

ASP.NET Core's Tag Helpers are short codes you can use in your CSHTML (Razor) views or pages.

When ASP.NET Core serves a page with Tag Helpers it finds all the Tag Helpers and converts them into regular HTML markup.

This means you can write something like this...

```
<input asp-for="Email" />
```

And ASP.NET Core will turn it into this (before returning it for the browser to render).

```
<input type="text" name="Email" id="Email" value="" />
```

There are a number of built-in Tag Helpers (as well as many created by third parties, and you can create your own if you want).

Here are 5 most useful "built-in" Tag Helpers.

#1 form

When you declare a form in a CSHTML view/page, you're already using a Tag Helper (even though you might not realise it).

```
<form method="post">
  <label>Title</label>
  <input type="text" name="title" />
  <input type="submit" class="btn" />
</form>
```

Even with a simple form like this, ASP.NET Core recognises the `form` tag as a Tag Helper and enhances it before sending it up to be rendered in the browser.

Look at the source code in your browser and you'll see something like this...

```
<form method="post">
  <label>Title</label>
  <input type="text" name="title" />
  <input type="submit" class="btn" />
  <input name="__RequestVerificationToken" type="hidden"
  value="cFdJ8JKuw5POut90h6SqY5ytmHOTzno4IMDuDDrCCuG90GYvfEek2WY0QUC6T
  cprJRWAGitZYEvRQiwc80jB1yFo4ES5AvwAW4S7LfZPGPtOy8s8HL09Gwtoj3n9m1Jgi
  0pSpV_GVgMr5Rdw4z64wlc8A8k" /></form>
```

ASP.NET Core has automatically added the `__RequestVerificationToken` hidden input.

This is important for one primary reason; it helps protect your application from XSRF and CSRF.

The perils of cross-site request forgery (XSRF or CSRF)

Imagine this scenario.

We set up an authentication mechanism for our site and allow users to log in so they can see their own data as well as being able to add/edit/delete etc.

- User A comes along to the application and logs in (e.g. `https://somewhere.io/login`).
- Our app authenticates the user and sends a response back which includes an authentication cookie.

So far so good, our user is logged in to our app and the cookie is stored in their browser, meaning they can access pages in the app without re-entering their login details every time.

- But now they're somehow tricked into going to a malicious site, hosted somewhere else (nothing to do with our app).
- This malicious site has a form which posts back to our application.

```
<h1>Your PC has been infected!</h1>
<form method="post"
  action="https://somewhere.io/account/delete">
  <input type="hidden" name="accountId" value="1">
  <input type="submit" value="Click here to fix">
</form>
```

Now we have a significant problem.

This form posts back to our application, even though it's hosted somewhere else entirely.

When a user submits this form the browser automatically sends any cookies with the request (including the authentication cookie our user has stored in their browser).

Now the code in our app which lives at `/account/delete` will run as if it was deliberately triggered by the logged in user.

Worse still, whilst this form required the user to click a button, the page could just have easily run a script which automatically submits the form (or made the request as an AJAX request using javascript) meaning the innocent user wouldn't need to click on anything for the malicious code to wreak havoc.

The Form Tag Helper to the rescue

That strange looking input (`__RequestVerificationToken`) we saw in the rendered HTML for our form is our first step towards preventing Cross Site Request Forgery for our site.

When we request a page via the browser, that request is sent to our application and handled by ASP.NET Core.

Using this tag helper means that ASP.NET Core automatically generates a secure token on the server when handling that request, then returns this token in the markup that it generates.

This markup is returned and rendered in the browser. The user won't see it because the field is of type `hidden`.

When we subsequently submit this form the hidden token is posted back to our application.

On the server side, ASP.NET Core can then validate this token to ensure the request came from a form which was generated using our application, and not some random form that lives elsewhere and just posts directly to our application.

A malicious form hosted elsewhere has no way of generating a token that ASP.NET Core will accept as proof that the request is legitimate.

#2 asp-for

So the `form` tag helper made it easy for us to be more secure, but what else can we do?

If you use a Model with your Razor page, you can take advantage of 'asp-for' to render the HTML elements for you (and automatically wire them up to the Model).

Here's an example.

