`, and `@inject` statements. For more information, see [Importing Shared Directives](#).

## Change default area folder where views are stored

The following code changes the default area folder from `"Areas"` to `"MyAreas"`:

C#

```
using Microsoft.AspNetCore.Mvc.Razor;  
  
var builder = WebApplication.CreateBuilder(args);
```

```

builder.Services.Configure<RazorViewEngineOptions>(options =>
{
    options.AreaViewLocationFormats.Clear();

    options.AreaViewLocationFormats.Add("/MyAreas/{2}/Views/{1}/{0}.cshtml");

    options.AreaViewLocationFormats.Add("/MyAreas/{2}/Views/Shared/{0}.cshtml");
    options.AreaViewLocationFormats.Add("/Views/Shared/{0}.cshtml");
});

builder.Services.AddControllersWithViews();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(
    name: "MyArea",
    pattern: "{area:exists}/{controller=Home}/{action=Index}/{id?}");

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

app.Run();

```

## Areas with Razor Pages

Areas with Razor Pages require an `Areas/<area name>/Pages` folder in the root of the app. The following folder structure is used with the [sample app](#):

- Project name
  - Areas
    - Products
      - Pages
        - \_ViewImports
        - About
        - Index

- Services
  - Pages
    - Manage
      - About
      - Index

## Link generation with Razor Pages and areas

The following code from the [sample download](#) shows link generation with the area specified (for example, `asp-area="Products"`):

C#HTML

```
<li>Anchor Tag Helper links</li>
<ul>
  <li>
    <a asp-area="Products" asp-page="/About">
      Products/About
    </a>
  </li>
  <li>
    <a asp-area="Services" asp-page="/Manage/About">
      Services/Manage/About
    </a>
  </li>
  <li>
    <a asp-area="" asp-page="/About">
      /About
    </a>
  </li>
</ul>
<li>Url.Page generated links</li>
<ul>
  <li>
    <a href='@Url.Page("/Manage/About", new { area = "Services" })'>
      Services/Manage/About
    </a>
  </li>
  <li>
    <a href='@Url.Page("/About", new { area = "Products" })'>
      Products/About
    </a>
  </li>
</ul>
```

The sample download includes a [partial view](#) that contains the preceding links and the same links without specifying the area. The partial view is referenced in the [layout file](#), so every page in the app displays the generated links. The links generated without specifying the area are only valid when referenced from a page in the same area.

When the area is not specified, routing depends on the *ambient* values. The current route values of the current request are considered ambient values for link generation. In many cases for the sample app, using the ambient values generates incorrect links. For example, consider the links generated from the following code:

CHTML

```
<li>
  <a asp-page="/Manage/About">
    Services/Manage/About
  </a>
</li>
<li>
  <a asp-page="/About">
    /About
  </a>
</li>
```

For the preceding code:

- The link generated from `<a asp-page="/Manage/About">` is correct only when the last request was for a page in `Services` area. For example, `/Services/Manage/`, `/Services/Manage/Index`, Or `/Services/Manage/About`.
- The link generated from `<a asp-page="/About">` is correct only when the last request was for a page in `/Home`.
- The code is from the [sample download](#).

## Import namespace and Tag Helpers with `_ViewImports` file

A `_ViewImports.cshtml` file can be added to each area `Pages` folder to import the namespace and Tag Helpers to each Razor Page in the folder.

Consider the `Services` area of the sample code, which doesn't contain a `_ViewImports.cshtml` file. The following markup shows the `/Services/Manage/About` Razor Page:

CHTML

```
@page
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@model RPareas.Areas.Services.Pages.Manage.AboutModel
@{
    ViewData["Title"] = "Srv Mng About";
}
```

```
<div>
    ViewData["routeInfo"]: @ViewData["routeInfo"]
</div>

<a asp-area="Products" asp-page="/Index">
    Products/Index
</a>
```

In the preceding markup:

- The fully qualified class name must be used to specify the model (`@model RPareas.Areas.Services.Pages.Manage.AboutModel`).
- [Tag Helpers](#) are enabled by `@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers`

In the sample download, the Products area contains the following `_ViewImports.cshtml` file:

CSHTML

```
@namespace RPareas.Areas.Products.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

The following markup shows the `/Products/About` Razor Page:

CSHTML

```
@page
@model AboutModel
@{
    ViewData["Title"] = "Prod About";
}
```

In the preceding file, the namespace and `@addTagHelper` directive is imported to the file by the `Areas/Products/Pages/_ViewImports.cshtml` file.

For more information, see [Managing Tag Helper scope](#) and [Importing Shared Directives](#).

## Shared layout for Razor Pages Areas

To share a common layout for the entire app, move the `_ViewStart.cshtml` to the application root folder.

## Publishing Areas

All \*.cshtml files and files within the *wwwroot* directory are published to output when `<Project Sdk="Microsoft.NET.Sdk.Web">` is included in the \*.csproj file.

## Add MVC Area with Visual Studio

In Solution Explorer, right click the project and select **ADD > New Scaffolded Item**, then select **MVC Area**.

## Additional resources

- [View or download sample code](#) [\(how to download\)](#). The download sample provides a basic app for testing areas.
- [MyDisplayRouteInfo](#) and [ToCtxString](#) [are provided by the Rick.Docs.Samples.RouteInfo](#) [NuGet package](#). The methods display `Controller` and `Razor Page` route information.

# Filters in ASP.NET Core

Article • 06/17/2024

By [Kirk Larkin](#), [Rick Anderson](#), [Tom Dykstra](#), and [Steve Smith](#)

*Filters* in ASP.NET Core allow code to run before or after specific stages in the request processing pipeline.

Built-in filters handle tasks such as:

- Authorization, preventing access to resources a user isn't authorized for.
- Response caching, short-circuiting the request pipeline to return a cached response.

Custom filters can be created to handle cross-cutting concerns. Examples of cross-cutting concerns include error handling, caching, configuration, authorization, and logging. Filters avoid duplicating code. For example, an error handling exception filter could consolidate error handling.

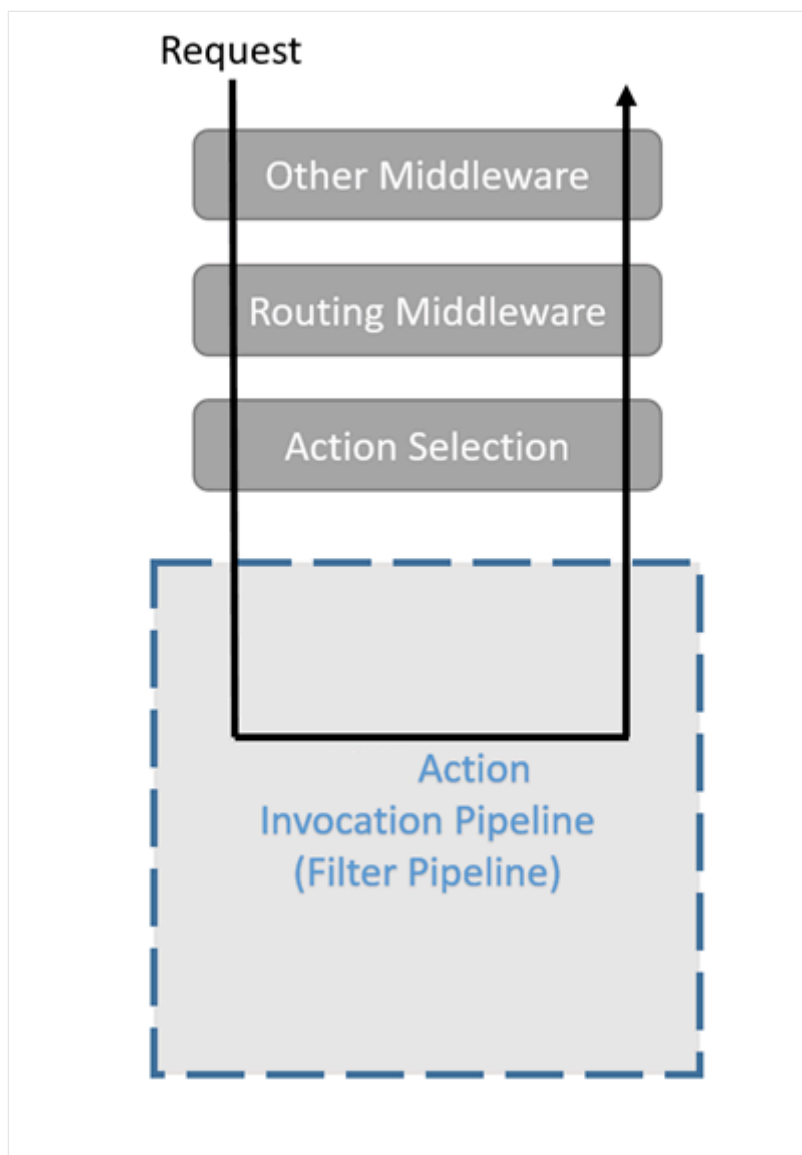
This document applies to Razor Pages, API controllers, and controllers with views. Filters don't work directly with [Razor components](#). A filter can only indirectly affect a component when:

- The component is embedded in a page or view.
- The page or controller and view uses the filter.

## How filters work

Filters run within the *ASP.NET Core action invocation pipeline*, sometimes referred to as the *filter pipeline*. The filter pipeline runs after ASP.NET Core selects the action to execute:





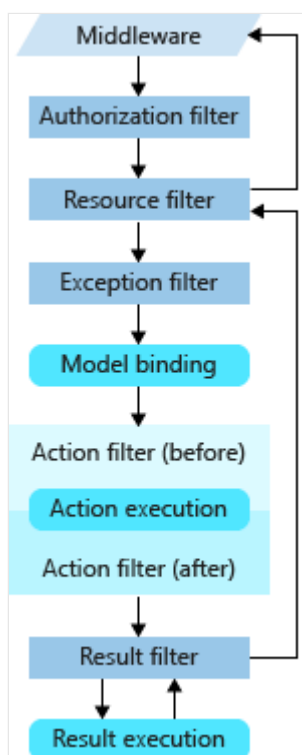
## Filter types

Each filter type is executed at a different stage in the filter pipeline:

- **Authorization filters:**
  - Run first.
  - Determine whether the user is authorized for the request.
  - Short-circuit the pipeline if the request is not authorized.
- **Resource filters:**
  - Run after authorization.
  - **OnResourceExecuting** runs code before the rest of the filter pipeline. For example, `OnResourceExecuting` runs code before model binding.
  - **OnResourceExecuted** runs code after the rest of the pipeline has completed.
- **Action filters:**
  - Run immediately before and after an action method is called.
  - Can change the arguments passed into an action.

- Can change the result returned from the action.
- Are **not** supported in Razor Pages.
- **Endpoint filters:**
  - Run immediately before and after an action method is called.
  - Can change the arguments passed into an action.
  - Can change the result returned from the action.
  - Are **not** supported in Razor Pages.
  - Can be invoked on both actions and route handler-based endpoints.
- **Exception filters** apply global policies to unhandled exceptions that occur before the response body has been written to.
- **Result filters:**
  - Run immediately before and after the execution of action results.
  - Run only when the action method executes successfully.
  - Are useful for logic that must surround view or formatter execution.

The following diagram shows how filter types interact in the filter pipeline:



Razor Pages also support **Razor Page filters**, which run before and after a Razor Page handler.

## Implementation

Filters support both synchronous and asynchronous implementations through different interface definitions.

Synchronous filters run before and after their pipeline stage. For example, [OnActionExecuting](#) is called before the action method is called. [OnActionExecuted](#) is called after the action method returns:

C#

```
public class SampleActionFilter : IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context)
    {
        // Do something before the action executes.
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        // Do something after the action executes.
    }
}
```

Asynchronous filters define an `On-Stage-ExecutionAsync` method. For example, [OnActionExecutionAsync](#):

C#

```
public class SampleAsyncActionFilter : IAsyncActionFilter
{
    public async Task OnActionExecutionAsync(
        ActionExecutingContext context, ActionExecutionDelegate next)
    {
        // Do something before the action executes.
        await next();
        // Do something after the action executes.
    }
}
```

In the preceding code, the `SampleAsyncActionFilter` has an [ActionExecutionDelegate](#), `next`, which executes the action method.

## Multiple filter stages

Interfaces for multiple filter stages can be implemented in a single class. For example, the [ActionFilterAttribute](#) class implements:

- Synchronous: [IActionFilter](#) and [IResultFilter](#)
- Asynchronous: [IAsyncActionFilter](#) and [IAsyncResultFilter](#)
- [IOrderedFilter](#)

Implement **either** the synchronous or the async version of a filter interface, **not** both. The runtime checks first to see if the filter implements the async interface, and if so, it calls that. If not, it calls the synchronous interface's method(s). If both asynchronous and synchronous interfaces are implemented in one class, only the async method is called. When using abstract classes like [ActionFilterAttribute](#), override only the synchronous methods or the asynchronous methods for each filter type.

## Built-in filter attributes

ASP.NET Core includes built-in attribute-based filters that can be subclassed and customized. For example, the following result filter adds a header to the response:

C#

```
public class ResponseHeaderAttribute : ActionFilterAttribute
{
    private readonly string _name;
    private readonly string _value;

    public ResponseHeaderAttribute(string name, string value) =>
        (_name, _value) = (name, value);

    public override void OnResultExecuting(ResultExecutingContext context)
    {
        context.HttpContext.Response.Headers.Add(_name, _value);

        base.OnResultExecuting(context);
    }
}
```

Attributes allow filters to accept arguments, as shown in the preceding example. Apply the `ResponseHeaderAttribute` to a controller or action method and specify the name and value of the HTTP header:

C#

```
[ResponseHeader("Filter-Header", "Filter Value")]
public class ResponseHeaderController : ControllerBase
{
    public IActionResult Index() =>
        Content("Examine the response headers using the F12 developer tools.");

    // ...
}
```

Use a tool such as the [browser developer tools](#) to examine the headers. Under **Response Headers**, `filter-header: Filter Value` is displayed.

The following code applies `ResponseHeaderAttribute` to both a controller and an action:

C#

```
[ResponseHeader("Filter-Header", "Filter Value")]
public class ResponseHeaderController : ControllerBase
{
    public IActionResult Index() =>
        Content("Examine the response headers using the F12 developer
tools.");

    // ...

    [ResponseHeader("Another-Filter-Header", "Another Filter Value")]
    public IActionResult Multiple() =>
        Content("Examine the response headers using the F12 developer
tools.");
}
```

Responses from the `Multiple` action include the following headers:

- `filter-header: Filter Value`
- `another-filter-header: Another Filter Value`

Several of the filter interfaces have corresponding attributes that can be used as base classes for custom implementations.

Filter attributes:

- [ActionFilterAttribute](#)
- [ExceptionFilterAttribute](#)
- [ResultFilterAttribute](#)
- [FormatFilterAttribute](#)
- [ServiceFilterAttribute](#)
- [TypeFilterAttribute](#)

Filters cannot be applied to Razor Page handler methods. They can be applied either to the Razor Page model or globally.

## Filter scopes and order of execution

A filter can be added to the pipeline at one of three *scopes*:

- Using an attribute on a controller or Razor Page.
- Using an attribute on a controller action. Filter attributes cannot be applied to Razor Pages handler methods.
- Globally for all controllers, actions, and Razor Pages as shown in the following code:

```
C#

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews(options =>
{
    options.Filters.Add<GlobalSampleActionFilter>();
});
```

## Default order of execution

When there are multiple filters for a particular stage of the pipeline, scope determines the default order of filter execution. Global filters surround class filters, which in turn surround method filters.

As a result of filter nesting, the *after* code of filters runs in the reverse order of the *before* code. The filter sequence:

- The *before* code of global filters.
  - The *before* code of controller filters.
    - The *before* code of action method filters.
    - The *after* code of action method filters.
  - The *after* code of controller filters.
- The *after* code of global filters.

The following example illustrates the order in which filter methods run for synchronous action filters:

 Expand table

Sequence	Filter scope	Filter method
1	Global	OnActionExecuting
2	Controller	OnActionExecuting
3	Action	OnActionExecuting

Sequence	Filter scope	Filter method
4	Action	OnActionExecuted
5	Controller	OnActionExecuted
6	Global	OnActionExecuted

## Controller level filters

Every controller that inherits from [Controller](#) includes the [OnActionExecuting](#), [OnActionExecutionAsync](#), and [OnActionExecuted](#) methods. These methods wrap the filters that run for a given action:

- [OnActionExecuting](#) runs before any of the action's filters.
- [OnActionExecuted](#) runs after all of the action's filters.
- [OnActionExecutionAsync](#) runs before any of the action's filters. Code after a call to [next](#) runs after the action's filters.

The following [ControllerFiltersController](#) class:

- Applies the [SampleActionFilterAttribute](#) ([\[SampleActionFilter\]](#)) to the controller.
- Overrides [OnActionExecuting](#) and [OnActionExecuted](#).

C#

```
[SampleActionFilter]
public class ControllerFiltersController : Controller
{
    public override void OnActionExecuting(ActionExecutingContext context)
    {
        Console.WriteLine(
            $"- {nameof(ControllerFiltersController)}.
{nameof(OnActionExecuting)}");

        base.OnActionExecuting(context);
    }

    public override void OnActionExecuted(ActionExecutedContext context)
    {
        Console.WriteLine(
            $"- {nameof(ControllerFiltersController)}.
{nameof(OnActionExecuted)}");

        base.OnActionExecuted(context);
    }

    public IActionResult Index()
```

```

{
    Console.WriteLine(
        $"- {nameof(ControllerFiltersController)}.{nameof(Index)}");

    return Content("Check the Console.");
}
}

```

Navigating to `https://localhost:<port>/ControllerFilters` runs the following code:

- `ControllerFiltersController.OnActionExecuting`
  - `GlobalSampleActionFilter.OnActionExecuting`
    - `SampleActionFilterAttribute.OnActionExecuting`
      - `ControllerFiltersController.Index`
    - `SampleActionFilterAttribute.OnActionExecuted`
  - `GlobalSampleActionFilter.OnActionExecuted`
- `ControllerFiltersController.OnActionExecuted`

Controller level filters set the [Order](#) property to `int.MinValue`. Controller level filters can **not** be set to run after filters applied to methods. Order is explained in the next section.

For Razor Pages, see [Implement Razor Page filters by overriding filter methods](#).

## Override the default order

The default sequence of execution can be overridden by implementing [IOrderedFilter](#).

`IOrderedFilter` exposes the [Order](#) property that takes precedence over scope to determine the order of execution. A filter with a lower `Order` value:

- Runs the *before* code before that of a filter with a higher value of `Order`.
- Runs the *after* code after that of a filter with a higher `Order` value.

In the [Controller level filters](#) example, `GlobalSampleActionFilter` has global scope so it runs before `SampleActionFilterAttribute`, which has controller scope. To make `SampleActionFilterAttribute` run first, set its order to `int.MinValue`:

C#

