

Welcome,

This brief exploration of Tag Helpers is taken from my book - [Practical ASP.NET Core MVC](#) where we build a kanban board application, using things like Tag Helpers to quickly spin up new features.

This Tag Helpers chapter comes just after we've created a form to enable users to create new Kanban boards.

You'll see how to add Tag Helper support to a blank ASP.NET Core project (note, if you create the project using the MVC or Razor Pages templates, this step is done for you).

And we look at handling cross-site request forgery (and why it matters) and a couple more examples of how Tag Helpers can simplify your markup.

If you have any questions about Tag Helpers (or anything else) drop me an email at jon@jonhilton.net

-- Jon

Improve your forms with Tag Helpers

In the last lesson we created a form to enable users to create new Kanban boards.

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

This works and in fact this form could be used with any web framework, it's just standard HTML.

It is a little brittle though.

What happens if you change the name of the property on your ViewModel from "Title" to something else? The form will still load and submit, but your ViewModel's `Title` property will be empty and you'll lose whatever value the user entered.

There is another issue here too.

Your app is currently susceptible to Cross-site request forgery.

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 boards/create new boards etc.

- User A comes along to our application and logs in (e.g. `https://donate11o.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 means 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://donatello.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 the authentication cookie with the request.

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 but 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 didn't need to click on anything for the malicious code to wreak havoc.

Improving our form (with bonus CSRF protection)

Thankfully ASP.NET Core has built in mechanisms to tackle this problem and these are quite simple to implement, especially if you use Tag Helpers.

Tag Helpers are special tags you can include in your Razor Views which render HTML markup.

They're easier to understand with an example so let's revisit our form.

Open up `Views\Home\Create.cshtml` and add this to the top of the view...

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

<!-- rest of view here... -->
```

This ensures that all of the tag helpers which have been created by Microsoft (and live in `Microsoft.AspNetCore.Mvc.TagHelpers`) are made available to our view.

For now we'll just do this here for this specific View but in a bit we'll see how to specify this in one place for every View.

Form tag helper

Now you might notice your `form` tag change color at this point...

```
<div class="container">
  <h2>Add a new Board</h2>
  <form method="post">
    <label>Title</label>
    <input type="text" name="title" />
    <input type="submit" class="btn" />
  </form>
</div>
```

This highlights that tag helpers are in effect and that extra HTML markup will be generated when this page is rendered by ASP.NET MVC Core (and served to the browser).

If you run your app now and navigate to `/home/create` you'll see something like this markup (view source in the browser).

```
<form method="post">
  <label>Title</label>
  <input type="text" name="title" />
  <input type="submit" class="btn" />
  <input name="__RequestVerificationToken" type="hidden"
value="CfDJ8JKuW5Pout90h6SqY5ytmHOTzno4IMDuDDrCCuG90GYvfEek2WY0QUC6T
cprJRWAGItZYEVrQiwc80jB1yFo4ES5AvwAW4S7LfZPGPtOy8s8HL09Gwtoj3n9m1Jgi
OpSpV_GVgMr5Rdw4z64wlc8A8k" /></form>
```

That strange looking input (`__RequestVerificationToken`) is our first step towards preventing Cross Site Request Forgery for our site.

When we request `/home/create` in 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 then 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.

Validating anti forgery tokens

By default ASP.NET Core won't check for these tokens, we need to tell it to do so.

All we need to do is add an attribute to our controllers.

Head over to **HomeController.cs**.

Add the `ValidateAntiForgeryToken` attribute to the controller...

