Chapter 15:
Doing more with the view

Gill Cleeren

@gillcleeren

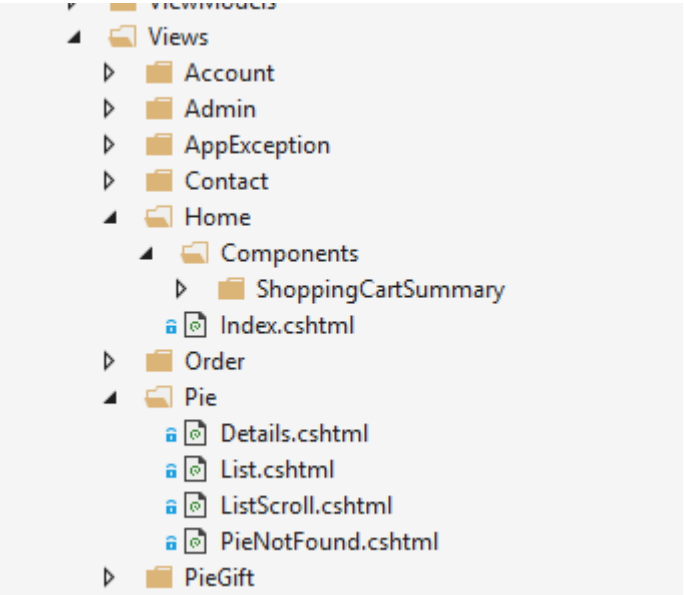
Agenda

- The V of View - recap
- Razor syntax one-on-one
- Building up the pages
- Partial views
- Shopping cart creation end-to-end
- View Components
- Custom tag helpers

The V of View - recap

Hey View, I can see you!

- View contains the presentation part of the user interaction with the app
  - Separation of concerns
- HTML template with code
  - Razor: minimal amount of code inside the markup
    - Easy switch between HTML and logic
  - Tag Helpers: cleaner than HTML Helpers
- *.cshtml files
- Stored in Views folder
  - One controller, one subfolder of views
- Partial views, Layout will help as well



Hey View, I can see you!

- Views specific to a controller live in the Views/[ControllerName] folder
- Views for multiple controllers will live in Views/Shared folder
- View name corresponds to action
  - About action in HomeController corresponds to About.cshtml in Views/Home

Getting to the View

- Views are returned when a ViewResult is returned from an Action
  - Typically done by returning View helper method (part of the Controller base)
    - Overloads allow returning a different view, passing a model…

```
public ViewResult List()
{
        return View(_pieRepository.Pies);
}
```

Getting to the View

- View discovery
  - Specific view?
    - return View("Views/Home/About.cshtml");
    - Typically not recommended, following convention is better
  - Convention-based folder
  - Shared
  - Exception

Passing data to the view

- Data can be passed in several ways
  - Model (strongly-typed view)
    - Can be view model specific for the view

```
@model PieDetailViewModel

<div class="thumbnail">
    <img alt="@Model.Pie.Name" src="@Model.Pie.ImageUrl">
    <div class="caption-full">
        <h3 class="pull-right">@Model.Pie.Price</h3>
        <h3>
            <a href="#">@Model.Pie.Name</a>
        </h3>
        <h4>@Model.Pie.ShortDescription</h4>
        <p>@Model.Pie.LongDescription</p>
        <p class="button">
            <a class="btn btn-primary" asp-controller="ShoppingCart" asp-action="AddToShoppingCart"
               asp-route-pieId="@Model.Pie.PieId">Add to cart</a>
        </p>
    </div>
</div>
```

Passing data to the view

- Data can be passed in several ways
  - Dynamic using ViewBag/ViewData dictionary
    - ViewBag is wrapper around ViewData
  - Requires extra casting step

```
public IActionResult SomeAction()
   {
       ViewData["Greeting"] = "Hello";
       ViewData["Address"]  = new Address()
       {
           Name = "Steve",
           Street = "123 Main St",
           City = "Hudson",
           State = "OH",
           PostalCode = "44236"
       };

       return View();
   }
```

