**Forms and Fields in ASP .NET Core**

By Shahed C on February 13, 2019

[6 Replies](https://wakeupandcode.com/forms-and-fields-in-asp-net-core/#comments)

This is the **sixth**of a [series of posts](https://wakeupandcode.com/aspnetcore/#aspnetcore2019) on ASP .NET Core for 2019. In this series, we’ll cover 26 topics over a span of 26 weeks from January through June 2019, titled **A-Z of ASP .NET Core!**

** A – Z of ASP .NET Core!**

**In this Article:**

* F is for Forms (and Fields)
* Tag Helpers for HTML form elements
* Input Tag Helper
* Checkboxes
* Hidden Fields
* Radio Buttons
* Textarea Tag Helper
* Label Tag Helper
* Select Tag Helper
* MVC Sample
* Razor Pages with BindProperty
* References

This article will refer to the following sample code on GitHub:

 Forms And Bindings: <https://github.com/shahedc/FormsAndBindings>

**F is for Forms (and Fields)**

Before Tag Helpers were available, you would have to use [HTML Helper methods](https://www.tutorialspoint.com/asp.net_mvc/asp.net_mvc_helpers.htm) to create forms and their elements in a ASP .NET Core views. This meant that your form could look something like this:

@using (Html.BeginForm())  
{  
 <input />  
}

With the introduction of Tag Helpers, you can now make your web pages much more cleaner. In fact, Tag Helpers work with both MVC Views and Razor Pages. The syntax is much simpler:

<form method="post">

This looks like HTML because it is HTML. You can add additional server-side attributes within the <form> tag for additional features.

<form asp-controller="ControllerName" asp-action="ActionMethod" method="post">

In this above example, you can see how the attributes **asp-controller** and **asp-action** can be used to specify a specific controller name and action method. When these optional attributes are omitted, the current controller and default action method will be used.

Optionally, you can also use a ***named route***, e.g.

<form asp-route="NamedRoute" method="post">

The asp-route attribute will look for a specific route with the name specified. When the form is submitted via HTTP POST, the action method will then attempt to read the form values via a passed values or bound properties.

In a Controller’s class file within an MVC app, you can set an optional **Name** for your action method’s **Route** attribute, as shown below:

[Route("/ControllerName/ActionMethod", Name = "NamedRoute")]  
public IActionResult ActionMethod()  
{  
}

While you won’t find new Tag Helper equivalents for each and every HTML Helper you may have used in the past, you should consider using a Tag Helper wherever possible. You can even create your own [custom Tag Helpers](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/authoring) as well. For more information on custom Tag Helpers, check out the official documentation:

* Author Tag Helpers: <https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/authoring>

**Tag Helpers for HTML form elements**

Below is a list of Tag Helpers with their corresponding HTML form elements:

**Input Tag Helper**

Let’s say you have a model with a couple of fields:

public class MyModel  
{  
 public string MyProperty1 { get; set; }  
 public string MyProperty2 { get; set; }  
}

You can use the following syntax to use an Input Tag Helper with an expression name assigned to the **asp-for** attribute. This allows you to refer to the properties without requiring the “Model.” prefix in your Views and Pages.

@model MyModel   
...  
<!-- Syntax -->  
<input asp-for="<Expression Name>" />  
...  
<!-- Examples -->  
<input asp-for="MyProperty1" />  
<input asp-for="MyProperty2" />

Corresponding to the Input Tag Helper, there are existing HTML Helpers, with some differences:

* Html.TextBox: doesn’t automatically set the type attribute
* Html.TextBoxFor: also doesn’t automatically set the type attribute; strongly typed
* Html.Editor: suitable for collections, complex objects and templates (while Input Tag Helper is not).
* Html.EditorFor: also suitable for collections, complex objects and templates; strongly typed

Since Input Tag Helpers use an inline variable or expression in your .cshtml files, you can assign the value using the @ syntax as shown below:

@{  
   var myValue = "Some Value";  
 }  
 <input asp-for="@myValue" />

This will generate the following textbox input field:

<input type="text" id="myValue" name="myValue" value="Some Value" />

To create more specific fields for email addresses, passwords, etc, you may use data-type attributes on your models to auto-generate the necessary fields. These may include [one of the following enum values](https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.dataannotations.datatype):

