RENDERING HTML USING RAZOR VIEWS

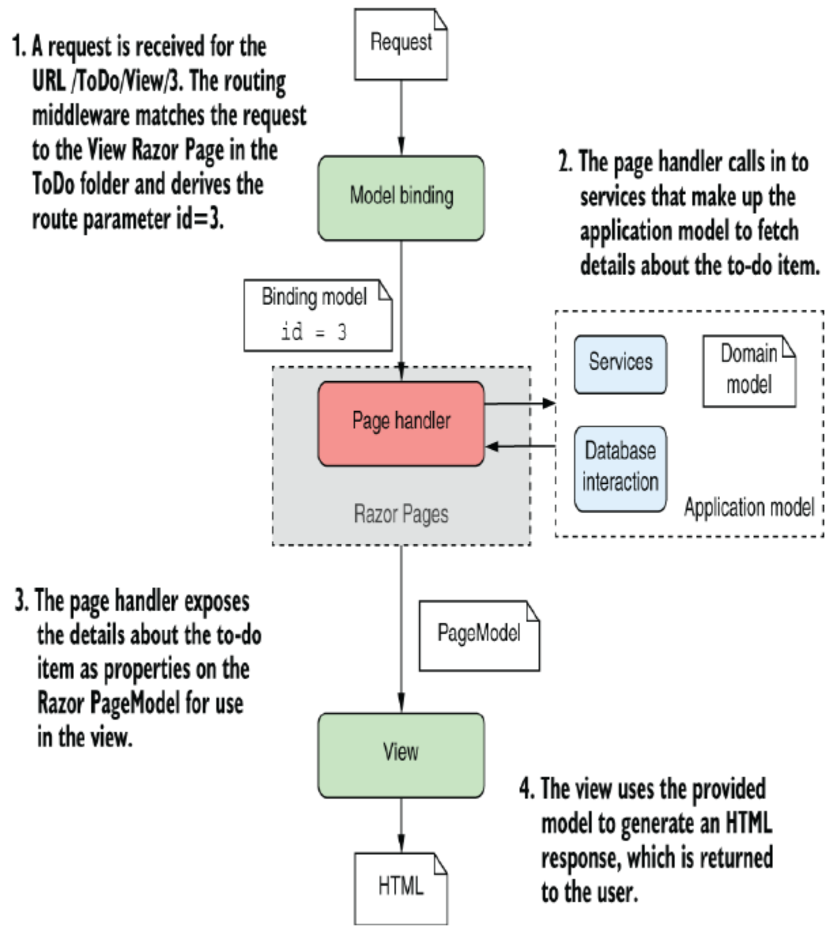
Kamal Beydoun
Lebanese University – Faculty of Sciences I
Kamal.beydoun@ul.edu.lb

REVIEW

Razor Pages—Razor Pages generally refers to the page-based paradigm that combines routing, model binding, and HTML generation using Razor views.

- **Razor Page**—A single Razor Page represents a **single page or endpoint**. It typically consists of two files: **a** .cshtml file containing the Razor view and a .cshtml.cs file containing the page's PageModel.
- **PageModel**—where most of the action happens.
  - It's where you define the **binding models** for a page, which extracts data from the incoming request.
  - It's also where you define the **page's page handlers**.
- **Page handler**—Each Razor Page typically handles a single route, but it can handle multiple HTTP verbs such as GET and POST. Each page handler typically handles a single HTTP verb.
- **Razor view**—Razor views (also called Razor templates) are used to generate HTML.
  - They are typically used in the **final stage of a Razor Page** to generate the HTML response to send back to the user.

VIEWS: RENDERING THE USER INTERFACE


1. A request is received for the URL /ToDo/View/3. The routing middleware matches the request to the View Razor Page in the ToDo folder and derives the route parameter id=3.
Request
Model binding
2. The page handler calls in to services that make up the application model to fetch details about the to-do item.
Binding model
id = 3
Page handler
Razor Pages
Services
Domain model
Database interaction
Application model
3. The page handler exposes the details about the to-do item as properties on the Razor PageModel for use in the view.
PageModel
View
4. The view uses the provided model to generate an HTML response, which is returned to the user.
HTML

VIEWS: RENDERING THE USER INTERFACE

The PageModel contains the data you wish to display on the page.