Using ViewData

```
@{
        // Requires cast
        var address = ViewData["Address"] as Address;
    }

    @ViewData["Greeting"] World!

    <address>
        @address.Name<br />
        @address.Street<br />
        @address.City, @address.State @address.PostalCode
    </address>
```

Using ViewBag

```
@ViewBag.Greeting World!

    <address>
        @ViewBag.Address.Name<br />
        @ViewBag.Address.Street<br />
        @ViewBag.Address.City,
@ViewBag.Address.State
@ViewBag.Address.PostalCode
    </address>
```

Razor syntax one-on-one

It’s all about the @

- @ allows transitioning from HTML to C#
  - Razor evaluates C# expressions and renders them in the HTML output
  - If followed by Razor keyword → Razor specific handling
  - Otherwise, plain C#
  - Can also be escaped

```
<p>@@TwitterName</p>
```

```
<p>@DateTime.Now</p>
<p>@DateTime.IsLeapYear(2016)</p>
```

It’s all about the @

- If it’s a code expression, we’ll need @( )

```
<p>Last week this time: @(DateTime.Now - TimeSpan.FromDays(7))</p>
```

- This will fail (spaces)

```
<p>Last week: @DateTime.Now - TimeSpan.FromDays(7)</p>
```

  - Renders

```
<p>Last week: 7/7/2016 4:39:52 PM - TimeSpan.FromDays(7)</p>
```

- Multiline expressions

```
@{
    var joe = new Person("Joe", 33);
 }

<p>Age@(joe.Age)</p>
```

It’s all about the @

- Razor does automatic HTML encoding

```
@("<span>Hello World</span>")
```

- Renders

```
&lt;span&gt;Hello World&lt;/span&gt;
```

- Can be overridden using @Html.Raw
  - Most of the time, not recommended (see security part later in this course)

Code blocks

- Surrounded by @ { }

```
@{
    var output = "Hello World";
}

<p>The rendered result: @output</p>
```

- Typically will transition back to HTML automatically

```
@{
    var inCSharp = true;
    <p>Now in HTML, was in C# @inCSharp</p>
}
```

Controlling the flow

- Razor knows @if, @for, @foreach, @switch…

```
@if (value % 2 == 0)
{
    <p>The value was even</p>
}
```

```
@if (value % 2 == 0)
{
    <p>The value was even</p>
}
else if (value >= 1337)
{
    <p>The value is large.</p>
}
else
{
    <p>The value was not large and is odd.</p>
}
```

Controlling the flow

- Switch

```
@switch (value)
{
    case 1:
        <p>The value is 1!</p>
        break;
    case 1337:
        <p>Your number is 1337!</p>
        break;
    default:
        <p>Your number was not 1 or 1337.</p>
        break;
}
```

Controlling the flow

- For

```
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>
}
```

Controlling the flow

- Foreach

```
@foreach (var person in people)
{
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>
}
```

Controlling the flow

- Do while

```
@{ var i = 0; }
@do
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>

    i++;
} while (i < people.Length);
```

Controlling the flow

- Try catch

```
@try
{
    throw new InvalidOperationException("You did something invalid.");
}
catch (Exception ex)
{
    <p>The exception message: @ex.Message</p>
}
finally
{
    <p>The finally statement.</p>
}
```

@using

- Special case: makes sure that used objects are disposed properly

```
@using (Html.BeginForm())
{
    <div>
        email:
        <input type="email" id="Email" name="Email" value="" />
        <button type="submit"> Register </button>
    </div>
}
```

@using

- Importing namespace

```
@using System.IO
@{
    var dir = Directory.GetCurrentDirectory();
}
<p>@dir</p>
```

Comments in Razor

```
@{
    /* C# comment. */
    // Another C# comment.
}
<!-- HTML comment -->
```

```
@*
 @{
     /* C# comment. */
     // Another C# comment.
 }
 <!-- HTML comment -->
*@
```

Working with the model

- @model directive allows to specify the type for the passed-in Model

```
@model TypeNameOfModel
```

```
@model PieDetailViewModel
```