```
[ValidateAntiForgeryToken]
public class HomeController : Controller
{
    // rest of controller here...
```

ASP.NET will now validate the token for every request to a `POST` action on this controller. If the request sends a valid token it will be processed, if not it will be rejected.

Keep your GET actions read-only

It's considered good practice not to write/update data in your app if the request is a `GET` request.

If you allow GET requests to change data then your app is very easily targeted even without submitting a form/POST request.

Even a humble image tag on a malicious site can be used to make a GET request to your site.

For that reason, the `ValidateAntiForgeryToken` only checks for valid tokens on `POST` requests in your controller and assumes your `GET` actions are safe to run without requiring tokens (it would be quite awkward to generate tokens for every `GET` request to your site).

Auto-validate everything by default

The only real downside to using the `ValidateAntiForgeryToken` attribute everywhere is that you have to remember to add it.

If you don't, ASP.NET will just let those potentially malicious requests straight through to your code.

Happily there's an easy alternative, you can configure this in one place instead of having to include it on every controller.

Head over to **Startup.cs** and the `ConfigureServices` method and modify the existing code...

We already have `services.AddMvc`, now we need to tweak it to configure its options...

```
public void ConfigureServices(IServiceCollection services)
{
    // rest of code omitted for brevity

    services.AddMvc(options =>
    {
        options.Filters.Add(new AutoValidateAntiForgeryTokenAttribute());
    });
}
```

This is a new concept for us, the **Global Filter**.

ASP.NET Core's global filters are applied to every single action in your MVC app and so are really handy for cases like this, where you don't want to have to remember to do something on every controller.

Now we can continue building our application safe in the knowledge that all POST/PUT/DELETE etc. requests will be intercepted and blocked if they don't include a valid token.

Further improving our form with asp-for

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

Currently any changes we make to property names on our `NewBoard` ViewModel will break our form because the property names will diverge from the values we've specified in `name` attributes. Tag helpers can assist with this problem as well.

Remember our form posts back to a controller action which takes in a `NewBoard` ViewModel parameter.

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

Our view is currently ignorant of this fact. If we hadn't hardcoded the correct `name` attribute values (which we did simply because we happened to know what the corresponding ViewModel properties were) then our `Create` View (and form) would have no clue what data to send in the `POST` request.

We can make our View aware of the ViewModel we're using by adding a `@model` declaration.

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@model Donatello.ViewModels.NewBoard

<!-- rest of view here -->
```

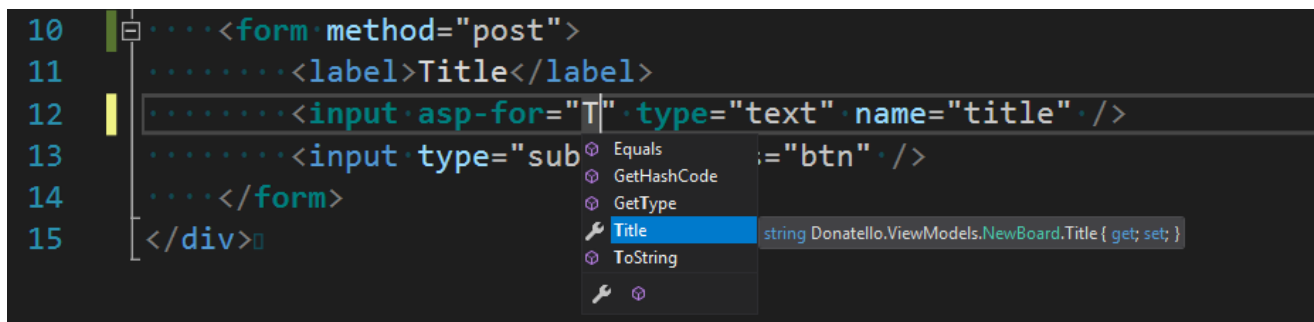
Now we can tweak our form, specifically the input for `title`.

Remember this line...

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

We want to add an `asp-for=""` attribute and remove the `name="title"` attribute.