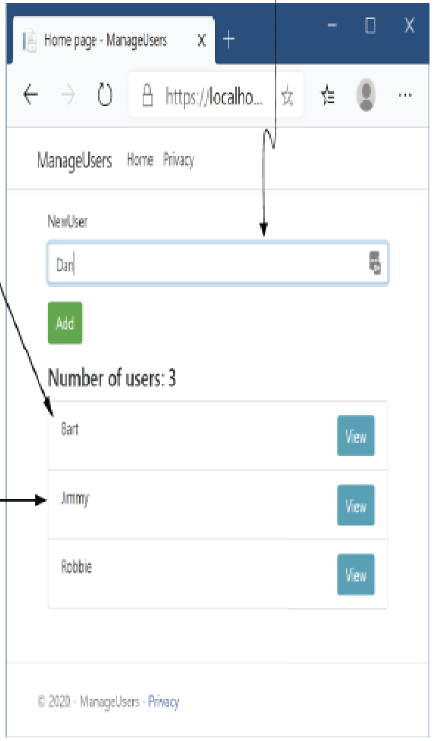
```
Model.ExistingUsers = new[] {
 "Andrew",
 "Robbie",
 "Jimmy",
 "Bart"
};
```

Razor markup describes how to display this data using a mixture of HTML and C#.

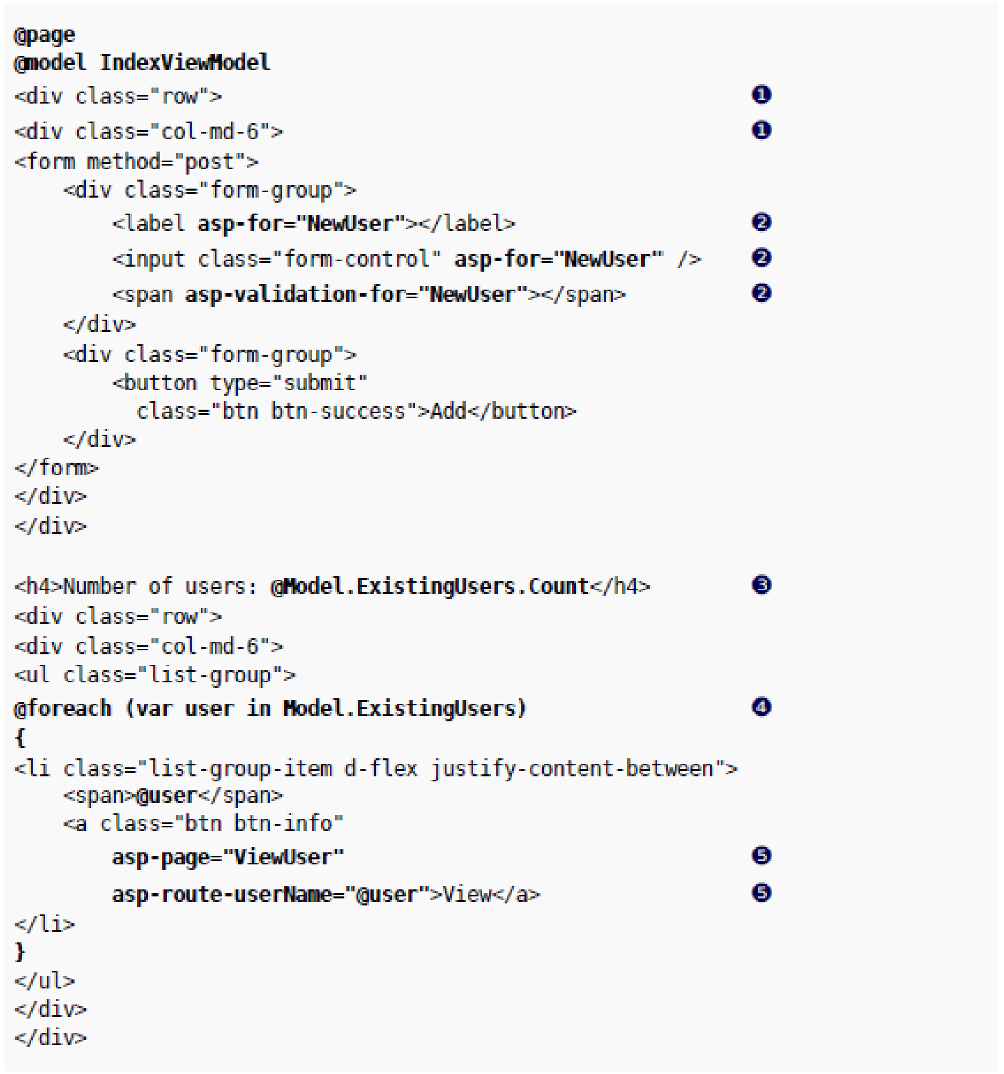
```
@foreach(var user in Model.ExistingUsers)
{
 <li>
 <span>@user</span>
<button>View</button>
 </li>
}
```