- Can then be used (including IntelliSense):

```
<div>The name of the pie: @Model.Pie</div>
```

Injection into views using @inject

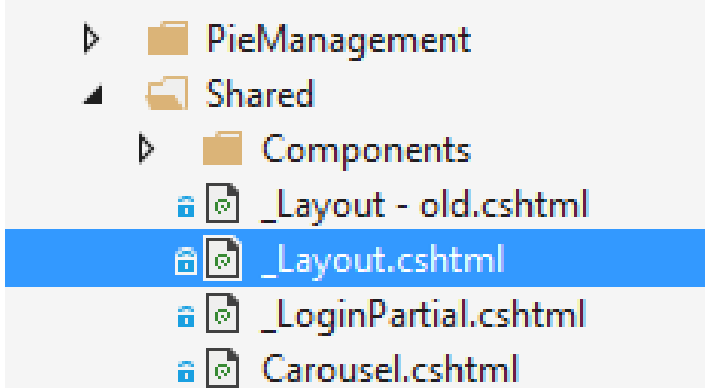
- ASP.NET Core allows DI into the view code
- Useful for view-specific services
  - Localization, authentication…
- Can be injected using @inject

```
@inject SignInManager<ApplicationUser> SignInManager
@inject IAuthorizationService AuthorizationService
@inject UserManager<ApplicationUser> UserManager
```

Building up the pages

Building up the pages

- Layout of web pages typically will contain some common elements
  - Header, navigation, footer…
  - We don’t want these repeated on all pages
- _Layout.cshtml to the rescue
  - Template for views
  - Part of the Shared folder
- Views can use it using
  - Searches own folder, than Shared folder
  - Can also be fully qualified




```
@{
    Layout = "_Layout";
}
```

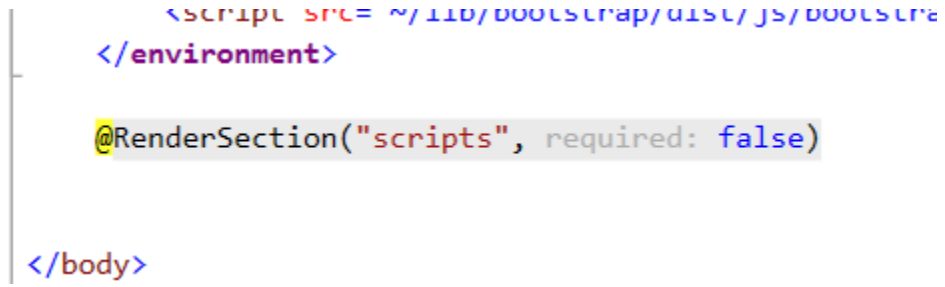
Show me the body!

- @RenderBody is where the view will be added

```
<div class="container">
    <div class="row">
        <div class="col-md-3">
            <p class="lead">
                <img class="img-responsive" alt="Bethany's Pie Shop'" src="~/Images/bethanylogo.png" asp-append-version="true" />
            </p>
        </div>
        <div class="col-md-9">
            @RenderBody()
        </div>
        @await Component.InvokeAsync("SystemStatusPage")
        @await Html.PartialAsync("LanguageSelection")
    </div>
</div>
```

Additional sections

- More blocks can be included using @RenderSection
  - Gives these blocks a place in the layout as well
  - If made required, will throw exception if not found on the view
    - Best to make them non-required!




```
@section Scripts {
     <script type="text/javascript" src="/scripts/main.js"></script>
   }
```

ViewImports

- If we have a lot of Razor directives which are shared across multiple views, we should use a _ViewImports.cshtml
- Supports
  - @addTagHelper
  - @removeTagHelper
  - @tagHelperPrefix
  - @using
  - @model
  - @inherits
  - @inject

```
@using WebApplication1
@using WebApplication1.Models
@using WebApplication1.Models.AccountViewModels
@using WebApplication1.Models.ManageViewModels
@using Microsoft.AspNetCore.Identity
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

ViewImports

- Typically placed in root Views folder
  - Can be overridden on a per-folder level
  - Some are merged such as @addTagHelper
  - Some are overridden such as @model
- Picked up automatically by the Razor engine
  - No action required from our side

ViewStart

- Place code here that needs to run before every view is rendered
  - Selecting the correct template