* CreditCard
* Currency
* Custom
* Date
* DateTime
* Duration
* EmailAddress
* Html
* ImageUrl
* MultilineText
* Password
* PhoneNumber
* PostalCode
* Text
* Time
* Upload
* Url

// For example:   
[DataType(DataType.Date)]  
public DateTime DateOfBirth { get; set; }

Note that each attribute can be applied on a field for the view/page generator to infer the data type, but is ***not***used for data validation. For validation, you should use the appropriate validation techniques in your code. We will cover validation in a future blog post, but you can refer to the official docs for now:

* Model validation in ASP.NET Core MVC: <https://docs.microsoft.com/en-us/aspnet/core/mvc/models/validation>
* Add validation to an ASP.NET Core Razor Page: <https://docs.microsoft.com/en-us/aspnet/core/tutorials/razor-pages/validation>

**Checkboxes**

Any boolean field in your model will automatically be turned into a checkbox in the HTML form. There is no extra work necessary to specify that the input type is a “checkbox”. In fact, the generated HTML includes the “checkbox” type automatically, sets the “checked” property if checked and wraps it in a label with the appropriate caption. For example, imagine a boolean field named “IsActive”:

// boolean field in a model class  
public bool IsActive { get; set; } <!-- input field in page/view wrapped in label -->  
<label class="form-check-label">  
 <input class="form-check-input" asp-for="IsActive" />   
 @Html.DisplayNameFor(model => model.IsActive)  
</label> <!-- HTML generated for boolean field -->  
<label class="form-check-label">  
<input   
 class="form-check-input"   
 type="checkbox"   
 checked="checked"   
 data-val="true"   
 data-val-required="The IsActive field is required."   
 id="IsActive"   
 name="IsActive"   
 value="true"> IsActive  
</label>

**Hidden Fields**

In case you’re wondering how you can generate a *hidden* <**input**> field, you can simply use the [**HiddenInput**] attribute on your hidden field’s property, as shown below. If you wish, you can explicitly set “**type=hidden**” in your Page/View, but I prefer to set the attribute in the model itself.

// hidden property in model class  
[HiddenInput]   
public string SomeHiddenField { get; set; } = "Some Value"; <!-- hidden field in page/view -->  
<input asp-for="SomeHiddenField" /> <!-- HTML generated for hidden field -->   
<input type="hidden" id="SomeHiddenField" name="SomeHiddenField" value="Some Value">

**Radio Buttons**

For radio buttons, you can create one <**input**> tag for each radio button option, with a reference to a common field, and a unique value for each radio button. Each input element can be wrapped in a label to include a proper (clickable) text caption . You can generate these in a loop or from a collection from dynamically generated radio buttons.

// string property in model class  
public string ExperienceLevel { get; set; } <!-- input fields for radio buttons in page/view -->  
<label><input asp-for="ExperienceLevel" value="N" type="radio" />Novice</label>   
<label><input asp-for="ExperienceLevel" value="B" type="radio"/>Beginner</label>   
<label><input asp-for="ExperienceLevel" value="I" type="radio" />Intermediate</label>   
<label><input asp-for="ExperienceLevel" value="A" type="radio"/>Advanced</label> <!-- HTML generated for radio buttons -->   
<label><input value="N" type="radio" id="ExperienceLevel" name="ExperienceLevel">Novice</label>  
<label><input value="B" type="radio" id="ExperienceLevel" name="ExperienceLevel">Beginner</label>  
<label><input value="I" type="radio" id="ExperienceLevel" name="ExperienceLevel">Intermediate</label>  
<label><input value="A" type="radio" id="ExperienceLevel" name="ExperienceLevel">Advanced</label>

**Textarea Tag Helper**

The multiline <**textarea**> field can be easily represented by a **Textarea** Tag Helper. This is useful for longer strings of text that need to be seen and edited across multiple lines.

public class MyModel  
{  
 [MinLength(5)]  
 [MaxLength(1024)]  
 public string MyLongTextProperty { get; set; }  
}

As you would expect, you can use the following syntax to use a Textarea Tag Helper with an expression name assigned to the **asp-for** attribute.