This `Create` action takes an instance of a `NewBoard` model and then uses it to create a new "board" in the system (this is a Kanban board application).

```
[HttpPost]
public IActionResult Create(NewBoard viewModel)
{
    boardService.AddBoard(viewModel);
    return RedirectToAction(nameof(Index));
}
```

We could create an input for each property on that model manually...

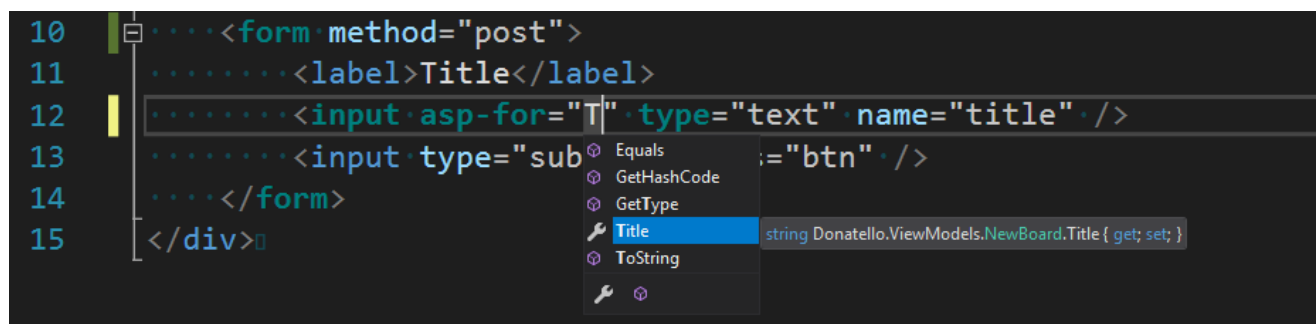
For example, if `NewBoard` had a `Title` property, we could write this HTML.

```
<input type="text" name="title">
```

When this form is posted, ASP.NET Core will spot the incoming value for 'name' and populate the `NewBoard` model's `Title` property accordingly.

But we can simplify this using the `asp-for` attribute (and remove the `name="title"` attribute).

When you add an `asp-for=` attribute you'll find Visual Studio gives you a handy list of the properties which are available in the specified model (`NewBoard` in this case).



```
10 <form method="post">
11 <label>Title</label>
12 <input asp-for="Title" type="text" name="title" />
13 <input type="submit" value="Submit" />
14 </form>
15 </div>
```

If we chose `Title` we'd end up with a form something like this.

```
<form method="post">
  <label>Title</label>
  <input asp-for="Title" type="text" />
  <input type="submit" class="btn" />
</form>
```

Gone is `name="title"` and in its place is `asp-for="Title"`.

This would work exactly as if we'd manually hardcoded the `name` attribute to `title`.

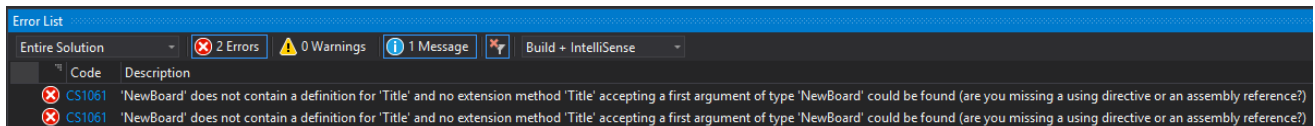
The real benefit comes when renaming properties.

If we changed the name of the `Title` property in `NewBoard` to something else...

```
public class NewBoard
{
    public string BrokenTitle { get; set; }
}
```

Then tried to build the project (**Ctrl+Shift+B**), we'd get a number of warnings.

One of these points out `Create.cshtml` references `Title` which no longer exists in the ViewModel.



Without the tag helper we could easily have broken the link between our form and the ViewModel (because the `name=` attribute simply used a string value to match them up).

With the tag helper in place, we can't proceed without resolving these compilation errors (which means fixing the form).

#3 asp-append-version

This one is really extremely handy and is used with your image tags.

The standard way of writing image tags is something like this.

```

```

And for the most part this works exactly as you'd expect.

However, you can run into issues with browser caching.

Modern browsers cache a lot (to try and improve performance) and they will likely cache any images on your site the first time they are displayed to a user.

Subsequently, even if you change the image (swap it for another one) the browser will continue to serve the image from its cache, rather than "re-download" the file.