By combining the data in your view model with the Razor markup, HTML can be generated dynamically, instead of being fixed at compile time.

Form elements can be used to send values back to the application.



VIEWS: RENDERING THE USER INTERFACE



```
@page
@model IndexViewModel
<div class="row">                                               ❶
<div class="col-md-6">                                          ❶
<form method="post">
    <div class="form-group">
        <label asp-for="NewUser"></label>                       ❷
        <input class="form-control" asp-for="NewUser" />        ❷
        <span asp-validation-for="NewUser"></span>              ❷
    </div>
    <div class="form-group">
        <button type="submit"
          class="btn btn-success">Add</button>
    </div>
</form>
</div>
</div>

<h4>Number of users: @Model.ExistingUsers.Count</h4>           ❸
<div class="row">
<div class="col-md-6">
<ul class="list-group">
@foreach (var user in Model.ExistingUsers)                      ❹
{
<li class="list-group-item d-flex justify-content-between">
    <span>@user</span>
    <a class="btn btn-info"
        asp-page="ViewUser"                                     ❺
        asp-route-userName="@user">View</a>                     ❺
</li>
}
</ul>
</div>
</div>
```

RAZOR VIEWS AND CODE-BEHIND

- The Razor view contains the **@page** directive, which makes it a Razor Page.
  - Without this directive, the Razor Pages framework will not route requests to the page, and the file is ignored for most purposes.
- Even though the .cshtml and .cshtml.cs files have the same name, such as ToDoItem.cshtml and ToDoItem.cshtml.cs, **it's not the filename that's linking them**.
- At the top of each Razor Page, after the @page directive, is the **@model directive** with a Type, indicating which PageModel is associated with the Razor view.

```
@page
@model ToDoItemModel
```

INTRODUCING RAZOR TEMPLATES

```
@page
@{                                                          ❶
    var tasks = new List<string>                            ❶
      { "Buy milk", "Buy eggs", "Buy bread" };              ❶
}                                                           ❶
<h1>Tasks to complete</h1>                                  ❷
<ul>
@for(var i=0; i< tasks.Count; i++)                          ❸
{                                                           ❸
  var task = tasks[i];                                      ❸
  <li>@i - @task</li>                                       ❸
}                                                           ❸
</ul>
```

❶ Arbitrary C# can be executed in a template. Variables remain in scope throughout the page.

❷ Standard HTML markup will be rendered to the output unchanged.