```
@{
    Layout = "_Layout";
}
```

DEMO

Taking a look at some views

Layout

ViewStart

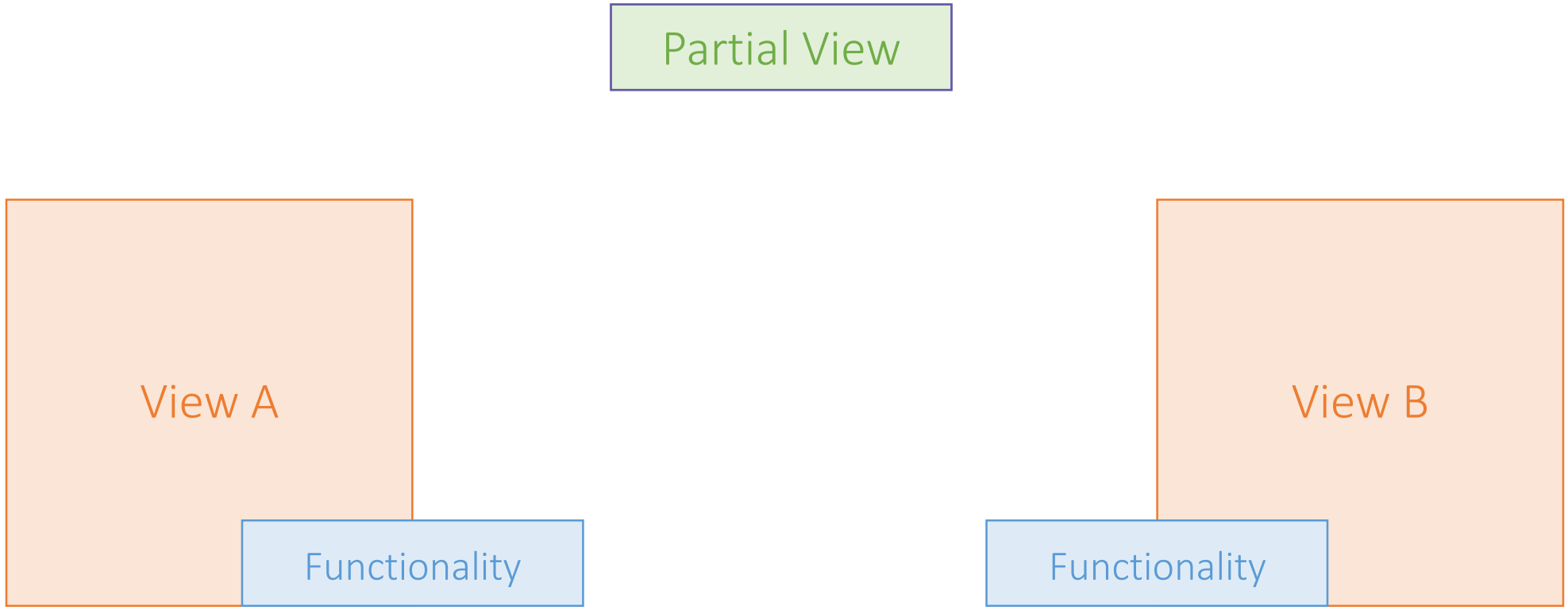
ViewImports

Partial views

Partial views

- View that’s rendered inside of other view
- Resulting HTML is rendered into parent view HTML
- Are just “plain” cshtml files
  - Can be called separately as well
  - Don’t run the ViewStart
- Can help splitting up large complex pages into smaller parts
  - Reduce code duplication
  - Not for real layout (that’s part of the Layout file)

Partial views



Creating a partial view

```
@model Pie
<div>
    <div class="thumbnail">
        <img src="@Model.ImageThumbnailUrl" alt="">
        <div class="caption">
            <h3 class="pull-right">@Model.Price.ToString("c")</h3>
            <p>@Model.ShortDescription</p>
        </div>
    </div>
</div>
```

Using a partial view