```

[SampleActionFilter(Order = int.MinValue)]
public class ControllerFiltersController : Controller
{
    // ...
}

```



To make the global filter `GlobalSampleActionFilter` run first, set its `Order` to `int.MinValue`:

C#

```
builder.Services.AddControllersWithViews(options =>
{
    options.Filters.Add<GlobalSampleActionFilter>(int.MinValue);
});
```

## Cancellation and short-circuiting

The filter pipeline can be short-circuited by setting the `Result` property on the `ResourceExecutingContext` parameter provided to the filter method. For example, the following Resource filter prevents the rest of the pipeline from executing:

C#

```
public class ShortCircuitingResourceFilterAttribute : Attribute,
IResourceFilter
{
    public void OnResourceExecuting(ResourceExecutingContext context)
    {
        context.Result = new ContentResult
        {
            Content = nameof(ShortCircuitingResourceFilterAttribute)
        };
    }

    public void OnResourceExecuted(ResourceExecutedContext context) { }
}
```

In the following code, both the `[ShortCircuitingResourceFilter]` and the `[ResponseHeader]` filter target the `Index` action method. The `ShortCircuitingResourceFilterAttribute` filter:

- Runs first, because it's a Resource Filter and `ResponseHeaderAttribute` is an Action Filter.
- Short-circuits the rest of the pipeline.

Therefore the `ResponseHeaderAttribute` filter never runs for the `Index` action. This behavior would be the same if both filters were applied at the action method level, provided the `ShortCircuitingResourceFilterAttribute` ran first. The `ShortCircuitingResourceFilterAttribute` runs first because of its filter type:

C#

```
[ResponseHeader("Filter-Header", "Filter Value")]
public class ShortCircuitingController : Controller
{
    [ShortCircuitingResourceFilter]
    public IActionResult Index() =>
        Content($"{nameof(ShortCircuitingController)}.{nameof(Index)}");
}
```

## Dependency injection

Filters can be added by type or by instance. If an instance is added, that instance is used for every request. If a type is added, it's type-activated. A type-activated filter means:

- An instance is created for each request.
- Any constructor dependencies are populated by [dependency injection](#) (DI).

Filters that are implemented as attributes and added directly to controller classes or action methods cannot have constructor dependencies provided by [dependency injection](#) (DI). Constructor dependencies cannot be provided by DI because attributes must have their constructor parameters supplied where they're applied.

The following filters support constructor dependencies provided from DI:

- [ServiceFilterAttribute](#)
- [TypeFilterAttribute](#)
- [IFilterFactory](#) implemented on the attribute.

The preceding filters can be applied to a controller or an action.

Loggers are available from DI. However, avoid creating and using filters purely for logging purposes. The [built-in framework logging](#) typically provides what's needed for logging. Logging added to filters:

- Should focus on business domain concerns or behavior specific to the filter.
- Should **not** log actions or other framework events. The built-in filters already log actions and framework events.

## ServiceFilterAttribute

Service filter implementation types are registered in `Program.cs`. A [ServiceFilterAttribute](#) retrieves an instance of the filter from DI.

The following code shows the `LoggingResponseHeaderFilterService` class, which uses DI:

C#

```
public class LoggingResponseHeaderFilterService : IResultFilter
{
    private readonly ILogger _logger;

    public LoggingResponseHeaderFilterService(
        ILogger<LoggingResponseHeaderFilterService> logger) =>
        _logger = logger;

    public void OnResultExecuting(ResultExecutingContext context)
    {
        _logger.LogInformation(
            $"- {nameof(LoggingResponseHeaderFilterService)}.
{nameof(OnResultExecuting)}");

        context.HttpContext.Response.Headers.Add(
            nameof(OnResultExecuting),
            nameof(LoggingResponseHeaderFilterService));
    }

    public void OnResultExecuted(ResultExecutedContext context)
    {
        _logger.LogInformation(
            $"- {nameof(LoggingResponseHeaderFilterService)}.
{nameof(OnResultExecuted)}");
    }
}
```

In the following code, `LoggingResponseHeaderFilterService` is added to the DI container:

C#

```
builder.Services.AddScoped<LoggingResponseHeaderFilterService>();
```

In the following code, the `ServiceFilter` attribute retrieves an instance of the `LoggingResponseHeaderFilterService` filter from DI:

C#

```
[ServiceFilter<LoggingResponseHeaderFilterService>]
public IActionResult WithServiceFilter() =>
    Content($"- {nameof(FilterDependenciesController)}.
{nameof(WithServiceFilter)}");
```

When using `ServiceFilterAttribute`, setting `ServiceFilterAttribute.IsReusable`:

- Provides a hint that the filter instance *may* be reused outside of the request scope it was created within. The ASP.NET Core runtime doesn't guarantee:
  - That a single instance of the filter will be created.
  - The filter will not be re-requested from the DI container at some later point.
- Shouldn't be used with a filter that depends on services with a lifetime other than singleton.

[ServiceFilterAttribute](#) implements [IFilterFactory](#). [IFilterFactory](#) exposes the [CreateInstance](#) method for creating an [IFilterMetadata](#) instance. [CreateInstance](#) loads the specified type from DI.

## TypeFilterAttribute

[TypeFilterAttribute](#) is similar to [ServiceFilterAttribute](#), but its type isn't resolved directly from the DI container. It instantiates the type by using [Microsoft.Extensions.DependencyInjection.ObjectFactory](#).

Because [TypeFilterAttribute](#) types aren't resolved directly from the DI container:

- Types that are referenced using the [TypeFilterAttribute](#) don't need to be registered with the DI container. They do have their dependencies fulfilled by the DI container.
- [TypeFilterAttribute](#) can optionally accept constructor arguments for the type.

When using [TypeFilterAttribute](#), setting [TypeFilterAttribute.IsReusable](#):

- Provides hint that the filter instance *may* be reused outside of the request scope it was created within. The ASP.NET Core runtime provides no guarantees that a single instance of the filter will be created.
- Should not be used with a filter that depends on services with a lifetime other than singleton.

The following example shows how to pass arguments to a type using

[TypeFilterAttribute](#):

C#

```
[TypeFilter(typeof(LoggingResponseHeaderFilter),
    Arguments = new object[] { "Filter-Header", "Filter Value" })]
public IActionResult WithTypeFilter() =>
    Content($"- {nameof(FilterDependenciesController)}.
{nameof(WithTypeFilter)}");
```

# Authorization filters

Authorization filters:

- Are the first filters run in the filter pipeline.
- Control access to action methods.
- Have a before method, but no after method.

Custom authorization filters require a custom authorization framework. Prefer configuring the authorization policies or writing a custom authorization policy over writing a custom filter. The built-in authorization filter:

- Calls the authorization system.
- Does not authorize requests.

Do **not** throw exceptions within authorization filters:

- The exception will not be handled.
- Exception filters will not handle the exception.

Consider issuing a challenge when an exception occurs in an authorization filter.

Learn more about [Authorization](#).

## Resource filters

Resource filters:

- Implement either the [IResourceFilter](#) or [IAsyncResourceFilter](#) interface.
- Execution wraps most of the filter pipeline.
- Only [Authorization filters](#) run before resource filters.

Resource filters are useful to short-circuit most of the pipeline. For example, a caching filter can avoid the rest of the pipeline on a cache hit.

Resource filter examples:

- [The short-circuiting resource filter](#) shown previously.
- [DisableFormValueModelBindingAttribute](#) [↗](#):
  - Prevents model binding from accessing the form data.
  - Used for large file uploads to prevent the form data from being read into memory.

# Action filters

Action filters do **not** apply to Razor Pages. Razor Pages supports [IPageFilter](#) and [IAsyncPageFilter](#). For more information, see [Filter methods for Razor Pages](#).

Action filters:

- Implement either the [IActionFilter](#) or [IAsyncActionFilter](#) interface.
- Their execution surrounds the execution of action methods.

The following code shows a sample action filter:

```
C#  
  
public class SampleActionFilter : IActionFilter  
{  
    public void OnActionExecuting(ActionExecutingContext context)  
    {  
        // Do something before the action executes.  
    }  
  
    public void OnActionExecuted(ActionExecutedContext context)  
    {  
        // Do something after the action executes.  
    }  
}
```

The [ActionExecutingContext](#) provides the following properties:

- [ActionArguments](#) - enables reading the inputs to an action method.
- [Controller](#) - enables manipulating the controller instance.
- [Result](#) - setting `Result` short-circuits execution of the action method and subsequent action filters.

Throwing an exception in an action method:

- Prevents running of subsequent filters.
- Unlike setting `Result`, is treated as a failure instead of a successful result.

The [ActionExecutedContext](#) provides `Controller` and `Result` plus the following properties:

- [Canceled](#) - True if the action execution was short-circuited by another filter.
- [Exception](#) - Non-null if the action or a previously run action filter threw an exception. Setting this property to null:
  - Effectively handles the exception.

- `Result` is executed as if it was returned from the action method.

For an `IAsyncActionFilter`, a call to the [ActionExecutionDelegate](#):

- Executes any subsequent action filters and the action method.
- Returns `ActionExecutedContext`.

To short-circuit, assign [Microsoft.AspNetCore.Mvc.Filters.ActionExecutingContext.Result](#) to a result instance and don't call `next` (the `ActionExecutionDelegate`).

The framework provides an abstract [ActionFilterAttribute](#) that can be subclassed.

The `OnActionExecuting` action filter can be used to:

- Validate model state.
- Return an error if the state is invalid.

C#

```
public class ValidateModelAttribute : ActionFilterAttribute
{
    public override void OnActionExecuting(ActionExecutingContext context)
    {
        if (!context.ModelState.IsValid)
        {
            context.Result = new BadRequestObjectResult(context.ModelState);
        }
    }
}
```

### ⓘ Note

Controllers annotated with the `[ApiController]` attribute automatically validate model state and return a 400 response. For more information, see [Automatic HTTP 400 responses](#).

The `OnActionExecuted` method runs after the action method:

- And can see and manipulate the results of the action through the [Result](#) property.
- [Canceled](#) is set to true if the action execution was short-circuited by another filter.
- [Exception](#) is set to a non-null value if the action or a subsequent action filter threw an exception. Setting `Exception` to null:
  - Effectively handles an exception.
  - `ActionExecutedContext.Result` is executed as if it were returned normally from the action method.

# Exception filters

Exception filters:

- Implement [IExceptionFilter](#) or [IAsyncExceptionFilter](#).
- Can be used to implement common error handling policies.

The following sample exception filter displays details about exceptions that occur when the app is in development:

C#

```
public class SampleExceptionFilter : IExceptionFilter
{
    private readonly IHostEnvironment _hostEnvironment;

    public SampleExceptionFilter(IHostEnvironment hostEnvironment) =>
        _hostEnvironment = hostEnvironment;

    public void OnException(ExceptionContext context)
    {
        if (!_hostEnvironment.IsDevelopment())
        {
            // Don't display exception details unless running in
            Development.
            return;
        }

        context.Result = new ContentResult
        {
            Content = context.Exception.ToString()
        };
    }
}
```

The following code tests the exception filter:

C#

```
[TypeFilter<SampleExceptionFilter>]
public class ExceptionController : Controller
{
    public IActionResult Index() =>
        Content($"{nameof(ExceptionController)}.{nameof(Index)}");
}
```

Exception filters:

- Don't have before and after events.



- Implement [OnException](#) or [OnExceptionAsync](#).
- Handle unhandled exceptions that occur in Razor Page or controller creation, [model binding](#), action filters, or action methods.
- Do **not** catch exceptions that occur in resource filters, result filters, or MVC result execution.

To handle an exception, set the [ExceptionHandled](#) property to `true` or assign the [Result](#) property. This stops propagation of the exception. An exception filter can't turn an exception into a "success". Only an action filter can do that.

Exception filters:

- Are good for trapping exceptions that occur within actions.
- Are not as flexible as error handling middleware.

Prefer middleware for exception handling. Use exception filters only where error handling *differs* based on which action method is called. For example, an app might have action methods for both API endpoints and for views/HTML. The API endpoints could return error information as JSON, while the view-based actions could return an error page as HTML.

## Result filters

Result filters:

- Implement an interface:
  - [IResultFilter](#) or [IAsyncResultFilter](#)
  - [IAlwaysRunResultFilter](#) or [IAsyncAlwaysRunResultFilter](#)
- Their execution surrounds the execution of action results.

## IResultFilter and IAsyncResultFilter

The following code shows a sample result filter:

C#

```
public class SampleResultFilter : IResultFilter
{
    public void OnResultExecuting(ResultExecutingContext context)
    {
        // Do something before the result executes.
    }

    public void OnResultExecuted(ResultExecutedContext context)
    {
```

```
        // Do something after the result executes.  
    }  
}
```

The kind of result being executed depends on the action. An action returning a view includes all razor processing as part of the [ViewResult](#) being executed. An API method might perform some serialization as part of the execution of the result. Learn more about [action results](#).

Result filters are only executed when an action or action filter produces an action result. Result filters are not executed when:

- An authorization filter or resource filter short-circuits the pipeline.
- An exception filter handles an exception by producing an action result.

The [Microsoft.AspNetCore.Mvc.Filters.IResultFilter.OnResultExecuting](#) method can short-circuit execution of the action result and subsequent result filters by setting [Microsoft.AspNetCore.Mvc.Filters.ResultExecutingContext.Cancel](#) to `true`. Write to the response object when short-circuiting to avoid generating an empty response. Throwing an exception in `IResultFilter.OnResultExecuting`:

- Prevents execution of the action result and subsequent filters.
- Is treated as a failure instead of a successful result.

When the [Microsoft.AspNetCore.Mvc.Filters.IResultFilter.OnResultExecuted](#) method runs, the response has probably already been sent to the client. If the response has already been sent to the client, it cannot be changed.

`ResultExecutedContext.Canceled` is set to `true` if the action result execution was short-circuited by another filter.

`ResultExecutedContext.Exception` is set to a non-null value if the action result or a subsequent result filter threw an exception. Setting `Exception` to null effectively handles an exception and prevents the exception from being thrown again later in the pipeline. There is no reliable way to write data to a response when handling an exception in a result filter. If the headers have been flushed to the client when an action result throws an exception, there's no reliable mechanism to send a failure code.

For an [IAsyncResultFilter](#), a call to `await next` on the [ResultExecutionDelegate](#) executes any subsequent result filters and the action result. To short-circuit, set [ResultExecutingContext.Cancel](#) to `true` and don't call the `ResultExecutionDelegate`:

```

public class SampleAsyncResultFilter : IAsyncResultFilter
{
    public async Task OnResultExecutionAsync(
        ResultExecutingContext context, ResultExecutionDelegate next)
    {
        if (context.Result is not EmptyResult)
        {
            await next();
        }
        else
        {
            context.Cancel = true;
        }
    }
}

```

The framework provides an abstract `ResultFilterAttribute` that can be subclassed. The `ResponseHeaderAttribute` class shown previously is an example of a result filter attribute.

## IAlwaysRunResultFilter and IAsyncAlwaysRunResultFilter

The `IAlwaysRunResultFilter` and `IAsyncAlwaysRunResultFilter` interfaces declare an `IResultFilter` implementation that runs for all action results. This includes action results produced by:

- Authorization filters and resource filters that short-circuit.
- Exception filters.

For example, the following filter always runs and sets an action result (`ObjectResult`) with a 422 *Unprocessable Entity* status code when content negotiation fails:

C#

```

public class UnprocessableResultFilter : IAlwaysRunResultFilter
{
    public void OnResultExecuting(ResultExecutingContext context)
    {
        if (context.Result is StatusCodeResult statusCodeResult
            && statusCodeResult.StatusCode ==
            StatusCodes.Status415UnsupportedMediaType)
        {
            context.Result = new ObjectResult("Unprocessable")
            {
                StatusCode = StatusCodes.Status422UnprocessableEntity
            };
        }
    }
}

```

```
    public void OnResultExecuted(ResultExecutedContext context) { }  
}
```

## IFilterFactory

`IFilterFactory` implements `IFilterMetadata`. Therefore, an `IFilterFactory` instance can be used as an `IFilterMetadata` instance anywhere in the filter pipeline. When the runtime prepares to invoke the filter, it attempts to cast it to an `IFilterFactory`. If that cast succeeds, the `CreateInstance` method is called to create the `IFilterMetadata` instance that is invoked. This provides a flexible design, since the precise filter pipeline doesn't need to be set explicitly when the app starts.

`IFilterFactory.IsReusable`:

- Is a hint by the factory that the filter instance created by the factory may be reused outside of the request scope it was created within.
- Should **not** be used with a filter that depends on services with a lifetime other than singleton.

The ASP.NET Core runtime doesn't guarantee:

- That a single instance of the filter will be created.
- The filter will not be re-requested from the DI container at some later point.

### Warning

Only configure `IFilterFactory.IsReusable` to return `true` if the source of the filters is unambiguous, the filters are stateless, and the filters are safe to use across multiple HTTP requests. For instance, don't return filters from DI that are registered as scoped or transient if `IFilterFactory.IsReusable` returns `true`.

`IFilterFactory` can be implemented using custom attribute implementations as another approach to creating filters:

C#