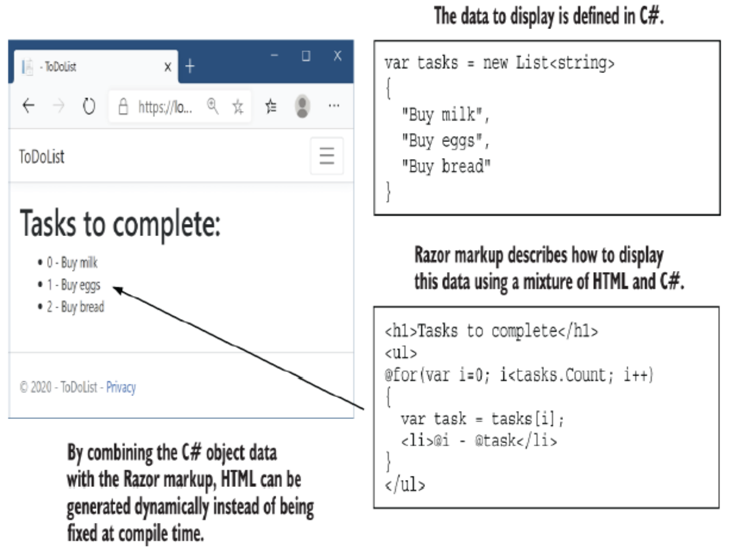
❸ Mixing C# and HTML allows you to create HTML dynamically at runtime.

```
<h1>Tasks to complete</h1>    ❶
<ul>                          ❶
  <li>0 - Buy milk</li>       ❷
  <li>1 - Buy eggs</li>       ❷
  <li>2 - Buy bread</li>      ❷
</ul>                         ❸
```

❶ HTML from the Razor template is written directly to the output.

❷ The <li> elements are generated dynamically by the for loop, based on the data provided.

❸ HTML from the Razor template is written directly to the output.


The data to display is defined in C#.
ToDoList
https://lo...
ToDoList
Tasks to complete:
0 - Buy milk
1 - Buy eggs
2 - Buy bread
© 2020 - ToDoList - Privacy
Razor markup describes how to display this data using a mixture of HTML and C#.
By combining the C# object data with the Razor markup, HTML can be generated dynamically instead of being fixed at compile time.

```
var tasks = new List<string>
{
  "Buy milk",
  "Buy eggs",
  "Buy bread"
}
```

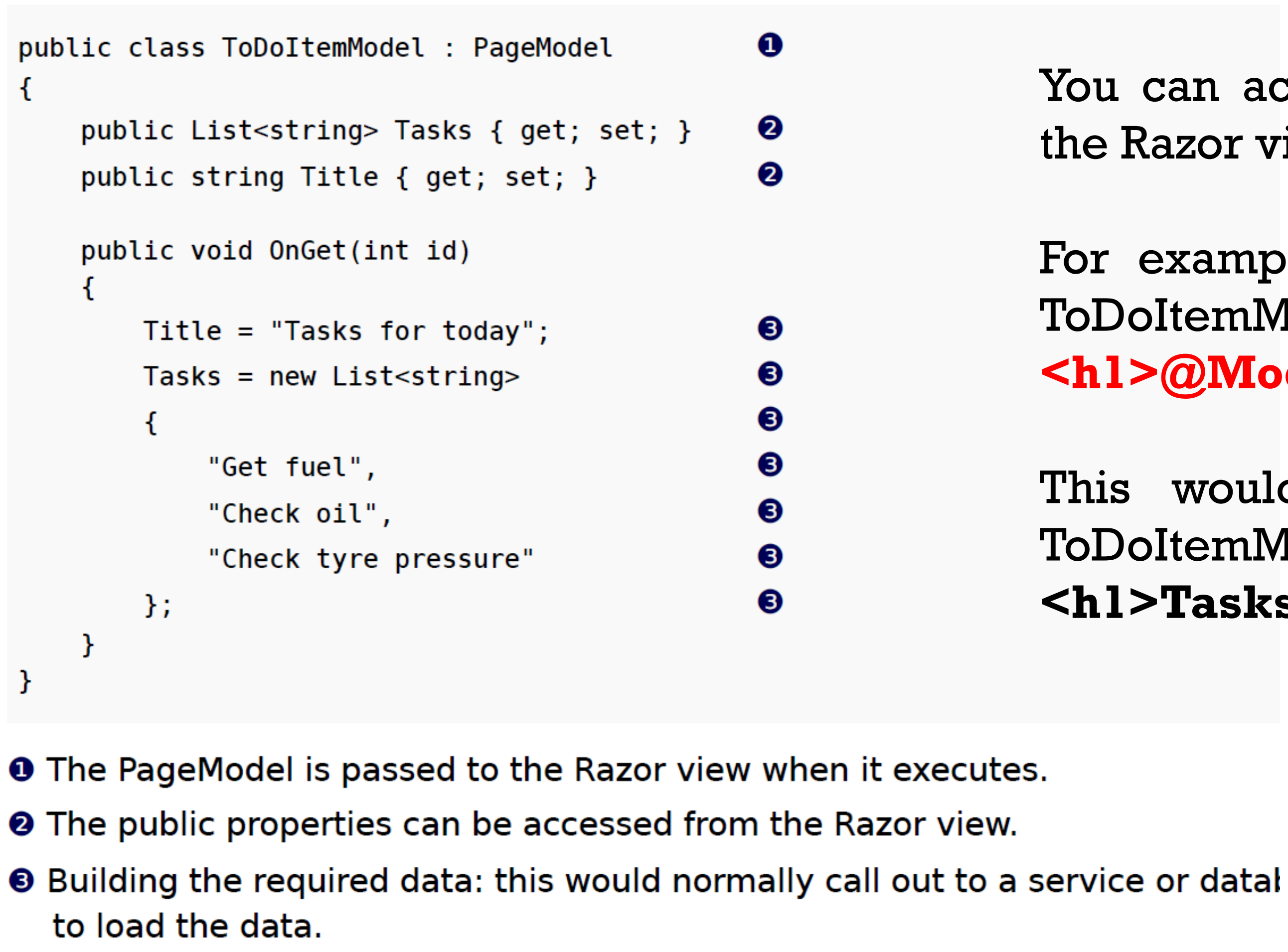
```
<h1>Tasks to complete</h1>
<ul>
@for(var i=0; i<tasks.Count; i++)
{
  var task = tasks[i];
  <li>@i - @task</li>
}
</ul>
```