When you add the `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>
```

Choose **Title** and you should end up with a form 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"`.

Test this in the browser and you'll see the rendered markup still has a `name` attribute only this time it's been generated by ASP.NET because of that tag helper.

The real benefit comes when you decide to rename that property.

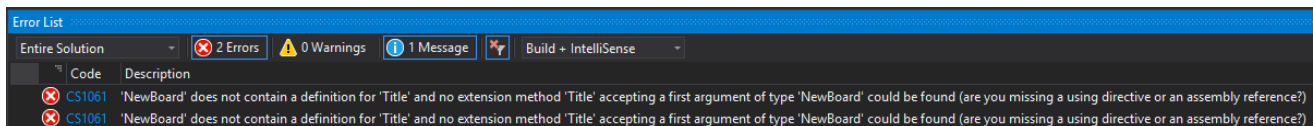
Try it out for yourself by clicking on `Title` (next to `asp-for`) in **Create.cshtml**, press **F12** and you'll navigate straight to the property in **NewBoard.cs**.

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

Now try changing the name of that property from `Title` to `BrokenTitle`.

Build your project (**Ctrl+Shift+B**) and you'll get a couple of warnings.

One of those is because `Create.cshtml` references `Title` which no longer exists in the ViewModel.



Without the tag helper you might well have missed that the view referenced this property and deployed a broken application to your users.

(Remember to change it back to the correct name before you move on!)

Being explicit about our controllers and actions

Finally, you can also use Tag Helpers to be specific about where you want your forms to be posted.

By default our form will post back to the same URL as the page it's hosted on.

```
http://<your-app-here>/home/create
```

This is because we didn't specify an **action** for the form.

We could make this more explicit by specifying a value for the `action` attribute...

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

This behaves the same as not including the `action` attribute at all (in our example).

Alternatively you can use tag helpers here instead...

```
<form method="post" asp-controller="Home" asp-action="Create">
  <label>Title</label>
  <input asp-for="Title" type="text" name="title" />
  <input type="submit" class="btn" />
</form>
```

This is the same again, but now carries the benefit that we can change the routing for our application, potentially making that action available at a different URL, but the form would still render a correct value for the form's `action` attribute.

Try it in the browser and check the rendered markup for yourself.

Linking from the Board List page to Add New

Remember our Boards List page?

It would be useful to have an Add button on there which takes users straight over to the **Add Board** page.

Head over to **Views/Home/Index.cshtml** and add an a tag directly under the existing `<h2>Boards</h2>` tag...