```
public class ResponseHeaderFilterFactory : Attribute, IFilterFactory  
{  
    public bool IsReusable => false;  
}
```

```

public IFilterMetadata CreateInstance(IServiceProvider serviceProvider)
=>
    new InternalResponseHeaderFilter();

private class InternalResponseHeaderFilter : IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context) =>
        context.HttpContext.Response.Headers.Add(
            nameof(OnActionExecuting),
            nameof(InternalResponseHeaderFilter));

    public void OnActionExecuted(ActionExecutedContext context) { }
}

```

The filter is applied in the following code:

C#

```

[ResponseHeaderFilterFactory]
public IActionResult Index() =>
    Content($"{nameof(FilterFactoryController)}.{nameof(Index)}");

```

## IFilterFactory implemented on an attribute

Filters that implement `IFilterFactory` are useful for filters that:

- Don't require passing parameters.
- Have constructor dependencies that need to be filled by DI.

`TypeFilterAttribute` implements `IFilterFactory`. `IFilterFactory` exposes the `CreateInstance` method for creating an `IFilterMetadata` instance. `CreateInstance` loads the specified type from the services container (DI).

C#

```

public class SampleActionTypeFilterAttribute : TypeFilterAttribute
{
    public SampleActionTypeFilterAttribute()
        : base(typeof(InternalSampleActionFilter)) { }

    private class InternalSampleActionFilter : IActionFilter
    {
        private readonly ILogger<InternalSampleActionFilter> _logger;

        public
        InternalSampleActionFilter(ILogger<InternalSampleActionFilter> logger) =>
            _logger = logger;

        public void OnActionExecuting(ActionExecutingContext context)

```

```

    {
        _logger.LogInformation(
            $"- {nameof(InternalSampleActionFilter)}".
{nameof(OnActionExecuting)}");
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        _logger.LogInformation(
            $"- {nameof(InternalSampleActionFilter)}".
{nameof(OnActionExecuted)}");
    }
}
}

```

The following code shows three approaches to applying the filter:

C#

```

[SampleActionTypeFilter]
public IActionResult WithDirectAttribute() =>
    Content($"- {nameof(FilterFactoryController)}".
{nameof(WithDirectAttribute)}");

[TypeFilter<SampleActionTypeFilterAttribute>]
public IActionResult WithTypeFilterAttribute() =>
    Content($"- {nameof(FilterFactoryController)}".
{nameof(WithTypeFilterAttribute)}");

[ServiceFilter<SampleActionTypeFilterAttribute>]
public IActionResult WithServiceFilterAttribute() =>
    Content($"- {nameof(FilterFactoryController)}".
{nameof(WithServiceFilterAttribute)}");

```

In the preceding code, the first approach to applying the filter is preferred.

## Use middleware in the filter pipeline

Resource filters work like [middleware](#) in that they surround the execution of everything that comes later in the pipeline. But filters differ from middleware in that they're part of the runtime, which means that they have access to context and constructs.

To use middleware as a filter, create a type with a `Configure` method that specifies the middleware to inject into the filter pipeline. The following example uses middleware to set a response header:

C#

```

public class FilterMiddlewarePipeline
{
    public void Configure(IApplicationBuilder app)
    {
        app.Use(async (context, next) =>
        {
            context.Response.Headers.Add("Pipeline", "Middleware");

            await next();
        });
    }
}

```

Use the [MiddlewareFilterAttribute](#) to run the middleware:

C#

```

[MiddlewareFilter<FilterMiddlewarePipeline>]
public class FilterMiddlewareController : Controller
{
    public IActionResult Index() =>
        Content($"{nameof(FilterMiddlewareController)}.{nameof(Index)}");
}

```

Middleware filters run at the same stage of the filter pipeline as Resource filters, before model binding and after the rest of the pipeline.

## Thread safety

When passing an *instance* of a filter into `Add`, instead of its `Type`, the filter is a singleton and is **not** thread-safe.

## Additional resources


- [View or download sample](#) <sup>↗</sup> (how to download).
- [Filter methods for Razor Pages in ASP.NET Core](#)

# ASP.NET Core Razor SDK

Article • 04/10/2024

By [Rick Anderson](#) 

## Overview

The [.NET 6.0 SDK](#)  includes the `Microsoft.NET.Sdk.Razor` MSBuild SDK (Razor SDK).

The Razor SDK:

- Is required to build, package, and publish projects containing [Razor](#) files for ASP.NET Core MVC-based or [Blazor](#) projects.
- Includes a set of predefined properties, and items that allow customizing the compilation of Razor (`.cshtml` or `.razor`) files.


The Razor SDK includes `Content` items with `Include` attributes set to the `**\*.cshtml` and `**\*.razor` globbing patterns. Matching files are published.

## Prerequisites

[.NET 6.0 SDK](#) 

## Use the Razor SDK

Most web apps aren't required to explicitly reference the Razor SDK.

To use the Razor SDK to build class libraries containing Razor views or Razor Pages, we recommend starting with the Razor class library (RCL) project template. An RCL that's used to build Blazor (`.razor`) files minimally requires a reference to the [Microsoft.AspNetCore.Components](#)  package. An RCL that's used to build Razor views or pages (`.cshtml` files) minimally requires targeting `netcoreapp3.0` or later and has a `FrameworkReference` to the [Microsoft.AspNetCore.App metapackage](#) in its project file.

## Properties

The following properties control the Razor's SDK behavior as part of a project build:

- `RazorCompileOnBuild`: When `true`, compiles and emits the Razor assembly as part of building the project. Defaults to `true`.



- `RazorCompileOnPublish`: When `true`, compiles and emits the Razor assembly as part of publishing the project. Defaults to `true`.
- `UseRazorSourceGenerator`: Defaults to `true`. When `true`:
  - Compiles using source generation.
  - Doesn't create `<app_name>.Views.dll`. Views are included in `<app_name>.dll`.
  - Supports [.NET Hot Reload](#).

The properties and items in the following table are used to configure inputs and output to the Razor SDK.

[Expand table](#)

Items	Description
<code>RazorGenerate</code>	Item elements ( <code>.cshtml</code> files) that are inputs to code generation.
<code>RazorComponent</code>	Item elements ( <code>.razor</code> files) that are inputs to Razor component code generation.
<code>RazorCompile</code>	Item elements ( <code>.cs</code> files) that are inputs to Razor compilation targets. Use this <code>ItemGroup</code> to specify additional files to be compiled into the Razor assembly.
<code>RazorEmbeddedResource</code>	Item elements added as embedded resources to the generated Razor assembly.

[Expand table](#)

Property	Description
<code>RazorOutputPath</code>	The Razor output directory.
<code>RazorCompileToolset</code>	Used to determine the toolset used to build the Razor assembly. Valid values are <code>Implicit</code> , <code>RazorSDK</code> , and <code>PrecompilationTool</code> .
<a href="#">EnableDefaultContentItems</a> <a href="#">↗</a>	Default is <code>true</code> . When <code>true</code> , includes <code>web.config</code> , <code>.json</code> , and <code>.cshtml</code> files as content in the project. When referenced via <code>Microsoft.NET.Sdk.Web</code> , files under <code>wwwroot</code> and config files are also included.
<code>EnableDefaultRazorGenerateItems</code>	When <code>true</code> , includes <code>.cshtml</code> files from <code>Content</code> items in <code>RazorGenerate</code> items.
<code>GenerateRazorTargetAssemblyInfo</code>	Not used in .NET 6 and later.

Property	Description
<code>EnableDefaultRazorTargetAssemblyInfoAttributes</code>	Not used in .NET 6 and later.
<code>CopyRazorGenerateFilesToPublishDirectory</code>	When <code>true</code> , copies <code>RazorGenerate</code> items ( <code>.cshtml</code> ) files to the publish directory. Typically, Razor files aren't required for a published app if they participate in compilation at build-time or publish-time. Defaults to <code>false</code> .
<code>PreserveCompilationReferences</code>	When <code>true</code> , copy reference assembly items to the publish directory. Typically, reference assemblies aren't required for a published app if Razor compilation occurs at build-time or publish-time. Set to <code>true</code> if your published app requires runtime compilation. For example, set the value to <code>true</code> if the app modifies <code>.cshtml</code> files at runtime or uses embedded views. Defaults to <code>false</code> .
<code>IncludeRazorContentInPack</code>	When <code>true</code> , all Razor content items ( <code>.cshtml</code> files) are marked for inclusion in the generated NuGet package. Defaults to <code>false</code> .
<code>EmbedRazorGenerateSources</code>	When <code>true</code> , adds <code>RazorGenerate</code> ( <code>.cshtml</code> ) items as embedded files to the generated Razor assembly. Defaults to <code>false</code> .
<code>GenerateMvcApplicationPartsAssemblyAttributes</code>	Not used in .NET 6 and later.
<code>DefaultWebContentItemExcludes</code>	A globbing pattern for item elements that are to be excluded from the <code>Content</code> item group in projects targeting the Web or Razor SDK
<code>ExcludeConfigFilesFromBuildOutput</code>	When <code>true</code> , <code>.config</code> and <code>.json</code> files do not get copied to the build output directory.
<code>AddRazorSupportForMvc</code>	When <code>true</code> , configures the Razor SDK to add support for the MVC configuration that is required when building applications containing MVC views or Razor Pages. This property is implicitly set for .NET Core 3.0 or later projects targeting the Web SDK
<code>RazorLangVersion</code>	The version of the Razor Language to target.
<code>EmitCompilerGeneratedFiles</code>	When set to <code>true</code> , the generated source files are written to disk. Setting to <code>true</code> is useful

Property	Description
	when debugging the compiler. The default is <code>false</code> .

For more information on properties, see [MSBuild properties](#).

## Runtime compilation of Razor views

- By default, the Razor SDK doesn't publish reference assemblies that are required to perform runtime compilation. This results in compilation failures when the application model relies on runtime compilation—for example, the app uses embedded views or changes views after the app is published. Set `CopyRefAssembliesToPublishDirectory` to `true` to continue publishing reference assemblies. Both code generation and compilation are supported by a single call to the compiler. A single assembly is produced that contains the app types and the generated views.
- For a web app, ensure your app is targeting the `Microsoft.NET.Sdk.Web` SDK.

## Razor language version

When targeting the `Microsoft.NET.Sdk.Web` SDK, the Razor language version is inferred from the app's target framework version. For projects targeting the `Microsoft.NET.Sdk.Razor` SDK or in the rare case that the app requires a different Razor language version than the inferred value, a version can be configured by setting the `<RazorLangVersion>` property in the app's project file:

XML

```
<PropertyGroup>
  <RazorLangVersion>{VERSION}</RazorLangVersion>
</PropertyGroup>
```

Razor's language version is tightly integrated with the version of the runtime that it was built for. Targeting a language version that isn't designed for the runtime is unsupported and likely produces build errors.

## Additional resources

- [Additions to the csproj format for .NET Core](#)


- [Common MSBuild project items](#)

# View components in ASP.NET Core

Article • 09/25/2023

By [Rick Anderson](#) 

## View components

View components are similar to partial views, but they're much more powerful. View components don't use model binding, they depend on the data passed when calling the view component. This article was written using controllers and views, but view components work with [Razor Pages](#) .

A view component:

- Renders a chunk rather than a whole response.
- Includes the same separation-of-concerns and testability benefits found between a controller and view.
- Can have parameters and business logic.
- Is typically invoked from a layout page.

View components are intended anywhere reusable rendering logic that's too complex for a partial view, such as:

- Dynamic navigation menus
- Tag cloud, where it queries the database
- Sign in panel
- Shopping cart
- Recently published articles
- Sidebar content on a blog
- A sign in panel that would be rendered on every page and show either the links to sign out or sign in, depending on the sign in state of the user

A view component consists of two parts:

- The class, typically derived from [ViewComponent](#)
- The result it returns, typically a view.

Like controllers, a view component can be a POCO, but most developers take advantage of the methods and properties available by deriving from [ViewComponent](#).

When considering if view components meet an app's specifications, consider using Razor components instead. Razor components also combine markup with C# code to

produce reusable UI units. Razor components are designed for developer productivity when providing client-side UI logic and composition. For more information, see [ASP.NET Core Razor components](#). For information on how to incorporate Razor components into an MVC or Razor Pages app, see [Integrate ASP.NET Core Razor components into ASP.NET Core apps](#).

## Create a view component

This section contains the high-level requirements to create a view component. Later in the article, we'll examine each step in detail and create a view component.

### The view component class

A view component class can be created by any of the following:

- Deriving from [ViewComponent](#)
- Decorating a class with the [\[ViewComponent\]](#) attribute, or deriving from a class with the `[ViewComponent]` attribute
- Creating a class where the name ends with the suffix [ViewComponent](#)

Like controllers, view components must be public, non-nested, and non-abstract classes. The view component name is the class name with the `ViewComponent` suffix removed. It can also be explicitly specified using the [Name](#) property.

A view component class:

- Supports constructor [dependency injection](#)
- Doesn't take part in the controller lifecycle, therefore [filters](#) can't be used in a view component

To prevent a class that has a case-insensitive `ViewComponent` suffix from being treated as a view component, decorate the class with the [\[NonViewComponent\]](#) attribute:

C#

```
using Microsoft.AspNetCore.Mvc;

[NonViewComponent]
public class ReviewComponent
{
    public string Status(string name) => JobStatus.GetCurrentStatus(name);
}
```

# View component methods

A view component defines its logic in an:

- `InvokeAsync` method that returns `Task<IViewComponentResult>`.
- `Invoke` synchronous method that returns an `IViewComponentResult`.

Parameters come directly from invocation of the view component, not from model binding. A view component never directly handles a request. Typically, a view component initializes a model and passes it to a view by calling the `View` method. In summary, view component methods:

- Define an `InvokeAsync` method that returns a `Task<IViewComponentResult>` or a synchronous `Invoke` method that returns an `IViewComponentResult`.
- Typically initializes a model and passes it to a view by calling the `ViewComponent.View` method.
- Parameters come from the calling method, not HTTP. There's no model binding.
- Aren't reachable directly as an HTTP endpoint. They're typically invoked in a view. A view component never handles a request.
- Are overloaded on the signature rather than any details from the current HTTP request.

## View search path

The runtime searches for the view in the following paths:

- `/Views/{Controller Name}/Components/{View Component Name}/{View Name}`
- `/Views/Shared/Components/{View Component Name}/{View Name}`
- `/Pages/Shared/Components/{View Component Name}/{View Name}`
- `/Areas/{Area Name}/Views/Shared/Components/{View Component Name}/{View Name}`

The search path applies to projects using controllers + views and Razor Pages.

The default view name for a view component is `Default`, which means view files will typically be named `Default.cshtml`. A different view name can be specified when creating the view component result or when calling the `View` method.

We recommend naming the view file `Default.cshtml` and using the `Views/Shared/Components/{View Component Name}/{View Name}` path. The `PriorityList` view component used in this sample uses

`Views/Shared/Components/PriorityList/Default.cshtml` for the view component view.

## Customize the view search path

To customize the view search path, modify Razor's `ViewLocationFormats` collection. For example, to search for views within the path `/Components/{View Component Name}/{View Name}`, add a new item to the collection:

C#

```
using Microsoft.EntityFrameworkCore;
using ViewComponentSample.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews()
    .AddRazorOptions(options =>
    {
        options.ViewLocationFormats.Add("/{0}.cshtml");
    });

builder.Services.AddDbContext<ToDoContext>(options =>
    options.UseInMemoryDatabase("db"));

var app = builder.Build();

// Remaining code removed for brevity.
```

In the preceding code, the placeholder `{0}` represents the path `Components/{View Component Name}/{View Name}`.

## Invoke a view component

To use the view component, call the following inside a view:

C#

```
@await Component.InvokeAsync("Name of view component",
    {Anonymous Type Containing Parameters})
```

The parameters are passed to the `InvokeAsync` method. The `PriorityList` view component developed in the article is invoked from the `Views/ToDo/Index.cshtml` view file. In the following code, the `InvokeAsync` method is called with two parameters:

C#

```
</table>
```



```

<div>
    Maximum Priority: @ViewData["maxPriority"] <br />
    Is Complete: @ViewData["isDone"]
    @await Component.InvokeAsync("PriorityList",
        new {
            maxPriority = ViewData["maxPriority"],
            isDone = ViewData["isDone"] }
    )
</div>

```

## Invoke a view component as a Tag Helper

A View Component can be invoked as a [Tag Helper](#):

C#HTML

```

<div>
    Maxium Priority: @ViewData["maxPriority"] <br />
    Is Complete: @ViewData["isDone"]
    @{
        int maxPriority = Convert.ToInt32(ViewData["maxPriority"]);
        bool isDone = Convert.ToBoolean(ViewData["isDone"]);
    }
    <vc:priority-list max-priority=maxPriority is-done=isDone>
    </vc:priority-list>
</div>

```

Pascal-cased class and method parameters for Tag Helpers are translated into their [kebab case](#). The Tag Helper to invoke a view component uses the `<vc></vc>` element. The view component is specified as follows:

C#HTML

```

<vc:[view-component-name]
    parameter1="parameter1 value"
    parameter2="parameter2 value">
</vc:[view-component-name]>

```

To use a view component as a Tag Helper, register the assembly containing the view component using the `@addTagHelper` directive. If the view component is in an assembly called `MyWebApp`, add the following directive to the `_ViewImports.cshtml` file:

C#HTML

```

@addTagHelper *, MyWebApp

```

A view component can be registered as a Tag Helper to any file that references the view component. See [Managing Tag Helper Scope](#) for more information on how to register Tag Helpers.

The `InvokeAsync` method used in this tutorial:

C#HTML