PASSING DATA TO VIEWS

You should use the mechanisms in the following order:

- **PageModel properties**—You should generally expose any data that needs to be displayed as **properties** on your PageModel. The PageModel object is **available** in the view when it's rendered.
- **ViewData**—This is a dictionary of objects with string keys that can be used to pass arbitrary data from the page handler to the view. In addition, it allows you to pass data to layout files.
- **TempData**—TempData is a dictionary of objects with string keys, similar to ViewData, that is stored until it's read in a **different request**.
  - This is commonly used to temporarily persist data when using the POST-REDIRECT-GET pattern. By default TempData stores the data in an encrypted cookie.
- **HttpContext**—Technically, the HttpContext object is available in both the page handler and Razor view, so you could use it to transfer data between them.
- **@inject services**—You can use dependency injection (DI) to make services available in your views, though this should normally be used **carefully**.
  - Using the directive **@inject** Service myService injects a variable called myService of type Service from the DI container, which you can use in your Razor view.

PASSING DATA TO VIEWS - PROPERTIES



```
public class ToDoItemModel : PageModel                    ❶
{
    public List<string> Tasks { get; set; }               ❷
    public string Title { get; set; }                     ❷

    public void OnGet(int id)
    {
        Title = "Tasks for today";                        ❸
        Tasks = new List<string>                          ❸
        {                                                 ❸
            "Get fuel",                                   ❸
            "Check oil",                                  ❸
            "Check tyre pressure"                         ❸
        };                                                ❸
    }
}
```

❶ The PageModel is passed to the Razor view when it executes.

❷ The public properties can be accessed from the Razor view.

❸ Building the required data: this would normally call out to a service or datal
to load the data.

You can access the PageModel instance itself from the Razor view using the Model property.

For example, to display the Title property of the ToDoItemModel in the Razor view, you'd use **<h1>@Model.Title</h1>.**

This would render the string provided in the ToDoItemModel.Title property, producing the **<h1>Tasks for today</h1>** HTML.

PASSING DATA TO VIEWS - VIEWDATA

- Parctical, when you want to pass data between view layouts.

```
@{
    ViewData["Title"] = "Home Page";
}
<h2>@ViewData["Title"].</h2>
```

```
public class IndexModel: PageModel
{
    [ViewData]                                  ❶
    public string Title { get; set; }

    public void OnGet()
    {
        Title = "Home Page";                    ❷
        ViewData["Subtitle"] = "Welcome";       ❸
    }
}
```

❶ Properties marked with the [ViewData] attribute are set in the ViewData.