```
<div class="row">
  <h2>Boards</h2>
  <a asp-action="Create" asp-controller="Home"
    class="waves-effect waves-light btn">
    New Board
  </a>
</div>
```

With our new-found knowledge of Tag Helpers we can guess that this is going render an `a` tag with the correct url for our **Add Board** page.

Check it out in the browser though and you'll find it doesn't work...

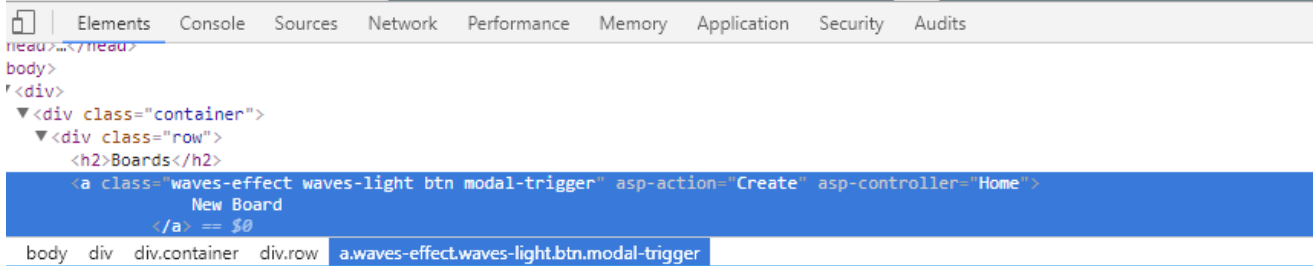
Inspect the **New Board** button in whichever browser you're using and you'll see this...

Boards

NEW BOARD

Jon's Board

Todo Board



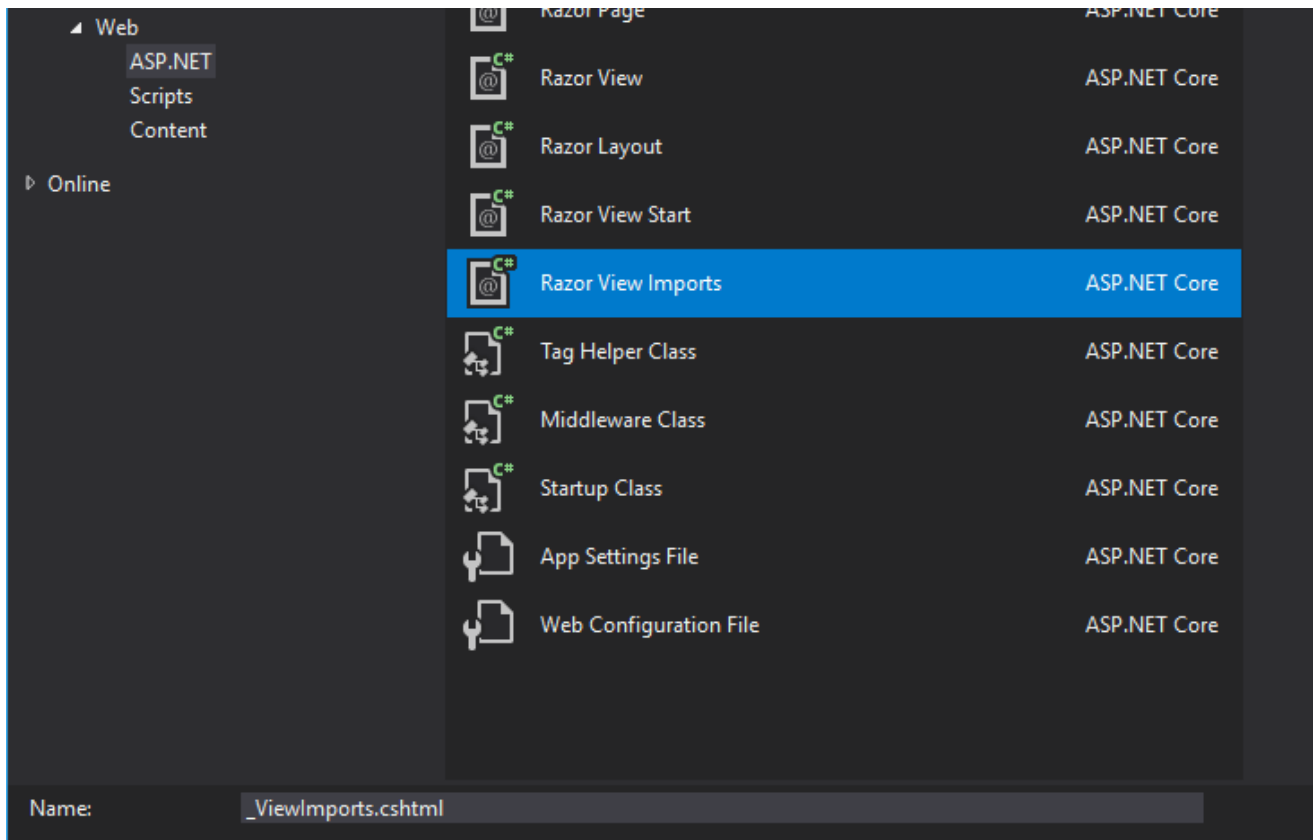
Rather than replacing the tag helpers with valid HTML tags, ASP.NET Core has completely ignored them and let them through exactly as they are in our code...

This is simply because we forget to reference the Tag Helpers at the top of this view...

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

But it would be nice if we didn't have to remember this for every page and the good news is you don't have to.

Add a New Item to your **Views** folder and select **Razor View Imports** as the type (keep the default name).



Pop the TagHelpers reference in there..

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

And that's it, your tag helpers will now work on every view in this folder.