```
@await Component.InvokeAsync("PriorityList",
    new {
        maxPriority = ViewData["maxPriority"],
        isDone = ViewData["isDone"] }
)
```

In the preceding markup, the `PriorityList` view component becomes `priority-list`. The parameters to the view component are passed as attributes in kebab case.

## Invoke a view component directly from a controller

View components are typically invoked from a view, but they can be invoked directly from a controller method. While view components don't define endpoints like controllers, a controller action that returns the content of a `ViewComponentResult` can be implemented.

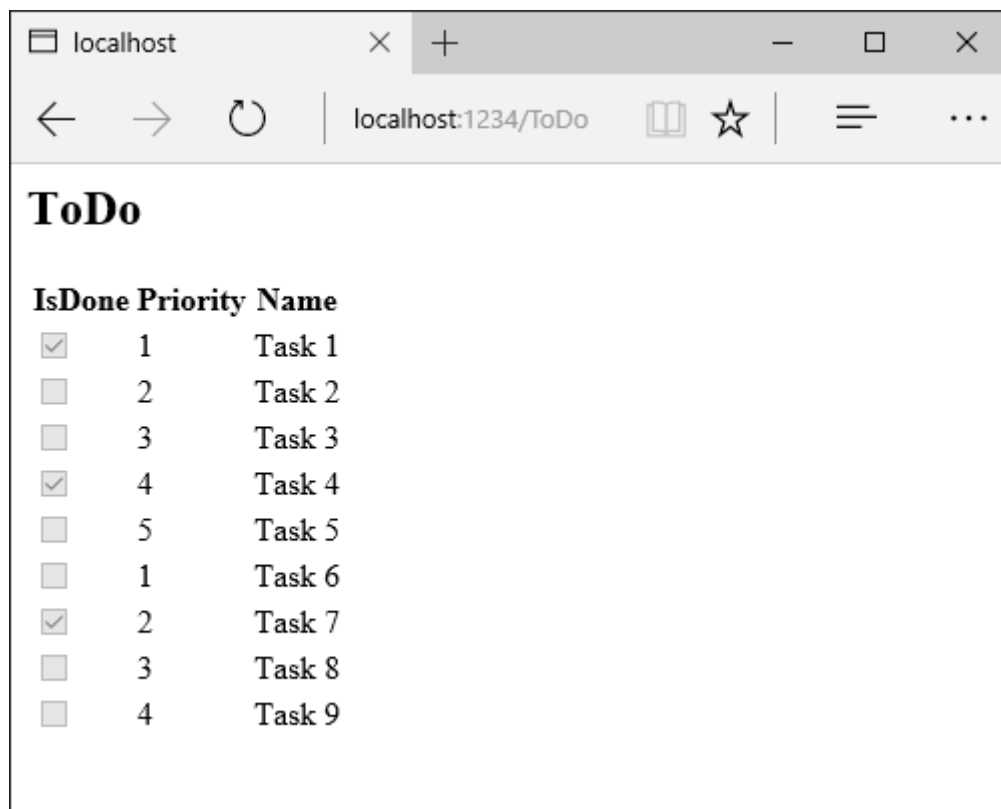
In the following example, the view component is called directly from the controller:

C#

```
public IActionResult IndexVC(int maxPriority = 2, bool isDone = false)
{
    return ViewComponent("PriorityList",
        new {
            maxPriority = maxPriority,
            isDone = isDone
        });
}
```

## Create a basic view component

[Download](#), build and test the starter code. It's a basic project with a `ToDo` controller that displays a list of *ToDo* items.



## Update the controller to pass in priority and completion status

Update the `Index` method to use priority and completion status parameters:

C#

```
using Microsoft.AspNetCore.Mvc;
using ViewComponentSample.Models;

namespace ViewComponentSample.Controllers;
public class ToDoController : Controller
{
    private readonly ToDoContext _ToDoContext;

    public ToDoController(ToDoContext context)
    {
        _ToDoContext = context;
        _ToDoContext.Database.EnsureCreated();
    }

    public IActionResult Index(int maxPriority = 2, bool isDone = false)
    {
        var model = _ToDoContext!.ToDo!.ToList();
        ViewData["maxPriority"] = maxPriority;
        ViewData["isDone"] = isDone;
        return View(model);
    }
}
```

# Add a ViewComponent class

Add a ViewComponent class to `ViewComponents/PriorityListViewComponent.cs`:

C#

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using ViewComponentSample.Models;

namespace ViewComponentSample.ViewComponents;

public class PriorityListViewComponent : ViewComponent
{
    private readonly TodoContext db;

    public PriorityListViewComponent(TodoContext context) => db = context;

    public async Task<IViewComponentResult> InvokeAsync(
        int maxPriority, bool isDone)
    {
        var items = await GetItemsAsync(maxPriority, isDone);
        return View(items);
    }

    private Task<List<TodoItem>> GetItemsAsync(int maxPriority, bool isDone)
    {
        return db!.ToDo!.Where(x => x.IsDone == isDone &&
            x.Priority <= maxPriority).ToListAsync();
    }
}
```

Notes on the code:

- View component classes can be contained in **any** folder in the project.
- Because the class name **PriorityListViewComponent** ends with the suffix **ViewComponent**, the runtime uses the string `PriorityList` when referencing the class component from a view.
- The `[ViewComponent]` attribute can change the name used to reference a view component. For example, the class could have been named `XYZ` with the following `[ViewComponent]` attribute:

C#

```
[ViewComponent(Name = "PriorityList")]
public class XYZ : ViewComponent
```

- The `[ViewComponent]` attribute in the preceding code tells the view component selector to use:
  - The name `PriorityList` when looking for the views associated with the component
  - The string "PriorityList" when referencing the class component from a view.
- The component uses [dependency injection](#) to make the data context available.
- `InvokeAsync` exposes a method that can be called from a view, and it can take an arbitrary number of arguments.
- The `InvokeAsync` method returns the set of `ToDo` items that satisfy the `isDone` and `maxPriority` parameters.

## Create the view component Razor view

- Create the `Views/Shared/Components` folder. This folder **must** be named *Components*.
- Create the `Views/Shared/Components/PriorityList` folder. This folder name must match the name of the view component class, or the name of the class minus the suffix. If the `ViewComponent` attribute is used, the class name would need to match the attribute designation.
- Create a `Views/Shared/Components/PriorityList/Default.cshtml` Razor view:

CSSHTML

```
@model IEnumerable<ViewComponentSample.Models.TODOItem>

<h3>Priority Items</h3>
<ul>
    @foreach (var todo in Model)
    {
        <li>@todo.Name</li>
    }
</ul>
```

The Razor view takes a list of `ToDoItem` and displays them. If the view component `InvokeAsync` method doesn't pass the name of the view, *Default* is used for the view name by convention. To override the default styling for a specific controller, add a view to the controller-specific view folder (for example `Views/ToDo/Components/PriorityList/Default.cshtml`).

If the view component is controller-specific, it can be added to the controller-specific folder. For example, `Views/ToDo/Components/PriorityList/Default.cshtml` is controller-specific.

- Add a `div` containing a call to the priority list component to the bottom of the `Views/ToDo/index.cshtml` file:

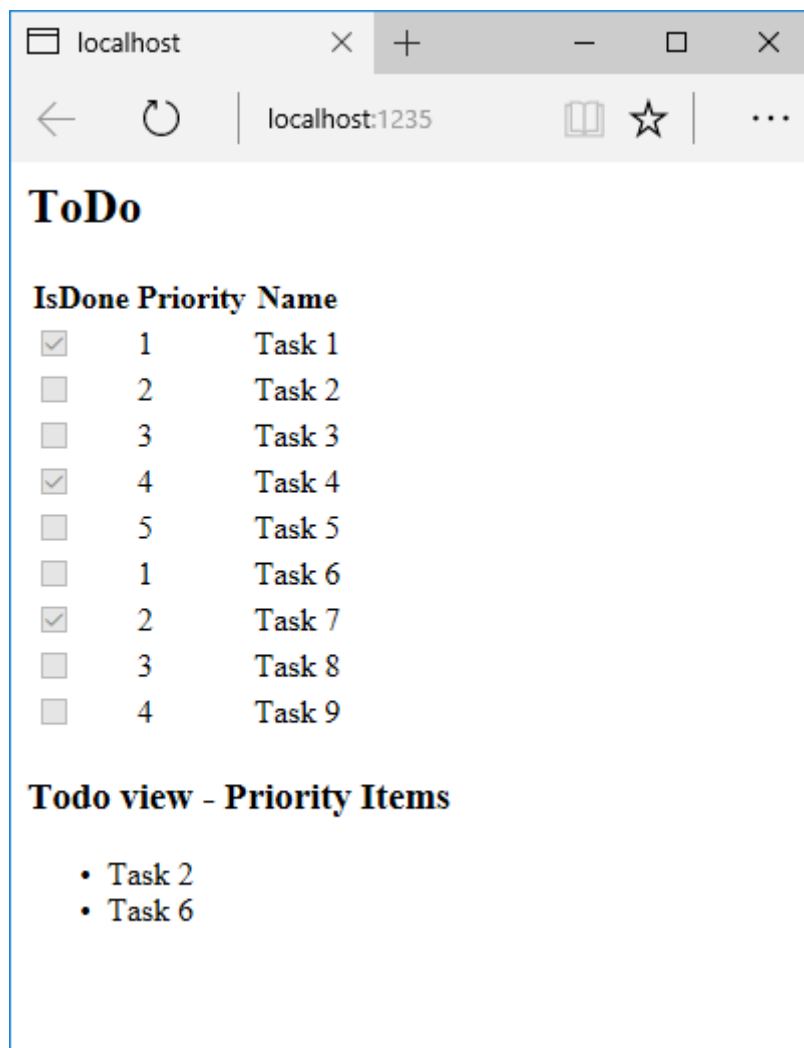
```
CSHTML

</table>

<div>
    Maximum Priority: @ViewData["maxPriority"] <br />
    Is Complete: @ViewData["isDone"]
    @await Component.InvokeAsync("PriorityList",
        new {
            maxPriority = ViewData["maxPriority"],
            isDone = ViewData["isDone"] }
    )
</div>
```

The markup `@await Component.InvokeAsync` shows the syntax for calling view components. The first argument is the name of the component we want to invoke or call. Subsequent parameters are passed to the component. `InvokeAsync` can take an arbitrary number of arguments.

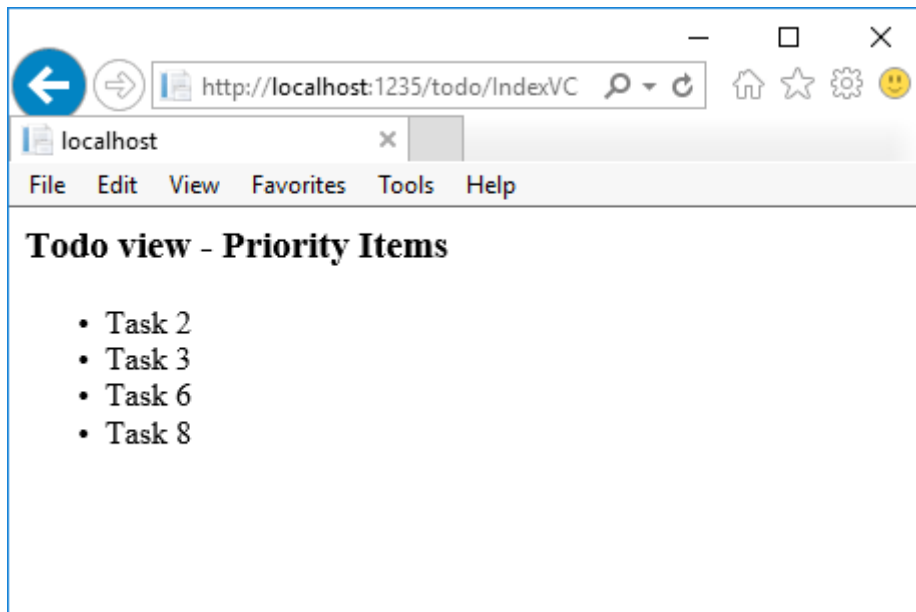
Test the app. The following image shows the ToDo list and the priority items:



The view component can be called directly from the controller:

C#

```
public IActionResult IndexVC(int maxPriority = 2, bool isDone = false)
{
    return ViewComponent("PriorityList",
        new {
            maxPriority = maxPriority,
            isDone = isDone
        });
}
```



## Specify a view component name

A complex view component might need to specify a non-default view under some conditions. The following code shows how to specify the "PVC" view from the `InvokeAsync` method. Update the `InvokeAsync` method in the `PriorityListViewComponent` class.

C#

```
public async Task<IViewComponentResult> InvokeAsync(
    int maxPriority, bool isDone)
{
    string MyView = "Default";
    // If asking for all completed tasks, render with the "PVC" view.
    if (maxPriority > 3 && isDone == true)
    {
        MyView = "PVC";
    }
    var items = await GetItemsAsync(maxPriority, isDone);
    return View(MyView, items);
}
```

Copy the `Views/Shared/Components/PriorityList/Default.cshtml` file to a view named `Views/Shared/Components/PriorityList/PVC.cshtml`. Add a heading to indicate the PVC view is being used.

CSHTML

```
@model IEnumerable<ViewComponentSample.Models.TODOItem>

<h2> PVC Named Priority Component View</h2>
<h4>@ViewBag.PriorityMessage</h4>
```

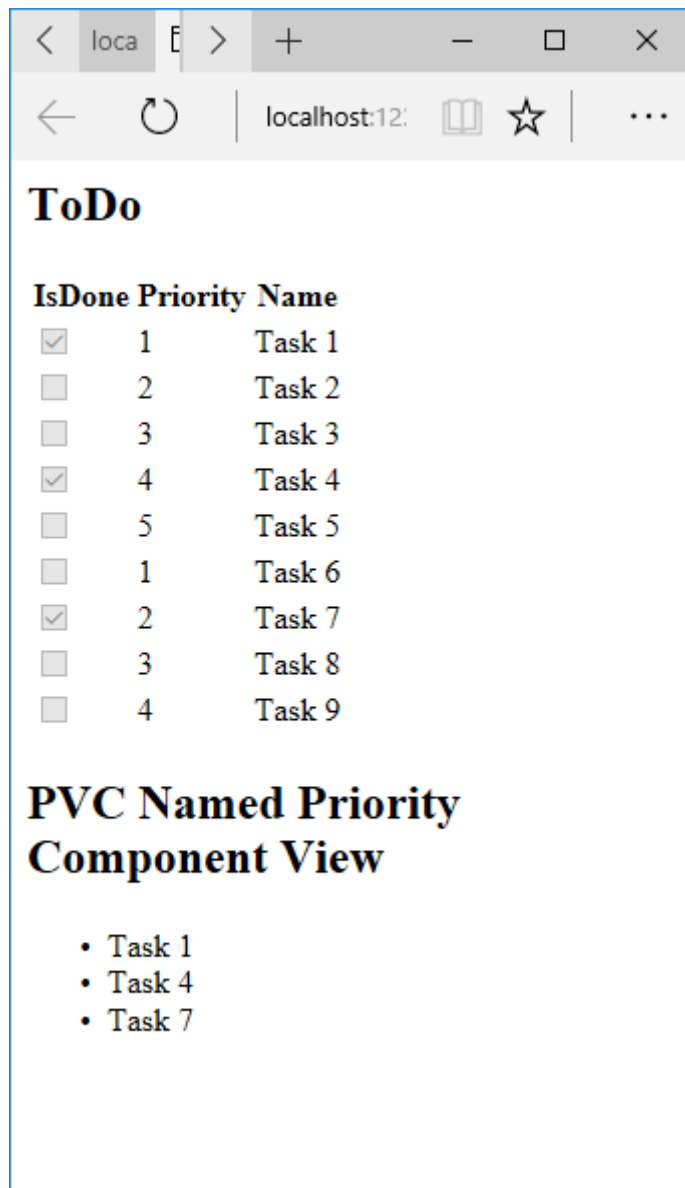


```

<ul>
  @foreach (var todo in Model)
  {
    <li>@todo.Name</li>
  }
</ul>

```

Run the app and verify PVC view.



If the PVC view isn't rendered, verify the view component with a priority of 4 or higher is called.

## Examine the view path

- Change the priority parameter to three or less so the priority view isn't returned.
- Temporarily rename the `Views/ToDo/Components/PriorityList/Default.cshtml` to `1Default.cshtml`.

- Test the app, the following error occurs:

txt

```
An unhandled exception occurred while processing the request.  
InvalidOperationException: The view 'Components/PriorityList/Default'  
wasn't found. The following locations were searched:  
/Views/ToDo/Components/PriorityList/Default.cshtml  
/Views/Shared/Components/PriorityList/Default.cshtml
```

- Copy `Views/ToDo/Components/PriorityList/1Default.cshtml` to `Views/Shared/Components/PriorityList/Default.cshtml`.
- Add some markup to the *Shared* ToDo view component view to indicate the view is from the *Shared* folder.
- Test the **Shared** component view.