```
@foreach (var pie in Model.Pies)
{
    @Html.Partial("PieOverviewSummary", pie)
}
```

```
@foreach (var pie in Model.Pies)
{
    @Html.PartialAsync("PieOverviewSummary", pie)
}
```

Finding the partial view

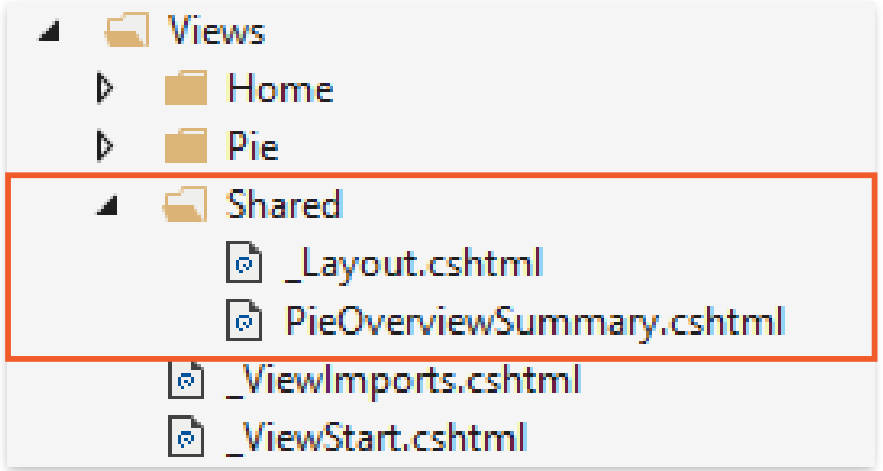
```
// Uses a view in current folder with this name
// If none is found, searches the Shared folder
@Html.Partial("ViewName")

// A view with this name must be in the same folder
@Html.Partial("ViewName.cshtml")

// Locate the view based on the application root
// Paths that start with "/" or "~/" refer to the
application root
@Html.Partial("~/Views/Folder/ViewName.cshtml")
@Html.Partial("/Views/Folder/ViewName.cshtml")

// Locate the view using relative paths
@Html.Partial("../Account/LoginPartial.cshtml")
```

Finding the partial view



Data in the partial view

- By default, partial view gets access to the ViewData of the parent view
- Most of the time, model is needed

```
@Html.Partial("PieOverviewSummary", pie)
```

LAB

Exercise 6. Creating a partial view

Shopping cart creation end-to-end

Shopping cart introduces

- Model
- Controller
- View
- ShoppingCart instance
  - Requires use of Sessions

Enabling session use in Startup

```
public void
ConfigureServices(IServiceCollection services)
{
    services.AddMemoryCache();
    services.AddSession();
}
public void Configure(IApplicationBuilder app,
IHostingEnvironment env,
  ILoggerFactory loggerFactory)
{
    app.UseSession();
}
```

LAB

Exercise 7. Creating the shopping cart

View Components

The problem with Partial Views

- It’s just model binding, nothing more…