This means your users will almost certainly see an outdated version of any images you include on your site.

A tried and trusted (but also fiddly and manual) way to avoid this is to put some bogus querystring at the end of the url for your images.

```

```

This way, every time you change the image you need to update the querystring (e.g. `v=2`) and the browser will see it as a new file (then download and cache the new version).

With ASP.NET Core you can remove the manual step of updating the querystring by simply using a Tag Helper.

```

```

With that in place, ASP.NET Core will generate a unique "cache-busting" url for your images and automatically generate a new URL when the image is changed.

```

```

This is achieved by generating the Sha512 hash value for the file. This hash is then used as the value for `v` in the querystring. When you change the file ASP.NET Core generates the hash again, which will be different because the file is different, then uses that new hash as the value for `v`.

Suffice to say it's a real time-saver and means you need never worry about cached images again!

#4 environment

Sometimes you might want to show different content on your pages based on which environment you're running in.

ASP.NET Core introduced the concept of environments (the standard ones being Development, Staging and Production).

The idea is you can alter things like application settings based on whether you're running locally or in production.

The most obvious use case is to have different database connection strings for your local machine vs the production server (usually a good idea!)

But environments can be useful for more than just settings.

Let's say you want to serve unminified javascript files when you're working on your application, but minified versions when you're running in production.

You can do this using the `environment` tag helper.

```
<environment include="Development">
  <script src="myscripts.js"></script>
</environment>

<environment include="Staging, Production">
  <script src="myscripts.min.js"></script>
</environment>
```

In Development you'd see this in your HTML source...

```
<script src="myscripts.js"></script>
```

And in Staging or Production you'd get the minified version...

```
<script src="myscripts.min.js"></script>
```

Another way to achieve the same thing is to use the `exclude` syntax...

```
<environment include="Development">
  <script src="myscripts.js"></script>
</environment>

<environment exclude="Development">
  <script src="myscripts.min.js"></script>
</environment>
```

Now you'll get the minified version in all environments except Development.

This has the advantage that it will also handle non-standard environments (you can have as many environments as you like in ASP.NET Core) without having to explicitly list them all (as we would have done in the first example).

#5 partial

Partial Views are a handy feature in ASP.NET Core MVC which let you include Razor markup from one file in another.

For example, you might want to break up a big CSHTML page into lots of smaller components, or take something common and render it in various places throughout your application.

With a Partial View you can declare some CSHTML in one file, then include it wherever you want it to be displayed in your other Razor Views and Pages.

In Core (and previous versions of MVC) you can use HTML Helpers to render partials...

```
@HTML.Partial()
```

But now we also have Tag Helpers.

```
<partial name="Shared/ContactUs.cshtml" />
```

This will render the contents of `ContactUs.cshtml` wherever we include this `partial` tag helper.

For example...

```
<h1>
  welcome
</h1>
<partial name="Shared/ContactUs.cshtml" />
<p>
  Lots more content here...
</p>
<partial name="Shared/ContactUs.cshtml" />
```

Here we'd get two instances of our ContactUs form.

You can also pass a model to the Partial.

Take this partial...

ProductSummary.cshtml

```
@model MyApp.Models.Product
<h2>@Model.Name</h2>
<p>
  @Model.Description
</p>
```

This is looking to render a product summary and requires a model of type `ProductDetails`.

Wherever we want to render this partial, we'd need to pass in an instance of `ProductDetails`.

ProductList.cshtml

```
@model MyApp.Models.ProductList

@foreach (var product in Model.Products)
{
  <partial name="Products/Summary.cshtml"
    for="@product" />
}
}
```

This loops through a list of `Products` on a `ProductList` model and, for each product, renders the partial.

Here's an example of a ViewModel which would work in this scenario.

```
public class ProductList
{
    public List<Product> Products { get; set; } = new List<Product>();

    public class Product
    {
        public string Name { get; set; }
        public string Description { get; set; }
    }
}
```

In conclusion

Tag Helpers are very useful and can really save you a lot of time (and typing) when you're working on your ASP.NET Core pages.

But more than just saving time, they also enable some scenarios which are quite difficult to achieve otherwise (cache-busting images and environment-dependent content, to name but two).

For more Tag Helpers check out the [official documentation here](#).