## Avoid hard-coded strings

For compile time safety, replace the hard-coded view component name with the class name. Update the *PriorityListViewComponent.cs* file to not use the "ViewComponent" suffix:

C#

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using ViewComponentSample.Models;

namespace ViewComponentSample.ViewComponents;

public class PriorityList : ViewComponent
{
    private readonly TodoContext db;

    public PriorityList(TodoContext context)
    {
        db = context;
    }

    public async Task<IViewComponentResult> InvokeAsync(
        int maxPriority, bool isDone)
    {
        var items = await GetItemsAsync(maxPriority, isDone);
        return View(items);
    }

    private Task<List<TodoItem>> GetItemsAsync(int maxPriority, bool isDone)
    {
        return db!.ToDo!.Where(x => x.IsDone == isDone &&
            x.Priority <= maxPriority).ToListAsync();
    }
}
```

The view file:

CSSHTML

```
</table>

<div>
    Testing nameof(PriorityList) <br />

    Maxium Priority: @ViewData["maxPriority"] <br />
    Is Complete: @ViewData["isDone"]
    @await Component.InvokeAsync(nameof(PriorityList),
        new {
            maxPriority = ViewData["maxPriority"],
            isDone = ViewData["isDone"] }
    )
</div>
```

```
)  
</div>
```

An overload of `Component.InvokeAsync` method that takes a CLR type uses the `typeof` operator:

C#HTML

```
</table>  
  
<div>  
    Testing typeof(PriorityList) <br />  
  
    Maxium Priority: @ViewData["maxPriority"] <br />  
    Is Complete: @ViewData["isDone"]  
    @await Component.InvokeAsync(typeof(PriorityList),  
        new {  
            maxPriority = ViewData["maxPriority"],  
            isDone = ViewData["isDone"] }  
    )  
</div>
```

## Perform synchronous work

The framework handles invoking a synchronous `Invoke` method if asynchronous work isn't required. The following method creates a synchronous `Invoke` view component:

C#

```
using Microsoft.AspNetCore.Mvc;  
using ViewComponentSample.Models;  
  
namespace ViewComponentSample.ViewComponents  
{  
    public class PriorityListSync : ViewComponent  
    {  
        private readonly TodoContext db;  
  
        public PriorityListSync(TodoContext context)  
        {  
            db = context;  
        }  
  
        public IViewComponentResult Invoke(int maxPriority, bool isDone)  
        {  
  
            var x = db!.ToDo!.Where(x => x.IsDone == isDone &&  
                x.Priority <= maxPriority).ToList();  
  
            return View(x);  
        }  
    }  
}
```

```
    }  
  }  
}
```

The view component's Razor file:

C#HTML

```
<div>  
    Testing nameof(PriorityList) <br />  
  
    Maxium Priority: @ViewData["maxPriority"] <br />  
    Is Complete: @ViewData["isDone"]  
    @await Component.InvokeAsync(nameof(PriorityListSync),  
        new {  
            maxPriority = ViewData["maxPriority"],  
            isDone = ViewData["isDone"] }  
    )  
</div>
```

The view component is invoked in a Razor file (for example, `Views/Home/Index.cshtml`) using one of the following approaches:

- [IViewComponentHelper](#)
- [Tag Helper](#)

To use the [IViewComponentHelper](#) approach, call `Component.InvokeAsync`:

C#HTML

```
@await Component.InvokeAsync(nameof(PriorityList),  
    new { maxPriority = 4, isDone = true })
```

To use the Tag Helper, register the assembly containing the View Component using the `@addTagHelper` directive (the view component is in an assembly called `MyWebApp`):

C#HTML

```
@addTagHelper *, MyWebApp
```

Use the view component Tag Helper in the Razor markup file:

C#HTML

```
<vc:priority-list max-priority="999" is-done="false">  
</vc:priority-list>
```

The method signature of `PriorityList.Invoke` is synchronous, but Razor finds and calls the method with `Component.InvokeAsync` in the markup file.

## Additional resources

- [View or download sample code](#) [↗](#) (how to download)
- [Dependency injection into views](#)
- [View Components in Razor Pages](#) [↗](#)
- [Why You Should Use View Components, not Partial Views, in ASP.NET Core](#) [↗](#)

# Razor file compilation in ASP.NET Core

Article • 08/30/2024

Razor files with a `.cshtml` extension are compiled at both build and publish time using the [Razor SDK](#). Runtime compilation may be optionally enabled by configuring the project.

## ⓘ Note

Runtime compilation:

- Isn't supported for Razor components of Blazor apps.
- Doesn't support [global using directives](#).
- Doesn't support [implicit using directives](#).
- Disables [.NET Hot Reload](#).
- Is recommended for development, not for production.

## Razor compilation

Build-time and publish-time compilation of Razor files is enabled by default by the Razor SDK. When enabled, runtime compilation complements build-time compilation, allowing Razor files to be updated if they're edited while the app is running.

Updating Razor views and Razor Pages during development while the app is running is also supported using [.NET Hot Reload](#).

## ⓘ Note

When enabled, runtime compilation disables [.NET Hot Reload](#). We recommend using Hot Reload instead of Razor runtime compilation during development.

## Enable runtime compilation for all environments

To enable runtime compilation for all environments:

1. Install the [Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation](#) [NuGet](#) package.

2. Call [AddRazorRuntimeCompilation](#) in `Program.cs`:

```
C#  
  
var builder = WebApplication.CreateBuilder(args);  
  
builder.Services.AddRazorPages()  
    .AddRazorRuntimeCompilation();
```

## Enable runtime compilation conditionally

Runtime compilation can be enabled conditionally, which ensures that the published output:

- Uses compiled views.
- Doesn't enable file watchers in production.

To enable runtime compilation only for the Development environment:

1. Install the [Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation](#) [NuGet](#) package.
2. Call [AddRazorRuntimeCompilation](#) in `Program.cs` when the current environment is set to Development:

```
C#  
  
var builder = WebApplication.CreateBuilder(args);  
  
var mvcBuilder = builder.Services.AddRazorPages();  
  
if (builder.Environment.IsDevelopment())  
{  
    mvcBuilder.AddRazorRuntimeCompilation();  
}
```

Runtime compilation can also be enabled with a [hosting startup assembly](#). To enable runtime compilation in the Development environment for specific launch profiles:

1. Install the [Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation](#) [NuGet](#) package.
2. Modify the launch profile's `environmentVariables` section in `launchSettings.json`:
  - Verify that `ASPNETCORE_ENVIRONMENT` is set to `"Development"`.



- Set `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES` to `"Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation"`. For example, the following `launchSettings.json` enables runtime compilation for the `ViewCompilationSample` and `IIS Express` launch profiles:

JSON

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:7098",
      "sslPort": 44332
    }
  },
  "profiles": {
    "ViewCompilationSample": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "https://localhost:7173;http://localhost:5251",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development",
        "ASPNETCORE_HOSTINGSTARTUPASSEMBLIES": "Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development",
        "ASPNETCORE_HOSTINGSTARTUPASSEMBLIES": "Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation"
      }
    }
  }
}
```

With this approach, no code changes are needed in `Program.cs`. At runtime, ASP.NET Core searches for an [assembly-level `HostingStartup` attribute](#) in `Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation`. The `HostingStartup` attribute specifies the app startup code to execute and that startup code enables runtime compilation.

# Enable runtime compilation for a Razor class library

Consider a scenario in which a Razor Pages project references a [Razor class library \(RCL\)](#) named *MyClassLib*. The RCL contains a `_Layout.cshtml` file consumed by MVC and Razor Pages projects. To enable runtime compilation for the `_Layout.cshtml` file in that RCL, make the following changes in the Razor Pages project:

1. Enable runtime compilation with the instructions at [Enable runtime compilation conditionally](#).
2. Configure [MvcRazorRuntimeCompilationOptions](#) in `Program.cs`:

```
C#  
  
var builder = WebApplication.CreateBuilder(args);  
  
builder.Services.AddRazorPages();  
  
builder.Services.Configure<MvcRazorRuntimeCompilationOptions>(options  
=>  
{  
    var libraryPath = Path.GetFullPath(  
        Path.Combine(builder.Environment.ContentRootPath, "..",  
            "MyClassLib"));  
  
    options.FileProviders.Add(new PhysicalFileProvider(libraryPath));  
});
```

The preceding code builds an absolute path to the *MyClassLib* RCL. The [PhysicalFileProvider API](#) is used to locate directories and files at that absolute path. Finally, the `PhysicalFileProvider` instance is added to a file providers collection, which allows access to the RCL's `.cshtml` files.

## Additional resources

- [RazorCompileOnBuild and RazorCompileOnPublish](#) properties
- [Introduction to Razor Pages in ASP.NET Core](#)
- [Views in ASP.NET Core MVC](#)
- [ASP.NET Core Razor SDK](#)

# Display and Editor templates in ASP.NET Core

Article • 06/15/2023

By [Alexander Wicht](#)

Display and Editor templates specify the user interface layout of custom types. Consider the following `Address` model:

C#

```
public class Address
{
    public int Id { get; set; }
    public string FirstName { get; set; } = null!;
    public string MiddleName { get; set; } = null!;
    public string LastName { get; set; } = null!;
    public string Street { get; set; } = null!;
    public string City { get; set; } = null!;
    public string State { get; set; } = null!;
    public string Zipcode { get; set; } = null!;
}
```

A project that [scaffolds](#) the `Address` model displays the `Address` in the following form:

## Address

---

<b>FirstName</b>	Rick
<b>MiddleName</b>	M.
<b>LastName</b>	Anderson
<b>Street</b>	123 N 456 W
<b>City</b>	Seattle
<b>State</b>	WA
<b>Zipcode</b>	98009

[Edit](#) | [Back to List](#)

A web site could use a Display Template to show the `Address` in standard format:

## Address DM

---

Rick M. Anderson

123 N 456 W

Seattle WA 98009

[Edit](#) | [Back to List](#)

Display and Editor templates can also reduce code duplication and maintenance costs. Consider a web site that displays the `Address` model on 20 different pages. If the `Address` model changes, the 20 pages will all need to be updated. If a Display Template is used for the `Address` model, only the Display Template needs to be updated. For example, the `Address` model might be updated to include the country or region.

[Tag Helpers](#) provide an alternative way that enables server-side code to participate in creating and rendering HTML elements in Razor files. For more information, see [Tag Helpers compared to HTML Helpers](#).

## Display templates

`DisplayTemplates` customize the display of model fields or create a layer of abstraction between the model values and their display.

A `DisplayTemplate` is a [Razor](#) file placed in the `DisplayTemplates` folder:

- For Razor Pages apps, in the `Pages/Shared/DisplayTemplates` folder.
- For MVC apps, in the `Views/Shared/DisplayTemplates` folder or the `Views/ControllerName/DisplayTemplates` folder. Display templates in the `Views/Shared/DisplayTemplates` are used by all controllers in the app. Display templates in the `Views/ControllerName/DisplayTemplates` folder are resolved only by the `ControllerName` controller.

By convention, the `DisplayTemplate` file is named after the type to be displayed. The `Address.cshtml` template used in this sample:

CSSHTML

```
@model Address

<dl>
  <dd>@Model.FirstName @Model.MiddleName @Model.LastName</dd>
  <dd>@Model.Street</dd>
  <dd>@Model.City @Model.State @Model.Zipcode</dd>
</dl>
```

The view engine automatically looks for a file in the `DisplayTemplates` folder that matches the name of the type. If it doesn't find a matching template, it falls back to the built in templates.

The following code shows the Details view of the scaffolded project:

CSSHTML

```
@page
@model WebAddress.Pages.Adr.DetailsModel

@{
    ViewData["Title"] = "Details";
}

<h1>Details</h1>

<div>
  <h4>Address</h4>
  <hr />
  <dl class="row">
    <dt class="col-sm-2">
      @Html.DisplayNameFor(model => model.Address.FirstName)
    </dt>
    <dd class="col-sm-10">
      @Html.DisplayFor(model => model.Address.FirstName)
    </dd>
    <dt class="col-sm-2">
      @Html.DisplayNameFor(model => model.Address.MiddleName)
    </dt>
    <dd class="col-sm-10">
      @Html.DisplayFor(model => model.Address.MiddleName)
    </dd>
    <dt class="col-sm-2">
      @Html.DisplayNameFor(model => model.Address.LastName)
    </dt>
    <dd class="col-sm-10">
      @Html.DisplayFor(model => model.Address.LastName)
    </dd>
    <dt class="col-sm-2">
      @Html.DisplayNameFor(model => model.Address.Street)
    </dt>
```

```

        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Address.Street)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Address.City)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Address.City)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Address.State)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Address.State)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Address.Zipcode)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Address.Zipcode)
        </dd>
    </dl>
</div>
<div>
    <a asp-page="./Edit" asp-route-id="@Model.Address?.Id">Edit</a> |
    <a asp-page="./Index">Back to List</a>
</div>

```

The following code shows the Details view using the Address Display Template:

CSHTML

```

@page
@model WebAddress.Pages.Adr2.DetailsModel

@{
    ViewData["Title"] = "Details";
}

<h1>Details</h1>

<div>
    <h4>Address DM</h4>
    <hr />
    <dl class="row">
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Address)
        </dd>
    </dl>
</div>
<div>
    <a asp-page="./Edit" asp-route-id="@Model.Address?.Id">Edit</a> |

```

```
<a asp-page="./Index">Back to List</a>
</div>
```

To reference a template whose name doesn't match the type name, use the `templateName` parameter in the [DisplayFor](#) method. For example, the following markup displays the `Address` model with the `AddressShort` template:

CSSHTML

```
@page
@model WebAddress.Pages.Adr2.DetailsCCModel

@{
    ViewData["Title"] = "Details Short";
}

<h1>Details Short</h1>

<div>
    <h4>Address Short</h4>
    <hr />
    <dl class="row">
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Address, "AddressShort")
        </dd>
    </dl>
</div>
```

Use one of the available [DisplayFor overloads](#) that expose the `additionalViewData` parameter to pass additional view data that is merged into the [View Data Dictionary](#) instance created for the template.

## Editor templates

Editor templates are used in form controls when the model is edited or updated.

An `EditorTemplate` is a [Razor](#) file placed in the `EditorTemplates` folder:

- For Razor Pages apps, in the `Pages/Shared/EditorTemplates` folder.
- For MVC apps, in the `Views/Shared/EditorTemplates` folder or the `Views/ControllerName/EditorTemplates` folder.

The following markup shows the `Pages/Shared/EditorTemplates/Address.cshtml` used in the sample:

CSSHTML

```

@model Address

<dl>
    <dd>    Name:
        <input asp-for="FirstName" /> <input asp-for="MiddleName" /> <input
asp-for="LastName" />
    </dd>
    <dd>    Street:
        <input asp-for="Street" />
    </dd>

    <dd>    city/state/zip:
        <input asp-for="City" /> <input asp-for="State" /> <input asp-
for="Zipcode" />
    </dd>

</dl>

```

The following markup shows the *Edit.cshtml* page which uses the `Pages/Shared/EditorTemplates/Address.cshtml` template:

CSHTML

```

@page
@model WebAddress.Pages.Adr.EditModel

@{
    ViewData["Title"] = "Edit";
}

<h1>Edit</h1>

<h4>Address</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger">
            </div>
            <input type="hidden" asp-for="Address.Id" />
            @Html.EditorFor(model => model.Address)
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="./Index">Back to List</a>
</div>

```



```
@section Scripts {  
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}  
}
```

## Additional resources

- [View or download sample code](#) [↗](#) (how to download)
- [Tag Helpers](#)
- [Tag Helpers compared to HTML Helpers](#)

# Upload files in ASP.NET Core

Article • 09/27/2024

By [Rutger Storm](#) 

ASP.NET Core supports uploading one or more files using buffered model binding for smaller files and unbuffered streaming for larger files.

[View or download sample code](#)  ([how to download](#))

## Security considerations

Use caution when providing users with the ability to upload files to a server. Cyberattackers may attempt to:

- Execute [denial of service](#) attacks.
- Upload viruses or malware.
- Compromise networks and servers in other ways.

Security steps that reduce the likelihood of a successful attack are:

- Upload files to a dedicated file upload area, preferably to a non-system drive. A dedicated location makes it easier to impose security restrictions on uploaded files. Disable execute permissions on the file upload location.<sup>†</sup>
- Do **not** persist uploaded files in the same directory tree as the app.<sup>†</sup>
- Use a safe file name determined by the app. Don't use a file name provided by the user or the untrusted file name of the uploaded file.<sup>†</sup> HTML encode the untrusted file name when displaying it. For example, logging the file name or displaying in UI (Razor automatically HTML encodes output).
- Allow only approved file extensions for the app's design specification.<sup>†</sup>
- Verify that client-side checks are performed on the server.<sup>†</sup> Client-side checks are easy to circumvent.
- Check the size of an uploaded file. Set a maximum size limit to prevent large uploads.<sup>†</sup>
- When files shouldn't be overwritten by an uploaded file with the same name, check the file name against the database or physical storage before uploading the file.
- **Run a virus/malware scanner on uploaded content before the file is stored.**


<sup>†</sup>The sample app demonstrates an approach that meets the criteria.

 **Warning**

Uploading malicious code to a system is frequently the first step to executing code that can:

- Completely gain control of a system.
- Overload a system with the result that the system crashes.
- Compromise user or system data.
- Apply graffiti to a public UI.


For information on reducing vulnerabilities when accepting files from users, see the following resources:

- [Unrestricted File Upload](#) 
- [Azure Security: Ensure appropriate controls are in place when accepting files from users](#)

For more information on implementing security measures, including examples from the sample app, see the [Validation](#) section.

## Storage scenarios

Common storage options for files include:

- Database
  - For [small](#) file uploads, a database is often faster than physical storage (file system or network share) options.
  - A database is often more convenient than physical storage options because retrieval of a database record for user data can concurrently supply the file content (for example, an avatar image).
  - A database is potentially less expensive than using a cloud data storage service.
- Physical storage (file system or network share)
  - For large file uploads:
    - Database limits may restrict the size of the upload.
    - Physical storage is often less economical than storage in a database.
  - Physical storage is potentially less expensive than using a cloud data storage service.
  - The app's process must have read and write permissions to the storage location.  
**Never grant execute permission.**
- Cloud data storage service, for example, [Azure Blob Storage](#) .

- Services usually offer improved scalability and resiliency over on-premises solutions that are usually subject to single points of failure.
- Services are potentially lower cost in large storage infrastructure scenarios.

For more information, see [Quickstart: Use .NET to create a blob in object storage](#).

## Small and large files

The definition of small and large files depend on the computing resources available. Apps should benchmark the storage approach used to ensure it can handle the expected sizes. Benchmark memory, CPU, disk, and database performance.

While specific boundaries can't be provided on what is small versus large for your deployment, here are some of ASP.NET Core's related defaults for [FormOptions](#) <sup>(API documentation)</sup>:

- By default, [HttpRequest.Form](#) doesn't buffer the entire request body ([BufferBody](#)), but it does buffer any multipart form files included.
- [MultipartBodyLengthLimit](#) is the maximum size for buffered form files (default: 128 MB).
- [MemoryBufferThreshold](#) indicates the buffering threshold in memory before transitioning to a buffer file on disk (default: 64 KB). `MemoryBufferThreshold` acts as a boundary between small and large files, which is raised or lowered depending on the apps resources and scenarios.