```
@model Pie
<div>
  <div class="thumbnail">
    <img src="@Model.ImageThumbnailUrl" alt="">
    <div class="caption">
      <h3 class="pull-right">@Model.Price.ToString("c")</h3>
      <p>@Model.ShortDescription</p>
```

Introducing View Components

- Similar to Partial Views
- Don’t rely on Model Binding
  - We can pass it any data we want
- Renders partial content
- Can have logic and parameters
- Follows SoC pattern
- Linked to parent view
  - Typically invoked from Layout

Introducing View Components

- Used where it's too complex to work with just a Partial View
  - Dynamic navigation menus
  - Tag cloud (where it queries the database)
  - Login panel
  - Shopping cart
  - Recently published articles
  - Sidebar content on a typical blog
  - A login panel that would be rendered on every page and show either the links to log out or log in, depending on the log in state of the user

Creating a View Component

- We get 3 options:
  - Derive from base ViewComponent class
  - [ViewComponent]
  - Class that ends in ViewComponent
- A View Component is public, non-abstract class
- Supports Dependency Injection
- Contains 1 method: Invoke(Async)
  - Returns IViewComponentResult
- Returns a view which typically will get a model from the ViewComponent
  - Results aren't coming from Model Binding!
- Can't be invoked separately
  - Don't handle a request

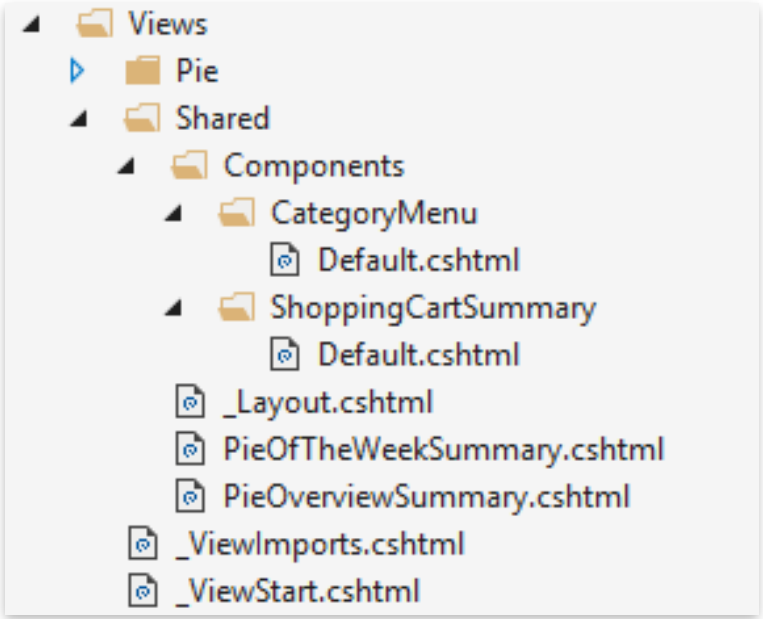
Sample view component

```
public class ShoppingCartSummary : ViewComponent
{
    public IViewComponentResult Invoke()
    {
        return View(model);
    }
}
```

Finding the View Component

- View Components are searched in
  - Views/<controller_name>/Components/<view_component_name>/<view_name>
  - Views/Shared/Components/<view_component_name>/<view_name>
- Filename is typically Default.cshtml

Finding the View Component



Invoking the view component

- Called from view code

```
@await Component.InvokeAsync("ShoppingCartSummary")
```

LAB

Exercises 8 and 9

8. Creating the Home Page

9. Creating a View Component

Custom tag helpers

Tag Helpers enable server-side C# code to participate in creating and rendering HTML elements in Razor files

Previously, we had HTML Helpers

```
@Html.ActionLink("Edit pie", "Edit", new { id=pie.Id })

@Html.DisplayFor(pie => pie.Price)
```

Tag Helpers

- Similar to HTML Helpers
- Cleaner HTML
  - No need for the designer to learn about Razor!
- IntelliSense support
- Built-in collection
- Can be custom-created
- Server-side thing!

Creating a custom tag helper

```
public class EmailTagHelper: TagHelper
{
    public override void Process(
      TagHelperContext context, TagHelperOutput output)
    {
      ...
    }
}
```

Creating a custom tag helper

- Naming convention: targets element with same name as class minus TagHelper
  - Should therefore end in TagHelper (not required, convention)
- Process method is to be overridden
  - Contains code that will be executed by the Tag Helper
- Context gives access to what the tag is working with

Registering the tag helper

```
@using BethanysPieShop.Models
@using BethanysPieShop.ViewModels
@addTagHelper BethanysPieShop.TagHelpers.*, BethanysPieShop
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Using the tag helper

```
<email address="info@@bethanyspieshop.com"
content="Contact us"></email>
```

LAB

Exercise 10. Creating a tag helper

Summary

- Views love Razor
- Tag helpers make HTML cleaner in ASP.NET Core MVC
- View Components go a step further than Partial Views
- HTML Helpers are still supported