@model MyModel   
...  
<textarea asp-for="MyLongTextProperty"></textarea>

This will generate the following textarea input field:

<textarea   
 data-val="true"   
 data-val-maxlength="The field ... maximum length of '1024'."   
 data-val-maxlength-max="1024"   
 data-val-minlength="The field ... minimum length of '5'."   
 data-val-minlength-min="5"   
 id="MyLongTextProperty"   
 maxlength="1024"   
 name="MyLongTextProperty"  
></textarea>

Note that the property name and its attributes are used to create that textarea with the necessary id, name, maxlength and data validation settings.

Corresponding to the Textarea Tag Helper, the existing HTML Helper is shown below:

* Html.TextAreaFor

**Label Tag Helper**

The <**label**> field can be represented by a **Label** Tag Helper. A label usually goes hand-in-hand with a specific <input> field, and is essential in creating text captions for more accessible web applications. The **Display** attribute from your model’s fields are used for the label’s displayed text values. *(You could use the* ***DisplayName*** *attribute instead and omit the* ***Name*** *parameter, but it limits your ability to use localized resources.)*

public class MyModel  
{  
 [Display(Name = "Long Text")]  
 public string MyLongTextProperty { get; set; }  
}

You can use the following syntax to use a **Label** Tag Helper along with an **Input** Tag Helper.

@model MyModel   
...  
<label asp-for="MyLongTextProperty"></label>  
<input asp-for="MyLongTextProperty" />

This will generate the following HTML elements:

<label for="MyLongTextProperty">Long Text</label>  
<input type="text" id="MyLongTextProperty" name="MyLongTextProperty" value="">

Note that the property name and its attributes are used to create both the label with its descriptive caption and also the input textbox with the necessary id and name.

Corresponding to the Label Tag Helper, the existing HTML Helper is shown below:

* Html.LabelFor

**Select Tag Helper**

The <**select**> field (with its nested <**option**> fields) can be represented by a Select Tag Helper. This visually represents a dropdown or listbox, from which the user may select one or more options. In your model, you can represent this with a **List<SelectListItem>** of items, made possible by the namespace [Microsoft.AspNetCore.Mvc.Rendering](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.rendering).

...  
using Microsoft.AspNetCore.Mvc.Rendering;  
  
public class MyModel  
{  
 public string MyItem { get; set; }  
   
 public List<SelectListItem> MyItems { get; } = new List<SelectListItem>  
 {  
 new SelectListItem { Value = "Item1", Text = "Item One" },  
 new SelectListItem { Value = "Item2", Text = "Item Two" },  
 new SelectListItem { Value = "Item3", Text = "Item Three" },  
 };  
}

You can use the following syntax to use a Select Tag Helper.

@model MyModel   
...  
<select asp-for="MyItem" asp-items="Model.MyItems"></select>

Note that the **asp-items** attribute ***does*** require a “Model.” prefix, unlike the **asp-for** attribute that we have been using so far. This will generate the following HTML:

<select id="MyItem" name="MyItem">  
 <option value="Item1">Item One</option>  
 <option value="Item2">Item Two</option>  
 <option value="Item3">Item Three</option>  
</select>

Note that the property name and its attributes are used to create both the dropdown list and also the nested options available for selection. For more customization, optgroups and multiple selections, check out the “Select Tag Helper” section in the Tag Helpers documentation at:

* [Select Tag Helper section] Tag Helpers in forms: <https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-2.2#the-select-tag-helper>

Corresponding to the Select Tag Helper, the existing HTML Helpers are shown below:

* Html.DropDownListFor
* Html.ListBoxFor

**MVC Sample**

In the sample repo, you’ll find an [MVC web project](https://github.com/shahedc/FormsAndBindings/tree/master/WebMvc) with various models, views and controllers.

* **Models**: In the “Models” folder, you’ll find a [Human.cs class](https://github.com/shahedc/FormsAndBindings/blob/master/WebMvc/Models/Human.cs) (shown below) with some fields we will use to display HTML form elements.
* **Views**: Within the [“Views” subfolder](https://github.com/shahedc/FormsAndBindings/tree/master/WebMvc/Views), the “Human” subfolder contains auto-generated views for the HumanController’s methods, while the “Attr” subfolder contains simple Index and Detail views to be used by the AttrController class.
* **Controllers**: The [HumanController class](https://github.com/shahedc/FormsAndBindings/blob/master/WebMvc/Controllers/HumanController.cs) was auto-generated for the Human model, while the [AttrController class](https://github.com/shahedc/FormsAndBindings/blob/master/WebMvc/Controllers/AttrController.cs) was manually written to illustrate the use of various attributes (e.g. FromQuery, FromRoute, FromForm) within the action methods.