For more information on [FormOptions](#), see the [FormOptions class](#) <sup>in the ASP.NET Core reference source</sup>.

### ⓘ 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 \(dotnet/AspNetCore.Docs #26205\)](#) .

## File upload scenarios

Two general approaches for uploading files are buffering and streaming.

### Buffering

The entire file is read into an [IFormFile](#). `IFormFile` is a C# representation of the file used to process or save the file.

The disk and memory used by file uploads depend on the number and size of concurrent file uploads. If an app attempts to buffer too many uploads, the site crashes when it runs out of memory or disk space. If the size or frequency of file uploads is exhausting app resources, use streaming.

Any single buffered file exceeding 64 KB is moved from memory to a temp file on disk.

Temporary files for larger requests are written to the location named in the `ASPNETCORE_TEMP` environment variable. If `ASPNETCORE_TEMP` is not defined, the files are written to the current user's temporary folder.

Buffering small files is covered in the following sections of this topic:

- [Physical storage](#)
- [Database](#)

## Streaming

The file is received from a multipart request and directly processed or saved by the app. Streaming doesn't improve performance significantly. Streaming reduces the demands for memory or disk space when uploading files.

Streaming large files is covered in the [Upload large files with streaming](#) section.

## Upload small files with buffered model binding to physical storage

To upload small files, use a multipart form or construct a POST request using JavaScript.

The following example demonstrates the use of a Razor Pages form to upload a single file (`Pages/BufferedSingleFileUploadPhysical.cshtml` in the sample app):

C#HTML

```
<form enctype="multipart/form-data" method="post">
  <dl>
    <dt>
      <label asp-for="FileUpload.FormFile"></label>
    </dt>
    <dd>
      <input asp-for="FileUpload.FormFile" type="file" />
      <span asp-validation-for="FileUpload.FormFile"></span>
    </dd>
  </dl>
```

```

    </dl>
    <input asp-page-handler="Upload" class="btn" type="submit"
value="Upload" />
</form>

```

The following example is analogous to the prior example except that:

- JavaScript's ([Fetch API](#)) is used to submit the form's data.
- There's no validation.

CSSHTML

```

<form action="BufferedSingleFileUploadPhysical/?handler=Upload"
    enctype="multipart/form-data" onsubmit="AJAXSubmit(this);return
false;"
    method="post">
    <dl>
        <dt>
            <label for="FileUpload_FormFile">File</label>
        </dt>
        <dd>
            <input id="FileUpload_FormFile" type="file"
                name="FileUpload.FormFile" />
        </dd>
    </dl>

    <input class="btn" type="submit" value="Upload" />

    <div style="margin-top:15px">
        <output name="result"></output>
    </div>
</form>

<script>
    async function AJAXSubmit (oFormElement) {
        var resultElement = oFormElement.elements.namedItem("result");
        const formData = new FormData(oFormElement);

        try {
            const response = await fetch(oFormElement.action, {
                method: 'POST',
                body: formData
            });

            if (response.ok) {
                window.location.href = '/';
            }

            resultElement.value = 'Result: ' + response.status + ' ' +
                response.statusText;
        } catch (error) {
            console.error('Error:', error);
        }
    }

```

```
}  
</script>
```

To perform the form POST in JavaScript for clients that [don't support the Fetch API](#), use one of the following approaches:

- Use a Fetch Polyfill (for example, [window.fetch polyfill \(github/fetch\)](#)).
- Use `XMLHttpRequest`. For example:

JavaScript

```
<script>  
  "use strict";  
  
  function AJAXSubmit (oFormElement) {  
    var oReq = new XMLHttpRequest();  
    oReq.onload = function(e) {  
      oFormElement.elements.namedItem("result").value =  
        'Result: ' + this.status + ' ' + this.statusText;  
    };  
    oReq.open("post", oFormElement.action);  
    oReq.send(new FormData(oFormElement));  
  }  
</script>
```

In order to support file uploads, HTML forms must specify an encoding type ( `enctype` ) of `multipart/form-data`.

For a `files` input element to support uploading multiple files provide the `multiple` attribute on the `<input>` element:

CSHTML

```
<input asp-for="FileUpload.FormFiles" type="file" multiple />
```

The individual files uploaded to the server can be accessed through [Model Binding](#) using `IFormFile`. The sample app demonstrates multiple buffered file uploads for database and physical storage scenarios.

### Warning

Do **not** use the `FileName` property of `IFormFile` other than for display and logging. When displaying or logging, HTML encode the file name. A cyberattacker can

provide a malicious filename, including full paths or relative paths. Applications should:

- Remove the path from the user-supplied filename.
- Save the HTML-encoded, path-removed filename for UI or logging.
- Generate a new random filename for storage.

The following code removes the path from the file name:

C#

```
string untrustedFileName = Path.GetFileName(pathName);
```

The examples provided thus far don't take into account security considerations. Additional information is provided by the following sections and the [sample app](#) <sup>↗</sup>:

- [Security considerations](#)
- [Validation](#)

When uploading files using model binding and `IFormFile`, the action method can accept:

- A single `IFormFile`.
- Any of the following collections that represent several files:
  - `IFormFileCollection`
  - `IEnumerable<IFormFile>`
  - `List<IFormFile>`

#### ⓘ Note

Binding matches form files by name. For example, the HTML `name` value in `<input type="file" name="formFile">` must match the C# parameter/property bound (`FormFile`). For more information, see the [Match name attribute value to parameter name of POST method](#) section.

The following example:

- Loops through one or more uploaded files.
- Uses `Path.GetTempFileName` to return a full path for a file, including the file name.
- Saves the files to the local file system using a file name generated by the app.
- Returns the total number and size of files uploaded.

C#



```

public async Task<IActionResult> OnPostUploadAsync(List<IFormFile> files)
{
    long size = files.Sum(f => f.Length);

    foreach (var formFile in files)
    {
        if (formFile.Length > 0)
        {
            var filePath = Path.GetTempFileName();

            using (var stream = System.IO.File.Create(filePath))
            {
                await formFile.CopyToAsync(stream);
            }
        }
    }

    // Process uploaded files
    // Don't rely on or trust the FileName property without validation.

    return Ok(new { count = files.Count, size });
}

```

Use `Path.GetRandomFileName` to generate a file name without a path. In the following example, the path is obtained from configuration:

```

C#

foreach (var formFile in files)
{
    if (formFile.Length > 0)
    {
        var filePath = Path.Combine(_config["StoredFilesPath"],
            Path.GetRandomFileName());

        using (var stream = System.IO.File.Create(filePath))
        {
            await formFile.CopyToAsync(stream);
        }
    }
}

```

The path passed to the `FileStream` *must* include the file name. If the file name isn't provided, an `UnauthorizedAccessException` is thrown at runtime.

Files uploaded using the `IFormFile` technique are buffered in memory or on disk on the server before processing. Inside the action method, the `IFormFile` contents are accessible as a `Stream`. In addition to the local file system, files can be saved to a network share or to a file storage service, such as [Azure Blob storage](#).

For another example that loops over multiple files for upload and uses safe file names, see `Pages/BufferedMultipleFileUploadPhysical.cshtml.cs` in the sample app.

### Warning

`Path.GetTempFileName` throws an [IOException](#) if more than 65,535 files are created without deleting previous temporary files. The limit of 65,535 files is a per-server limit. For more information on this limit on Windows OS, see the remarks in the following topics:

- [GetTempFileNameA function](#)
- [GetTempFileName](#)

## Upload small files with buffered model binding to a database

To store binary file data in a database using [Entity Framework](#), define a [Byte](#) array property on the entity:

C#

```
public class AppFile
{
    public int Id { get; set; }
    public byte[] Content { get; set; }
}
```

Specify a page model property for the class that includes an [IFormFile](#):

C#

```
public class BufferedSingleFileUploadDbModel : PageModel
{
    ...

    [BindProperty]
    public BufferedSingleFileUploadDb FileUpload { get; set; }

    ...
}

public class BufferedSingleFileUploadDb
{
    [Required]
    [Display(Name="File")]
}
```

```
public IFormFile FormFile { get; set; }  
}
```

### ⓘ Note

**IFormFile** can be used directly as an action method parameter or as a bound model property. The prior example uses a bound model property.

The `FileUpload` is used in the Razor Pages form:

CSSHTML

```
<form enctype="multipart/form-data" method="post">  
  <dl>  
    <dt>  
      <label asp-for="FileUpload.FormFile"></label>  
    </dt>  
    <dd>  
      <input asp-for="FileUpload.FormFile" type="file">  
    </dd>  
  </dl>  
  <input asp-page-handler="Upload" class="btn" type="submit"  
value="Upload">  
</form>
```

When the form is POSTed to the server, copy the `IFormFile` to a stream and save it as a byte array in the database. In the following example, `_dbContext` stores the app's database context:

C#

```
public async Task<IActionResult> OnPostUploadAsync()  
{  
    using (var memoryStream = new MemoryStream())  
    {  
        await FileUpload.FormFile.CopyToAsync(memoryStream);  
  
        // Upload the file if less than 2 MB  
        if (memoryStream.Length < 2097152)  
        {  
            var file = new AppFile()  
            {  
                Content = memoryStream.ToArray()  
            };  
  
            _dbContext.File.Add(file);  
  
            await _dbContext.SaveChangesAsync();  
        }  
    }  
}
```

```

    }
    else
    {
        ModelState.AddModelError("File", "The file is too large.");
    }
}

return Page();
}

```

The preceding example is similar to a scenario demonstrated in the sample app:

- `Pages/BufferedSingleFileUploadDb.cshtml`
- `Pages/BufferedSingleFileUploadDb.cshtml.cs`

### ⚠ Warning

Use caution when storing binary data in relational databases, as it can adversely impact performance.

Don't rely on or trust the `FileName` property of `IFormFile` without validation. The `FileName` property should only be used for display purposes and only after HTML encoding.

The examples provided don't take into account security considerations. Additional information is provided by the following sections and the [sample app](#) <sup>↗</sup>:

- [Security considerations](#)
- [Validation](#)

## Upload large files with streaming

The [3.1 example](#) <sup>↗</sup> demonstrates how to use JavaScript to stream a file to a controller action. The file's antiforgery token is generated using a custom filter attribute and passed to the client HTTP headers instead of in the request body. Because the action method processes the uploaded data directly, form model binding is disabled by another custom filter. Within the action, the form's contents are read using a `MultipartReader`, which reads each individual `MultipartSection`, processing the file or storing the contents as appropriate. After the multipart sections are read, the action performs its own model binding.

The initial page response loads the form and saves an antiforgery token in a cookie (via the `GenerateAntiforgeryTokenCookieAttribute` attribute). The attribute uses ASP.NET

Core's built-in [antiforgery support](#) to set a cookie with a request token:

C#

```
public class GenerateAntiforgeryTokenCookieAttribute : ResultFilterAttribute
{
    public override void OnResultExecuting(ResultExecutingContext context)
    {
        var antiforgery =
            context.HttpContext.RequestServices.GetService<IAntiforgery>();

        // Send the request token as a JavaScript-readable cookie
        var tokens = antiforgery.GetAndStoreTokens(context.HttpContext);

        context.HttpContext.Response.Cookies.Append(
            "RequestVerificationToken",
            tokens.RequestToken,
            new CookieOptions() { HttpOnly = false });
    }

    public override void OnResultExecuted(ResultExecutedContext context)
    {
    }
}
```

The `DisableFormValueModelBindingAttribute` is used to disable model binding:

C#

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public class DisableFormValueModelBindingAttribute : Attribute,
    IResourceFilter
{
    public void OnResourceExecuting(ResourceExecutingContext context)
    {
        var factories = context.ValueProviderFactories;
        factories.RemoveType<FormValueProviderFactory>();
        factories.RemoveType<FormFileValueProviderFactory>();
        factories.RemoveType<JQueryFormValueProviderFactory>();
    }

    public void OnResourceExecuted(ResourceExecutedContext context)
    {
    }
}
```

In the sample app, `GenerateAntiforgeryTokenCookieAttribute` and `DisableFormValueModelBindingAttribute` are applied as filters to the page application models of `/StreamedSingleFileUploadDb` and `/StreamedSingleFileUploadPhysical` in `Startup.ConfigureServices` using [Razor Pages conventions](#):

C#

```
services.AddRazorPages(options =>
{
    options.Conventions
        .AddPageApplicationModelConvention("/StreamedSingleFileUploadDb",
            model =>
            {
                model.Filters.Add(
                    new GenerateAntiforgeryTokenCookieAttribute());
                model.Filters.Add(
                    new DisableFormValueModelBindingAttribute());
            });
    options.Conventions
        .AddPageApplicationModelConvention("/StreamedSingleFileUploadPhysical",
            model =>
            {
                model.Filters.Add(
                    new GenerateAntiforgeryTokenCookieAttribute());
                model.Filters.Add(
                    new DisableFormValueModelBindingAttribute());
            });
});
```

Since model binding doesn't read the form, parameters that are bound from the form don't bind (query, route, and header continue to work). The action method works directly with the `Request` property. A `MultipartReader` is used to read each section. Key/value data is stored in a `KeyValueAccumulator`. After the multipart sections are read, the contents of the `KeyValueAccumulator` are used to bind the form data to a model type.

The complete `StreamingController.UploadDatabase` method for streaming to a database with EF Core:

C#

```
[HttpPost]
[DisableFormValueModelBinding]
[ValidateAntiForgeryToken]
public async Task<IActionResult> UploadDatabase()
{
    if (!MultipartRequestHelper.IsMultipartContentType(Request.ContentType))
    {
        ModelState.AddModelError("File",
            $"The request couldn't be processed (Error 1).");
        // Log error

        return BadRequest(ModelState);
    }
}
```

```

    // Accumulate the form data key-value pairs in the request
    (formAccumulator).
    var formAccumulator = new KeyValueAccumulator();
    var trustedFileNameForDisplay = string.Empty;
    var untrustedFileNameForStorage = string.Empty;
    var streamedFileContent = Array.Empty<byte>();

    var boundary = MultipartRequestHelper.GetBoundary(
        MediaTypeHeaderValue.Parse(Request.ContentType),
        _defaultFormOptions.MultipartBoundaryLengthLimit);
    var reader = new MultipartReader(boundary, HttpContext.Request.Body);

    var section = await reader.ReadNextSectionAsync();

    while (section != null)
    {
        var hasContentDispositionHeader =
            ContentDispositionHeaderValue.TryParse(
                section.ContentDisposition, out var contentDisposition);

        if (hasContentDispositionHeader)
        {
            if (MultipartRequestHelper
                .HasFileContentDisposition(contentDisposition))
            {
                untrustedFileNameForStorage =
contentDisposition.FileName.Value;
                // Don't trust the file name sent by the client. To display
                // the file name, HTML-encode the value.
                trustedFileNameForDisplay = WebUtility.HtmlEncode(
                    contentDisposition.FileName.Value);

                streamedFileContent =
                    await FileHelpers.ProcessStreamedFile(section,
contentDisposition,
                        ModelState, _permittedExtensions, _fileSizeLimit);

                if (!ModelState.IsValid)
                {
                    return BadRequest(ModelState);
                }
            }
            else if (MultipartRequestHelper
                .HasFormDataContentDisposition(contentDisposition))
            {
                // Don't limit the key name length because the
                // multipart headers length limit is already in effect.
                var key = HeaderUtilities
                    .RemoveQuotes(contentDisposition.Name).Value;
                var encoding = GetEncoding(section);

                if (encoding == null)
                {
                    ModelState.AddModelError("File",

```

```

        $"The request couldn't be processed (Error 2).");
        // Log error

        return BadRequest(ModelState);
    }

    using (var streamReader = new StreamReader(
        section.Body,
        encoding,
        detectEncodingFromByteOrderMarks: true,
        bufferSize: 1024,
        leaveOpen: true))
    {
        // The value length limit is enforced by
        // MultipartBodyLengthLimit
        var value = await streamReader.ReadToEndAsync();

        if (string.Equals(value, "undefined",
            StringComparison.OrdinalIgnoreCase))
        {
            value = string.Empty;
        }

        formAccumulator.Append(key, value);

        if (formAccumulator.ValueCount >
            _defaultFormOptions.ValueCountLimit)
        {
            // Form key count limit of
            // _defaultFormOptions.ValueCountLimit
            // is exceeded.
            ModelState.AddModelError("File",
                $"The request couldn't be processed (Error
3).");

            // Log error

            return BadRequest(ModelState);
        }
    }
}

// Drain any remaining section body that hasn't been consumed and
// read the headers for the next section.
section = await reader.ReadNextSectionAsync();
}

// Bind form data to the model
var formData = new FormData();
var formValueProvider = new FormValueProvider(
    BindingSource.Form,
    new FormCollection(formAccumulator.GetResults()),
    CultureInfo.CurrentCulture);
var bindingSuccessful = await TryUpdateModelAsync(formData, prefix: "",
    valueProvider: formValueProvider);

```



```

if (!bindingSuccessful)
{
    ModelState.AddModelError("File",
        "The request couldn't be processed (Error 5).");
    // Log error

    return BadRequest(ModelState);
}

// **WARNING!**
// In the following example, the file is saved without
// scanning the file's contents. In most production
// scenarios, an anti-virus/anti-malware scanner API
// is used on the file before making the file available
// for download or for use by other systems.
// For more information, see the topic that accompanies
// this sample app.

var file = new AppFile()
{
    Content = streamedFileContent,
    UntrustedName = untrustedFileNameForStorage,
    Note = formData.Note,
    Size = streamedFileContent.Length,
    UploadDT = DateTime.UtcNow
};

_context.File.Add(file);
await _context.SaveChangesAsync();

return Created(nameof(StreamingController), null);
}

```

MultipartRequestHelper (Utilities/MultipartRequestHelper.cs):

C#

```

using System;
using System.IO;
using Microsoft.Net.Http.Headers;

namespace SampleApp.Utilities
{
    public static class MultipartRequestHelper
    {
        // Content-Type: multipart/form-data; boundary="----
        WebKitFormBoundarymx2fSWqWsd00xQqq"
        // The spec at https://tools.ietf.org/html/rfc2046#section-5.1
        states that 70 characters is a reasonable limit.
        public static string GetBoundary(MediaTypeHeaderValue contentType,
            int lengthLimit)
        {

```

```

        var boundary =
HeaderUtilities.RemoveQuotes(contentType.Boundary).Value;

        if (string.IsNullOrEmpty(boundary))
        {
            throw new InvalidDataException("Missing content-type
boundary.");
        }

        if (boundary.Length > lengthLimit)
        {
            throw new InvalidDataException(
                $"Multipart boundary length limit {lengthLimit}
exceeded.");
        }

        return boundary;
    }

    public static bool IsMultipartContentType(string contentType)
    {
        return !string.IsNullOrEmpty(contentType)
            && contentType.IndexOf("multipart/",
StringComparison.OrdinalIgnoreCase) >= 0;
    }

    public static bool
HasFormDataContentDisposition(ContentDispositionHeaderValue
contentDisposition)
    {
        // Content-Disposition: form-data; name="key";
        return contentDisposition != null
            && contentDisposition.DispositionType.Equals("form-data")
            && string.IsNullOrEmpty(contentDisposition.FileName.Value)
            &&
string.IsNullOrEmpty(contentDisposition.FileNameStar.Value);
    }

    public static bool
HasFileContentDisposition(ContentDispositionHeaderValue contentDisposition)
    {
        // Content-Disposition: form-data; name="myfile1";
filename="Misc 002.jpg"
        return contentDisposition != null
            && contentDisposition.DispositionType.Equals("form-data")
            && (!string.IsNullOrEmpty(contentDisposition.FileName.Value)
                ||
!string.IsNullOrEmpty(contentDisposition.FileNameStar.Value));
    }
}

```

The complete `StreamingController.UploadPhysical` method for streaming to a physical location:

C#

```
[HttpPost]
[DisableFormValueModelBinding]
[ValidateAntiForgeryToken]
public async Task<IActionResult> UploadPhysical()
{
    if (!MultipartRequestHelper.IsMultipartContentType(Request.ContentType))
    {
        ModelState.AddModelError("File",
            $"The request couldn't be processed (Error 1).");
        // Log error

        return BadRequest(ModelState);
    }

    var boundary = MultipartRequestHelper.GetBoundary(
        MediaTypeHeaderValue.Parse(Request.ContentType),
        _defaultFormOptions.MultipartBoundaryLengthLimit);
    var reader = new MultipartReader(boundary, HttpContext.Request.Body);
    var section = await reader.ReadNextSectionAsync();

    while (section != null)
    {
        var hasContentDispositionHeader =
            ContentDispositionHeaderValue.TryParse(
                section.ContentDisposition, out var contentDisposition);

        if (hasContentDispositionHeader)
        {
            // This check assumes that there's a file
            // present without form data. If form data
            // is present, this method immediately fails
            // and returns the model error.
            if (!MultipartRequestHelper
                .HasFileContentDisposition(contentDisposition))
            {
                ModelState.AddModelError("File",
                    $"The request couldn't be processed (Error 2).");
                // Log error

                return BadRequest(ModelState);
            }
        }
        else
        {
            // Don't trust the file name sent by the client. To display
            // the file name, HTML-encode the value.
            var trustedFileNameForDisplay = WebUtility.HtmlEncode(
                contentDisposition.FileName.Value);
            var trustedFileNameForFileStorage =
                Path.GetRandomFileName();
        }
    }
}
```

```

        // **WARNING!**
        // In the following example, the file is saved without
        // scanning the file's contents. In most production
        // scenarios, an anti-virus/anti-malware scanner API
        // is used on the file before making the file available
        // for download or for use by other systems.
        // For more information, see the topic that accompanies
        // this sample.

        var streamedFileContent = await
FileHelpers.ProcessStreamedFile(
            section, contentDisposition, ModelState,
            _permittedExtensions, _fileSizeLimit);

        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }

        using (var targetStream = System.IO.File.Create(
            Path.Combine(_targetFilePath,
trustedFileNameForFileStorage)))
        {
            await targetStream.WriteAsync(streamedFileContent);

            _logger.LogInformation(
                "Uploaded file '{TrustedFileNameForDisplay}' saved
to " +
                "'{TargetFilePath}' as
{TrustedFileNameForFileStorage}",
                trustedFileNameForDisplay, _targetFilePath,
                trustedFileNameForFileStorage);
        }
    }

    // Drain any remaining section body that hasn't been consumed and
    // read the headers for the next section.
    section = await reader.ReadNextSectionAsync();
}

return Created(nameof(StreamingController), null);
}

```

In the sample app, validation checks are handled by `FileHelpers.ProcessStreamedFile`.

## Validation

The sample app's `FileHelpers` class demonstrates several checks for buffered `IFormFile` and streamed file uploads. For processing `IFormFile` buffered file uploads in the sample

app, see the `ProcessFormFile` method in the `Utilities/FileHelpers.cs` file. For processing streamed files, see the `ProcessStreamedFile` method in the same file.

### ⚠ Warning

The validation processing methods demonstrated in the sample app don't scan the content of uploaded files. In most production scenarios, a virus/malware scanner API is used on the file before making the file available to users or other systems.

Although the topic sample provides a working example of validation techniques, don't implement the `FileHelpers` class in a production app unless you:

- Fully understand the implementation.
- Modify the implementation as appropriate for the app's environment and specifications.

**Never indiscriminately implement security code in an app without addressing these requirements.**

## Content validation

**Use a third party virus/malware scanning API on uploaded content.**

Scanning files is demanding on server resources in high volume scenarios. If request processing performance is diminished due to file scanning, consider offloading the scanning work to a [background service](#), possibly a service running on a server different from the app's server. Typically, uploaded files are held in a quarantined area until the background virus scanner checks them. When a file passes, the file is moved to the normal file storage location. These steps are usually performed in conjunction with a database record that indicates the scanning status of a file. By using such an approach, the app and app server remain focused on responding to requests.

## File extension validation

The uploaded file's extension should be checked against a list of permitted extensions. For example:

C#

```
private string[] permittedExtensions = { ".txt", ".pdf" };

var ext = Path.GetExtension(uploadedFileName).ToLowerInvariant();
```

```
if (string.IsNullOrEmpty(ext) || !permittedExtensions.Contains(ext))
{
    // The extension is invalid ... discontinue processing the file
}
```

## File signature validation

A file's signature is determined by the first few bytes at the start of a file. These bytes can be used to indicate if the extension matches the content of the file. The sample app checks file signatures for a few common file types. In the following example, the file signature for a JPEG image is checked against the file:

C#

```
private static readonly Dictionary<string, List<byte[]>> _fileSignature =
    new Dictionary<string, List<byte[]>>
{
    { ".jpeg", new List<byte[]>
        {
            new byte[] { 0xFF, 0xD8, 0xFF, 0xE0 },
            new byte[] { 0xFF, 0xD8, 0xFF, 0xE2 },
            new byte[] { 0xFF, 0xD8, 0xFF, 0xE3 },
        }
    },
};

using (var reader = new BinaryReader(uploadedFileData))
{
    var signatures = _fileSignature[ext];
    var headerBytes = reader.ReadBytes(signatures.Max(m => m.Length));

    return signatures.Any(signature =>
        headerBytes.Take(signature.Length).SequenceEqual(signature));
}
```

To obtain additional file signatures, use a [file signatures database \(Google search result\)](#) and official file specifications. Consulting official file specifications may ensure that the selected signatures are valid.

## File name security

Never use a client-supplied file name for saving a file to physical storage. Create a safe file name for the file using [Path.GetRandomFileName](#) or [Path.GetTempFileName](#) to create a full path (including the file name) for temporary storage.

Razor automatically HTML encodes property values for display. The following code is safe to use:

CSSHTML

```
@foreach (var file in Model.DatabaseFiles) {  
    <tr>  
        <td>  
            @file.UntrustedName  
        </td>  
    </tr>  
}
```

Outside of Razor, always [HtmlEncode](#) file name content from a user's request.

Many implementations must include a check that the file exists; otherwise, the file is overwritten by a file of the same name. Supply additional logic to meet your app's specifications.

## Size validation

Limit the size of uploaded files.

In the sample app, the size of the file is limited to 2 MB (indicated in bytes). The limit is supplied via [Configuration](#) from the `appsettings.json` file:

JSON

```
{  
  "FileSizeLimit": 2097152  
}
```

The `FileSizeLimit` is injected into `PageModel` classes:

C#

```
public class BufferedSingleFileUploadPhysicalModel : PageModel  
{  
    private readonly long _fileSizeLimit;  
  
    public BufferedSingleFileUploadPhysicalModel(IConfiguration config)  
    {  
        _fileSizeLimit = config.GetValue<long>("FileSizeLimit");  
    }  
}
```

```
...  
}
```

When a file size exceeds the limit, the file is rejected:

C#

```
if (formFile.Length > _fileSizeLimit)  
{  
    // The file is too large ... discontinue processing the file  
}
```

## Match name attribute value to parameter name of POST method

In non-Razor forms that POST form data or use JavaScript's `FormData` directly, the name specified in the form's element or `FormData` must match the name of the parameter in the controller's action.

In the following example:

- When using an `<input>` element, the `name` attribute is set to the value `battlePlans`:

HTML

```
<input type="file" name="battlePlans" multiple>
```

- When using `FormData` in JavaScript, the name is set to the value `battlePlans`:

JavaScript

```
var formData = new FormData();  
  
for (var file in files) {  
    formData.append("battlePlans", file, file.name);  
}
```

Use a matching name for the parameter of the C# method (`battlePlans`):

- For a Razor Pages page handler method named `Upload`:

C#



```
public async Task<IActionResult> OnPostUploadAsync(List<IFormFile> battlePlans)
```

- For an MVC POST controller action method:

C#

```
public async Task<IActionResult> Post(List<IFormFile> battlePlans)
```

## Server and app configuration

### Multipart body length limit

[MultipartBodyLengthLimit](#) sets the limit for the length of each multipart body. Form sections that exceed this limit throw an [InvalidDataException](#) when parsed. The default is 134,217,728 (128 MB). Customize the limit using the [MultipartBodyLengthLimit](#) setting in `Startup.ConfigureServices`:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<FormOptions>(options =>
    {
        // Set the limit to 256 MB
        options.MultipartBodyLengthLimit = 268435456;
    });
}
```

[RequestFormLimitsAttribute](#) is used to set the [MultipartBodyLengthLimit](#) for a single page or action.

In a Razor Pages app, apply the filter with a [convention](#) in `Startup.ConfigureServices`:

C#

```
services.AddRazorPages(options =>
{
    options.Conventions
        .AddPageApplicationModelConvention("/FileUploadPage",
            model.Filters.Add(
                new RequestFormLimitsAttribute()
                {
                    // Set the limit to 256 MB
                }
            )
        );
});
```

```

        MultipartBodyLengthLimit = 268435456
    });
});

```

In a Razor Pages app or an MVC app, apply the filter to the page model or action method:

```

C#

// Set the limit to 256 MB
[RequestFormLimits(MultipartBodyLengthLimit = 268435456)]
public class BufferedSingleFileUploadPhysicalModel : PageModel
{
    ...
}

```

## Kestrel maximum request body size

For apps hosted by Kestrel, the default maximum request body size is 30,000,000 bytes, which is approximately 28.6 MB. Customize the limit using the [MaxRequestBodySize](#) Kestrel server option:

```

C#

public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.ConfigureKestrel((context, options) =>
            {
                // Handle requests up to 50 MB
                options.Limits.MaxRequestBodySize = 52428800;
            })
            .UseStartup<Startup>();
        });

```

[RequestSizeLimitAttribute](#) is used to set the [MaxRequestBodySize](#) for a single page or action.

In a Razor Pages app, apply the filter with a [convention](#) in `Startup.ConfigureServices`:

```

C#

services.AddRazorPages(options =>
{
    options.Conventions
        .AddPageApplicationModelConvention("/FileUploadPage",

```

```

        model =>
        {
            // Handle requests up to 50 MB
            model.Filters.Add(
                new RequestSizeLimitAttribute(52428800));
        });
    });

```

In a Razor pages app or an MVC app, apply the filter to the page handler class or action method:

```

C#

// Handle requests up to 50 MB
[RequestSizeLimit(52428800)]
public class BufferedSingleFileUploadPhysicalModel : PageModel
{
    ...
}

```

The `RequestSizeLimitAttribute` can also be applied using the `@attribute` Razor directive:

```

CSHTML

@attribute [RequestSizeLimitAttribute(52428800)]

```

## Other Kestrel limits

Other Kestrel limits may apply for apps hosted by Kestrel:

- [Maximum client connections](#)
- [Request and response data rates](#)

## IIS

The default request limit (`maxAllowedContentLength`) is 30,000,000 bytes, which is approximately 28.6 MB. Customize the limit in the `web.config` file. In the following example, the limit is set to 50 MB (52,428,800 bytes):

```

XML

<system.webServer>
  <security>
    <requestFiltering>

```

```
<requestLimits maxAllowedContentLength="52428800" />
</requestFiltering>
</security>
</system.webServer>
```

The `maxAllowedContentLength` setting only applies to IIS. For more information, see [Request Limits <requestLimits>](#).

## Troubleshoot

Below are some common problems encountered when working with uploading files and their possible solutions.

### Not Found error when deployed to an IIS server

The following error indicates that the uploaded file exceeds the server's configured content length:

```
HTTP 404.13 - Not Found
The request filtering module is configured to deny a request that exceeds
the request content length.
```

For more information, see the [IIS](#) section.

### Connection failure

A connection error and a reset server connection probably indicates that the uploaded file exceeds Kestrel's maximum request body size. For more information, see the [Kestrel maximum request body size](#) section. Kestrel client connection limits may also require adjustment.

### Null Reference Exception with IFormFile

If the controller is accepting uploaded files using [IFormFile](#) but the value is `null`, confirm that the HTML form is specifying an `enctype` value of `multipart/form-data`. If this attribute isn't set on the `<form>` element, the file upload doesn't occur and any bound [IFormFile](#) arguments are `null`. Also confirm that the [upload naming in form data matches the app's naming](#).

## Stream was too long

The examples in this topic rely upon [MemoryStream](#) to hold the uploaded file's content. The size limit of a `MemoryStream` is `int.MaxValue`. If the app's file upload scenario requires holding file content larger than 50 MB, use an alternative approach that doesn't rely upon a single `MemoryStream` for holding an uploaded file's content.

## Additional resources

- [HTTP connection request draining](#)
- [Unrestricted File Upload](#) ↗
- [Azure Security: Security Frame: Input Validation | Mitigations](#)
- [Azure Cloud Design Patterns: Valet Key pattern](#)

# ASP.NET Core Web SDK

Article • 04/10/2024

## Overview

`Microsoft.NET.Sdk.Web` is an [MSBuild project SDK](#) for building ASP.NET Core apps. It's possible to build an ASP.NET Core app without this SDK, however, the Web SDK is:

- Tailored towards providing a first-class experience.
- The recommended target for most users.

Use the Web.SDK in a project:


XML

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <!-- omitted for brevity -->
</Project>
```

Features enabled by using the Web SDK:

- Implicitly references:
  - The [ASP.NET Core shared framework](#).
  - [Analyzers](#) designed for building ASP.NET Core apps.
- The Web SDK imports MSBuild targets that enable the use of publish profiles and publishing using WebDeploy.

## Properties

 Expand table

Property	Description
<code>DisableImplicitFrameworkReferences</code>	Disables implicit reference to the <code>Microsoft.AspNetCore.App</code> shared framework.
<code>DisableImplicitAspNetCoreAnalyzers</code>	Disables implicit reference to ASP.NET Core analyzers.
<code>DisableImplicitComponentsAnalyzers</code>	Disables implicit reference to Razor Components analyzers when building Blazor (server) applications.

For more information on tasks, targets, properties, implicit blobs, globs, publishing, methods, and more, see the [README file](#) in the [WebSdk](#) repository.

# ASP.NET Core code generator tool

## (`aspnet-codegenerator`)

Article • 08/06/2024

### Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

The `dotnet aspnet-codegenerator` command runs the ASP.NET Core scaffolding engine. Running the `dotnet aspnet-codegenerator` command is required to scaffold from the command line or when using Visual Studio Code. The command isn't required to use scaffolding with Visual Studio, which includes the scaffolding engine by default.

## Install and update the code generator tool

Install the [.NET SDK](#).

`dotnet aspnet-codegenerator` is a [global tool](#) that must be installed. The following command installs the latest stable version of the ASP.NET Core code generator tool:

.NET CLI

```
dotnet tool install -g dotnet-aspnet-codegenerator
```

### Note

By default the architecture of the .NET binaries to install represents the currently running OS architecture. To specify a different OS architecture, see [dotnet tool install, --arch option](#). For more information, see GitHub issue [dotnet/AspNetCore.Docs #29262](#).

If the tool is already installed, the following command updates the tool to the latest stable version available from the installed .NET Core SDKs:



.NET CLI

```
dotnet tool update -g dotnet-aspnet-codegenerator
```

## Uninstall the code generator tool

It may be necessary to uninstall the ASP.NET Core code generator tool to resolve problems. For example, if you installed a preview version of the tool, uninstall it before installing the released version.

The following commands uninstall the ASP.NET Core code generator tool and install the latest stable version:

.NET CLI

```
dotnet tool uninstall -g dotnet-aspnet-codegenerator  
dotnet tool install -g dotnet-aspnet-codegenerator
```

## Synopsis