public class Human  
{  
 public int Id { get; set; }  
 public string FirstName { get; set; }  
 public string LastName { get; set; }  
 public string Address { get; set; }  
 public DateTime DateOfBirth { get; set; }  
 public int FavoriteNumber { get; set; }  
 public bool IsActive { get; set; }  
}

Take a look at the [Create](https://github.com/shahedc/FormsAndBindings/blob/master/WebMvc/Views/Human/Create.cshtml) and [Edit](https://github.com/shahedc/FormsAndBindings/blob/master/WebMvc/Views/Human/Edit.cshtml) views for the Human class, and you’ll recognize familiar sets of <**label**> and <**input**> fields that we discussed earlier.

...  
<div class="form-group">  
 <label asp-for="FirstName" class="control-label"></label>  
 <input asp-for="FirstName" class="form-control" />  
 <span asp-validation-for="FirstName" class="text-danger"></span>  
</div>  
...

To bind these fields, the **Create** and **Edit** methods that respond to HTTP POST both use a [**Bind**] attribute for the **Human** parameter with specific fields to bind from the model:

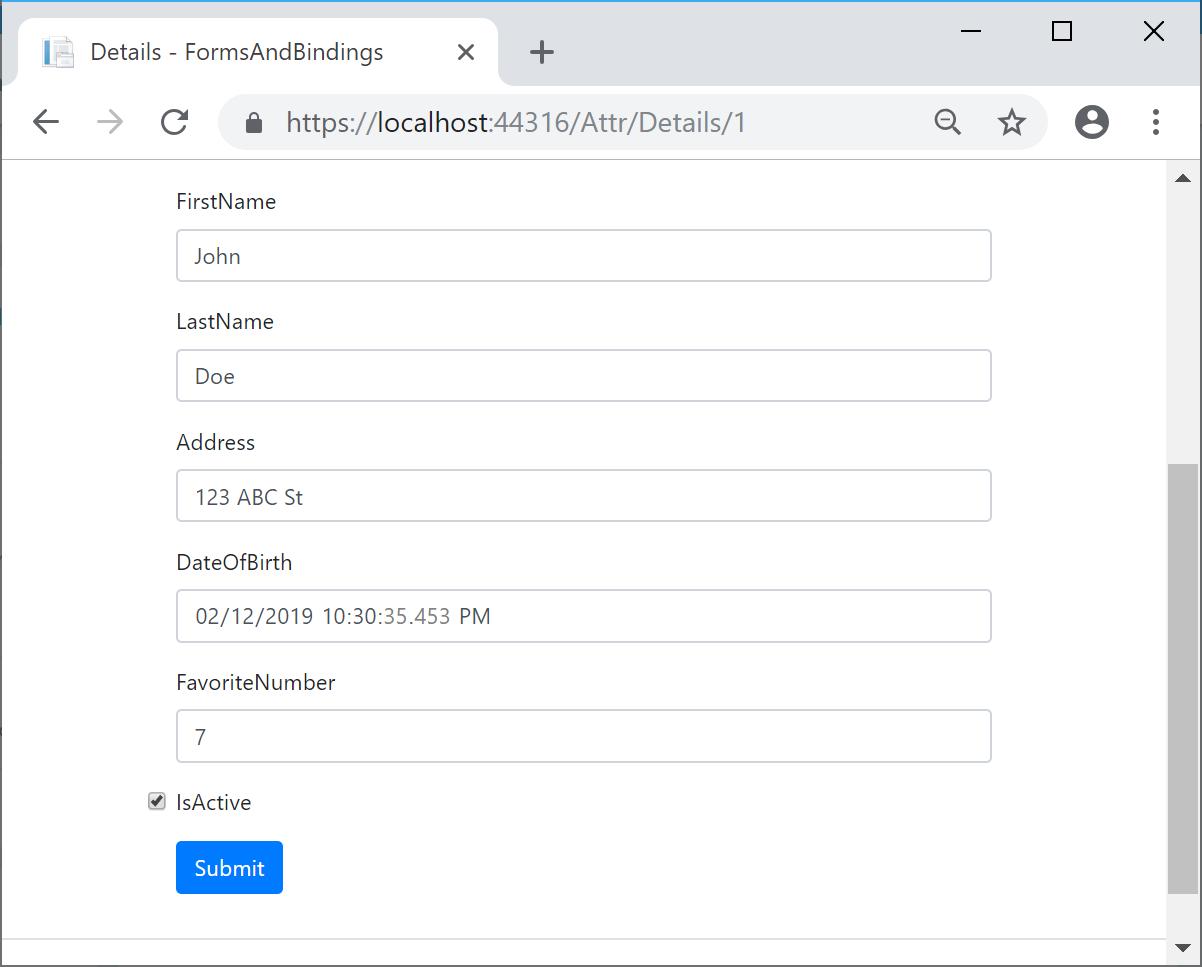
[HttpPost]  
public async Task<IActionResult> Create(  
 [Bind("Id,FirstName,LastName,Address,DateOfBirth,FavoriteNumber,IsActive")] Human human)  
  