```
dotnet aspnet-codegenerator [arguments] [-b|--build-base-path] [-c|--  
configuration] [-n|--nuget-package-dir] [--no-build] [-p|--project] [-tfm|--  
target-framework]  
dotnet aspnet-codegenerator [-h|--help]
```

## Description

The `dotnet aspnet-codegenerator` global command runs the ASP.NET Core code generator and scaffolding engine.

## Arguments

`generator`

The code generator to run. The available generators are shown in the following table.

[Expand table](#)

Generator	Operation
<code>area</code>	<a href="#">Scaffolds an area.</a>
<code>blazor</code>	<a href="#">Scaffolds Blazor create, read, update, delete, and list pages.</a>
<code>blazor-identity</code>	Generates Blazor Identity files.
<code>controller</code>	<a href="#">Scaffolds a controller.</a>
<code>identity</code>	<a href="#">Scaffolds Identity.</a>
<code>minimalapi</code>	Generates an endpoints file (with CRUD API endpoints) given a model and optional database context.
<code>razorpage</code>	<a href="#">Scaffolds Razor Pages.</a>
<code>view</code>	<a href="#">Scaffolds a view.</a>

## Options

`-b|--build-base-path`

The build base path.

`-c|--configuration {Debug|Release}`

Defines the build configuration. The default value is `Debug`.

`-h|--help`

Prints out a short help for the command.

`-n|--nuget-package-dir`

Specifies the NuGet package directory.

`--no-build`

Doesn't build the project before running. Passing `--no-build` also implicitly sets the `--no-restore` flag.

`-p|--project <PATH>`

Specifies the path of the project file to run (folder name or full path). If not specified, the tool defaults to the current directory.

`-tfm|--target-framework`

The target [framework](#) to use.

## Generator options

The following sections detail the options available for the supported generators:

- [Area \(area\)](#)
- [Controller \(controller\)](#)
- [Blazor \(blazor\)](#)
- [Blazor Identity \(blazor-identity\)](#)
- [Identity \(identity\)](#)
- [Minimal API \(minimalapi\)](#)
- [Razor page \(razorpage\)](#)
- [View \(view\)](#)

## Area options

Usage: `dotnet aspnet-codegenerator area {AREA NAME}`

The `{AREA NAME}` placeholder is the name of the area to generate.

The preceding command generates the following folders:

- `Areas`
  - `{AREA NAME}`
    - `Controllers`
    - `Data`
    - `Models`
    - `Views`

Use the `-h|--help` option for help:

```
.NET CLI
```

```
dotnet aspnet-codegenerator area -h
```

## Blazor options

Razor components can be individually scaffolded for Blazor apps by specifying the name of the template to use. The supported templates are:

- Empty
- Create
- Edit
- Delete
- Details
- List
- **CRUD**: *CRUD* is an acronym for Create, Read, Update, and Delete. The **CRUD** template produces **Create**, **Edit**, **Delete**, **Details**, and **Index** (**List**) components for the app.

The options for the **blazor** generator are shown in the following table.

[Expand table](#)

Option	Description
<code>-dbProvider --databaseProvider</code>	Database provider to use. Options include <b>sqlserver</b> (default), <b>sqlite</b> , <b>cosmos</b> , or <b>postgres</b> .
<code>-dc --dataContext</code>	Database context class to use.
<code>-m --model</code>	Model class to use.
<code>-ns --namespaceName</code>	Specify the name of the namespace to use for the generated Endpoints file.
<code>--relativeFolderPath -outDir</code>	Relative output folder path. If not specified, files are generated in the project folder.

The following example:

- Uses the **Edit** template to generate an **Edit** component (**Edit.razor**) in the **Components/Pages/MoviePages** folder of the app. If the **MoviePages** folder doesn't exist, the tool creates the folder automatically.
- Uses the SQLite database provider.
- Uses **BlazorWebAppMovies.Data.BlazorWebAppMoviesContext** for the database context.
- Uses the **Movie** model.

.NET CLI

```
dotnet aspnet-codegenerator blazor Edit -dbProvider sqlite -dc
BlazorWebAppMovies.Data.BlazorWebAppMoviesContext -m Movie -outDir
Components/Pages
```

Use the `-h|--help` option for help:

```
.NET CLI

dotnet aspnet-codegenerator blazor -h
```

For an example that uses the `blazor` generator, see [Build a Blazor movie database app \(Overview\)](#).

For more information, see [ASP.NET Core Blazor QuickGrid component](#).

## Blazor Identity options

Scaffold Identity Razor components into a Blazor app with the `blazor-identity` generator.

The options for the `blazor-identity` template are shown in the following table.

 Expand table

Option	Description
<code>-dbProvider --databaseProvider</code>	Database provider to use. Options include <code>sqlserver</code> (default) and <code>sqlite</code> .
<code>-dc --dataContext</code>	Database context class to use.
<code>-f --force</code>	Use this option to overwrite existing files.
<code>-fi --files</code>	List of semicolon separated files to scaffold. Use the <code>-lf --listFiles</code> option to see the available options.
<code>-lf --listFiles</code>	Lists the files that can be scaffolded by using the <code>-fi --files</code> option.
<code>-rn --rootNamespace</code>	Root namespace to use for generating Identity code.
<code>-u --userClass</code>	Name of the user class to generate.

Use the `-h|--help` option for help:

```
.NET CLI

dotnet aspnet-codegenerator blazor-identity -h
```

# Controller options

General options are shown in the following table.

 Expand table

Option	Description
<code>-b --bootstrapVersion</code>	Specifies the bootstrap version and creates a <code>wwwroot</code> folder for the Bootstrap assets if the folder isn't present.
<code>-dbProvider --databaseProvider</code>	Database provider to use. Options include <code>sqlserver</code> (default), <code>sqlite</code> , <code>cosmos</code> , <code>postgres</code> .
<code>-dc --dataContext</code>	The database context class to use or the name of the class to generate.
<code>-f --force</code>	Overwrite existing files.
<code>-l --layout</code>	Custom layout page to use.
<code>-m --model</code>	Model class to use.
<code>-outDir --relativeFolderPath</code>	Relative output folder path. If not specified, files are generated in the project folder.
<code>-scripts --referenceScriptLibraries</code>	Reference script libraries in the generated views. Adds <code>_ValidationScriptsPartial</code> to <code>Edit</code> and <code>Create</code> pages.
<code>-sqlite --useSqlite</code>	Flag to specify if the database context should use SQLite instead of SQL Server.
<code>-udl --useDefaultLayout</code>	Use the default layout for the views.

The options unique to `controller` are shown in the following table.

 Expand table

Option	Description
<code>-actions --readWriteActions</code>	Generate controller with read/write actions without a model.
<code>-api --restWithNoViews</code>	Generate a controller with REST style API. <code>noViews</code> is assumed and any view related options are ignored.
<code>-async --useAsyncActions</code>	Generate asynchronous controller actions.
<code>-name --controllerName</code>	Name of the controller.

Option	Description
<code>-namespace --controllerNamespace</code>	Specify the name of the namespace to use for the generated controller.
<code>-nv --noViews</code>	Generate <b>no</b> views.

Use the `-h|--help` option for help:

```
.NET CLI

dotnet aspnet-codegenerator controller -h
```

For an example that uses the `controller` generator, see [Part 4, add a model to an ASP.NET Core MVC app](#).

## Identity options

For more information, see [Scaffold Identity in ASP.NET Core projects](#).

## Minimal API options

Scaffold a Minimal API backend with the `minimalapi` template.

The options for `minimalapi` are shown in the following table.

 Expand table

Option	Description
<code>-dbProvider --databaseProvider</code>	Database provider to use. Options include <code>sqlserver</code> (default), <code>sqlite</code> , <code>cosmos</code> , or <code>postgres</code> .
<code>-dc --dataContext</code>	Database context class to use.
<code>-e --endpoints</code>	Endpoints class to use (not the file name).
<code>-m --model</code>	Model class to use.
<code>-namespace --endpointsNamespace</code>	Specify the name of the namespace to use for the generated endpoints file.
<code>-o --open</code>	Use this option to enable OpenAPI.
<code>-outDir --relativeFolderPath</code>	Relative output folder path. If not specified, files are generated in the project folder.

Option	Description
<code>-sqlite --useSqlite</code>	Flag to specify if the database context should use SQLite instead of SQL Server.

The following example:

- Generates an endpoints class named `SpeakersEndpoints` with API endpoints that map to database operations using the `ApplicationDbContext` database context class and the `BackEnd.Models.Speaker` model.
- Adds `app.MapSpeakerEndpoints();` to the `Program` file (`Program.cs`) to register the endpoints class.

.NET CLI

```
dotnet aspnet-codegenerator minimalapi -dc ApplicationDbContext -e
SpeakerEndpoints -m BackEnd.Models.Speaker -o
```

Use the `-h|--help` option for help:

.NET CLI

```
dotnet aspnet-codegenerator minimalapi -h
```

## Razor page options

Razor Pages can be individually scaffolded by specifying the name of the new page and the template to use. The supported templates are:

- `Empty`
- `Create`
- `Edit`
- `Delete`
- `Details`
- `List`

Typically, the template and generated file name isn't specified, which creates the following templates:

- `Create`
- `Edit`
- `Delete`



- [Details](#)
- [List](#)

General options are shown in the following table.

[Expand table](#)

Option	Description
<code>-b --bootstrapVersion</code>	Specifies the bootstrap version and creates a <code>wwwroot</code> folder for the Bootstrap assets if the folder isn't present.
<code>-dbProvider --databaseProvider</code>	Database provider to use. Options include <code>sqlserver</code> (default), <code>sqlite</code> , <code>cosmos</code> , <code>postgres</code> .
<code>-dc --dataContext</code>	The database context class to use or the name of the class to generate.
<code>-f --force</code>	Overwrite existing files.
<code>-l --layout</code>	Custom layout page to use.
<code>-m --model</code>	Model class to use.
<code>-outDir --relativeFolderPath</code>	Relative output folder path. If not specified, files are generated in the project folder.
<code>-scripts --referenceScriptLibraries</code>	Reference script libraries in the generated views. Adds <code>_ValidationScriptsPartial</code> to <code>Edit</code> and <code>Create</code> pages.
<code>-sqlite --useSqlite</code>	Flag to specify if the database context should use SQLite instead of SQL Server.
<code>-udl --useDefaultLayout</code>	Use the default layout for the views.

The options unique to `razorpage` are shown in the following table.

[Expand table](#)

Option	Description
<code>-namespace --namespaceName</code>	The name of the namespace to use for the generated <code>PageModel</code> class.
<code>-npm --noPageModel</code>	Don't generate a <code>PageModel</code> class for the <code>Empty</code> template.
<code>-partial --partialView</code>	Generate a partial view. Layout options <code>-l</code> and <code>-udl</code> are ignored if this is specified.

The following example uses the `Edit` template to generate `CustomEditPage.cshtml` and `CustomEditPage.cshtml.cs` in the `Pages/Movies` folder:

```
.NET CLI

dotnet aspnet-codegenerator razorpage CustomEditPage Edit -dc
RazorPagesMovieContext -m Movie -outDir Pages/Movies
```

Use the `-h|--help` option for help:

```
.NET CLI

dotnet aspnet-codegenerator razorpage -h
```

For an example that uses the `razorpage` generator, see [Part 2, add a model](#).

## View options

Views can be individually scaffolded by specifying the name of the view and the template. The supported templates are:

- `Empty`
- `Create`
- `Edit`
- `Delete`
- `Details`
- `List`

General options are shown in the following table.

 Expand table

Option	Description
<code>-b --bootstrapVersion</code>	Specifies the bootstrap version and creates a <code>wwwroot</code> folder for the Bootstrap assets if the folder isn't present.
<code>-dbProvider --databaseProvider</code>	Database provider to use. Options include <code>sqlserver</code> (default), <code>sqlite</code> , <code>cosmos</code> , <code>postgres</code> .
<code>-dc --dataContext</code>	The database context class to use or the name of the class to generate.
<code>-f --force</code>	Overwrite existing files.

Option	Description
<code>-l --layout</code>	Custom layout page to use.
<code>-m --model</code>	Model class to use.
<code>-outDir --relativeFolderPath</code>	Relative output folder path. If not specified, files are generated in the project folder.
<code>-scripts --referenceScriptLibraries</code>	Reference script libraries in the generated views. Adds <code>_ValidationScriptsPartial</code> to <code>Edit</code> and <code>Create</code> pages.
<code>-sqlite --useSqlite</code>	Flag to specify if the database context should use SQLite instead of SQL Server.
<code>-udl --useDefaultLayout</code>	Use the default layout for the views.

The options unique to `view` are shown in the following table.

[Expand table](#)

Option	Description
<code>-namespace --controllerNamespace</code>	Specify the name of the namespace to use for the generated controller.
<code>-partial --partialView</code>	Generate a partial view. Other layout options ( <code>-l</code> and <code>-udl</code> ) are ignored if this is specified.

The following example uses the `Edit` template to generate `CustomEditView.cshtml` in the `Views/Movies` folder:

.NET CLI

```
dotnet aspnet-codegenerator view CustomEditView Edit -dc MovieContext -m
Movie -outDir Views/Movies
```

Use the `-h|--help` option for help:

.NET CLI

```
dotnet aspnet-codegenerator view -h
```

# Choose between controller-based APIs and minimal APIs

Article • 04/11/2023

## Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

ASP.NET Core supports two approaches to creating APIs: a controller-based approach and minimal APIs. *Controllers* in an API project are classes that derive from [ControllerBase](#). *Minimal APIs* define endpoints with logical handlers in lambdas or methods. This article points out differences between the two approaches.

The design of minimal APIs hides the host class by default and focuses on configuration and extensibility via extension methods that take functions as lambda expressions. Controllers are classes that can take dependencies via constructor injection or property injection, and generally follow object-oriented patterns. Minimal APIs support dependency injection through other approaches such as accessing the service provider.

Here's sample code for an API based on controllers:

C#

```
namespace APIWithControllers;

public class Program
{
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

        builder.Services.AddControllers();
        var app = builder.Build();

        app.UseHttpsRedirection();

        app.MapControllers();

        app.Run();
    }
}
```

```
}  
}
```

C#

```
using Microsoft.AspNetCore.Mvc;  
  
namespace APIWithControllers.Controllers;  
[ApiController]  
[Route("[controller]")]  
public class WeatherForecastController : ControllerBase  
{  
    private static readonly string[] Summaries = new[]  
    {  
        "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy",  
        "Hot", "Sweltering", "Scorching"  
    };  
  
    private readonly ILogger<WeatherForecastController> _logger;  
  
    public WeatherForecastController(ILogger<WeatherForecastController>  
logger)  
    {  
        _logger = logger;  
    }  
  
    [HttpGet(Name = "GetWeatherForecast")]  
    public IEnumerable<WeatherForecast> Get()  
    {  
        return Enumerable.Range(1, 5).Select(index => new WeatherForecast  
        {  
            Date = DateOnly.FromDateTime(DateTime.Now.AddDays(index)),  
            TemperatureC = Random.Shared.Next(-20, 55),  
            Summary = Summaries[Random.Shared.Next(Summaries.Length)]  
        })  
        .ToArray();  
    }  
}
```

The following code provides the same functionality in a minimal API project. Notice that the minimal API approach involves including the related code in lambda expressions.

C#

```
namespace MinimalAPI;  
  
public class Program  
{  
    public static void Main(string[] args)  
    {  
        var builder = WebApplication.CreateBuilder(args);  
    }  
}
```

```

var app = builder.Build();

app.UseHttpsRedirection();

var summaries = new[]
{
    "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm",
    "Balmy", "Hot", "Sweltering", "Scorching"
};

app.MapGet("/weatherforecast", (HttpContext httpContext) =>
{
    var forecast = Enumerable.Range(1, 5).Select(index =>
        new WeatherForecast
        {
            Date =
                DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
            TemperatureC = Random.Shared.Next(-20, 55),
            Summary =
                summaries[Random.Shared.Next(summaries.Length)]
        })
        .ToArray();
    return forecast;
});

app.Run();
}
}

```

Both API projects refer to the following class:

C#

```

namespace APIWithControllers;

public class WeatherForecast
{
    public DateOnly Date { get; set; }

    public int TemperatureC { get; set; }

    public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);

    public string? Summary { get; set; }
}

```

Minimal APIs have many of the same capabilities as controller-based APIs. They support the configuration and customization needed to scale to multiple APIs, handle complex routes, apply authorization rules, and control the content of API responses. There are a

few capabilities available with controller-based APIs that are not yet supported or implemented by minimal APIs. These include:

- No built-in support for model binding ([IModelBinderProvider](#), [IModelBinder](#)). Support can be added with a custom binding shim.
- No built-in support for validation ([IModelValidator](#)).
- No support for [application parts](#) or the [application model](#). There's no way to apply or build your own conventions.
- No built-in view rendering support. We recommend using [Razor Pages](#) for rendering views.
- No support for [JsonPatch](#) [↗](#)
- No support for [OData](#) [↗](#)

## See also

- [Create web APIs with ASP.NET Core.](#)
- [Tutorial: Create a web API with ASP.NET Core](#)
- [Minimal APIs overview](#)
- [Tutorial: Create a minimal API with ASP.NET Core](#)

# Create web APIs with ASP.NET Core

Article • 06/01/2024

ASP.NET Core supports creating web APIs using controllers or using minimal APIs. *Controllers* in a web API are classes that derive from [ControllerBase](#). Controllers are activated and disposed on a per request basis.

This article shows how to use controllers for handling web API requests. For information on creating web APIs without controllers, see [Tutorial: Create a minimal API with ASP.NET Core](#).

## ControllerBase class

A controller-based web API consists of one or more controller classes that derive from [ControllerBase](#). The web API project template provides a starter controller:

C#

```
[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
```

Web API controllers should typically derive from [ControllerBase](#) rather from [Controller](#). [Controller](#) derives from [ControllerBase](#) and adds support for views, so it's for handling web pages, not web API requests. If the same controller must support views and web APIs, derive from [Controller](#).

The [ControllerBase](#) class provides many properties and methods that are useful for handling HTTP requests. For example, [CreatedAtAction](#) returns a 201 status code:

C#

```
[HttpPost]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public ActionResult<Pet> Create(Pet pet)
{
    pet.Id = _petsInMemoryStore.Any() ?
        _petsInMemoryStore.Max(p => p.Id) + 1 : 1;
    _petsInMemoryStore.Add(pet);

    return CreatedAtAction(nameof(GetById), new { id = pet.Id }, pet);
}
```