...   
[HttpPost]  
public async Task<IActionResult> Edit(  
 int id,   
 [Bind("Id,FirstName,LastName,Address,DateOfBirth,FavoriteNumber,IsActive")] Human human)

In the [AttrController](https://github.com/shahedc/FormsAndBindings/blob/master/WebMvc/Controllers/AttrController.cs) class, you’ll find a different approach to gathering information submitted in an HTML form.

* **FromQuery**: The **Index**() method uses a [**FromQuery**] attribute which can obtain information from the URL’s QueryString parameters
* **FromRoute**: The first **Details**() method uses a [**FromRoute**] attribute which can obtain information from route values.
* **FromForm**: The second **Details**() method uses a [**FromForm**] attribute which links the submitted form fields to the fields of the corresponding model passed to the method.

[HttpGet]  
public IActionResult Index([FromQuery] string humanInfo) { }  
...  
[HttpGet]  
public IActionResult Details([FromRoute] int id) { }  
...  
[HttpPost]  
public IActionResult Details([FromForm] Human human) { }  
...

The first two methods above can be reached via HTTP GET requests by accessing a URL, while the third one can be reached when submitting the Details form with a submit button. Run the sample application to click around and view its behavior.

[](https://wakeupandcode.com/wp-content/uploads/2019/02/Forms-Bindings.png)

<!-- From shared \_Layout.cshtml, link to Index page -->  
<a   
 class="nav-link text-dark"   
 asp-area=""   
 asp-controller="Attr"   
 asp-action="Index"   
 asp-route-humanInfo="John"  
>Attributes</a>  
  
 <!-- From Index.cshtml, link to Details page -->  
<a   
 class="nav-link text-dark"   
 asp-area=""   
 asp-controller="Attr"   
 asp-action="Details"   
 asp-route-id="1"  
>View Details</a> <!-- From Details.cshtml, self-submitting form -->  
<form asp-action="Details">  
...  
<input type="submit" value="Submit" class="btn btn-primary" />  
...  
</form>

Note that <a> tags have an asp-route-xx attribute that can have any text value appended to the end of it. These route parameters, e.g. id and humanInfo, correspond directly to action method parameters seen in **Index**() and **Details()** methods of the **AttrController** class.

**Razor Pages with BindProperty**

Compared to MVC views, the newer Razor Pages make it a lot easier to bind your model properties to your HTML forms. The [**BindProperty**] attribute can be applied to MVC Controllers as well, but is much more effective within Razor Pages.

In the sample repo, you’ll find a [Razor web project](https://github.com/shahedc/FormsAndBindings/tree/master/WebRazor) with multiple subfolders, including Models and Pages.

* Models: In the “Models” folder of the Razor web project, you’ll find a [Human.cs class](https://github.com/shahedc/FormsAndBindings/blob/master/WebRazor/Models/HumanModel.cs) (shown below) with some fields we will use to display HTML form elements.
* Pages: Within the [“Pages” subfolder](https://github.com/shahedc/FormsAndBindings/tree/master/WebRazor/Pages), the “Human” subfolder contains auto-generated Razor Pages along with corresponding .cs classes that contain the necessary Get/Post methods.

Here, the **HumanModel** class is named slightly differently from the aforementioned (MVC) example’s Human class, since the subfolder within Pages is also called Human. *(We would have to rename one or the other to avoid a naming conflict.)*

public class HumanModel  
 {  
 public int Id { get; set; }  
 public string FirstName { get; set; }  
 public string LastName { get; set; }  
 public string Address { get; set; }  
 public DateTime DateOfBirth { get; set; }  
 public int FavoriteNumber { get; set; }  
 public bool IsActive { get; set; }  
 }

This time, take a look at the [Create](https://github.com/shahedc/FormsAndBindings/blob/master/WebRazor/Pages/Human/Create.cshtml) and [Edit](https://github.com/shahedc/FormsAndBindings/blob/master/WebRazor/Pages/Human/Edit.cshtml) pages for the Human class, and you’ll once again recognize familiar sets of <**label**> and <**input**> fields that we discussed earlier.

<div class="form-group">  
 <label asp-for="HumanModel.FirstName" class="control-label"></label>  
 <input asp-for="HumanModel.FirstName" class="form-control" />  
 <span asp-validation-for="HumanModel.FirstName" class="text-danger"></span>  
 </div>

Since there are no controller classes in the Razor web project, let’s take a look at the corresponding C# classes for the **Create** and **Edit** pages, i.e. [Create.cshtml.cs](https://github.com/shahedc/FormsAndBindings/blob/master/WebRazor/Pages/Human/Create.cshtml.cs) and [Edit.cshtml.cs](https://github.com/shahedc/FormsAndBindings/blob/master/WebRazor/Pages/Human/Edit.cshtml.cs). In both of these classes, we’ll find the [**BindProperty**] attribute in use, right after the constructor and before the Get/Post methods.

[BindProperty]  
 public HumanModel HumanModel { get; set; }

This [**BindProperty**] attribute allows you to declaratively bind the HumanModel class and its properties for use by the HTML form in the corresponding Razor Page. This is an opt-in feature that allows to choose which properties to bind. If you wish, you could alternatively bind *all* public properties in the class by using the [**BindProperties**] attribute above the class, instead of above each individual member.

**NOTE**: By default, a Razor Page’s default methods for HTTP GET and HTTP POST are OnGet() and OnPost() respectively. If you wish to use custom page handlers in your HTML forms, you must create custom methods with the prefix OnPost followed by the name of the handler (and optionally followed by the word Async for async methods)

<!-- buttons with custom page handlers -->   
<input type="submit" asp-page-handler="Custom1" value="Submit 1" />  
<input type="submit" asp-page-handler="Custom2" value="Submit 2" /> // action methods in .cs file associated with a Razor Page   
public async Task<IActionResult> OnPostCustom1Async() { }   
public async Task<IActionResult> OnPostCustom2sync() { }

The standard set of Get/Post methods are shown below, from [Create.cshtml.cs](https://github.com/shahedc/FormsAndBindings/blob/master/WebRazor/Pages/Human/Create.cshtml.cs):

public IActionResult OnGet()  
{  
 return Page();  
} public async Task<IActionResult> OnPostAsync()  
{  
 if (!ModelState.IsValid)  
 {  
 return Page();  
 }   
 \_context.HumanModel.Add(HumanModel);  
 await \_context.SaveChangesAsync();   
 return RedirectToPage("./Index");  
}

Note that the **HumanModel** is passed to the DB Context to add it to the database. If you were to remove the aforementioned [**BindProperty**] attribute, HumanModel would be null and the save operation would fail. The above approach only opts in to accepting HTTP POST requests. To enable use of **BindProperty** for HTTP GET requests as well, simply set the optional parameter **SupportsGet** to true, as shown below.

[BindProperty(SupportsGet = true)]

**References**

* Tag Helpers in forms: <https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms>
* Anchor Tag Helper: <https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/built-in/anchor-tag-helper>
* Model Binding in MVC: <https://exceptionnotfound.net/asp-net-core-demystified-model-binding-in-mvc/>
* Model Binding in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/mvc/models/model-binding>
* BindProperty for Controllers or Razor Pages: <http://www.binaryintellect.net/articles/85fb9a1d-6b0d-4d1f-932c-555bd27ba401.aspx>
* Model Binding in Razor Pages: <https://www.learnrazorpages.com/razor-pages/model-binding>
* Introduction to Razor Pages: <https://docs.microsoft.com/en-us/aspnet/core/razor-pages>
* The ASP.NET Core Form Tag Helpers Cheat Sheet: <https://jonhilton.net/aspnet-core-forms-cheat-sheet/>