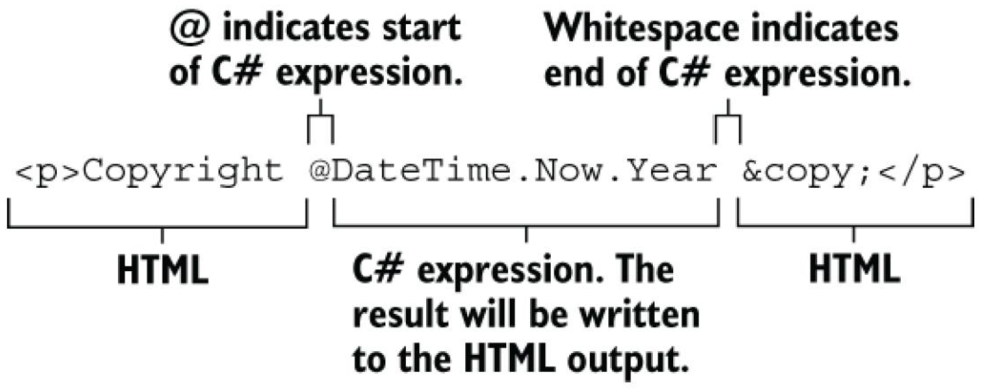
❷ The value of ViewData[“Title”] will be set to “Home Page”.

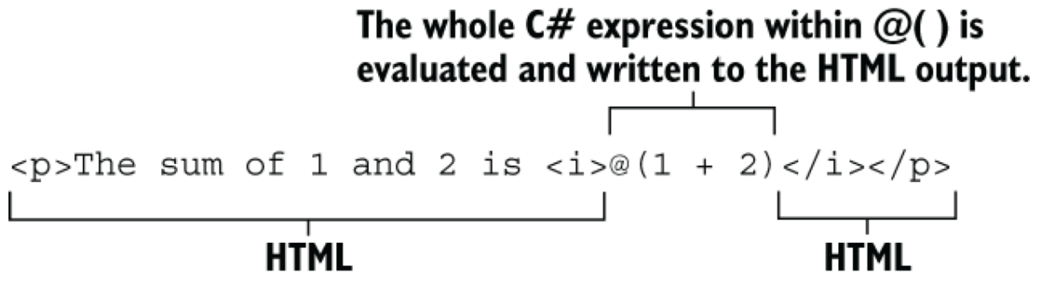
❸ You can set keys in the ViewData dictionary directly.

```
<h1>@ViewData["Title"]</h3>
<h2>@ViewData["Subtitle"]</h3>
```

CREATING DYNAMIC WEB PAGES WITH RAZOR - USING C# IN RAZOR TEMPLATE

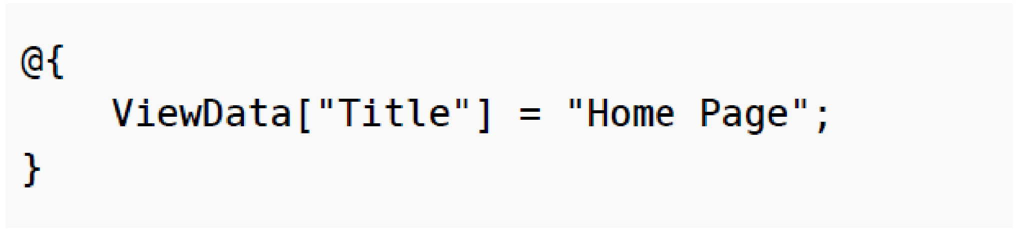
- If the code is a single statement, you can use the @ symbol to indicate you want to write the result to the HTML output


@ indicates start of C# expression.
Whitespace indicates end of C# expression.
<p>Copyright @DateTime.Now.Year &copy;</p>
HTML
C# expression. The result will be written to the HTML output.
HTML


The whole C# expression within @( ) is evaluated and written to the HTML output.
<p>The sum of 1 and 2 is <i>@(1 + 2)</i></p>
HTML
HTML

CREATING DYNAMIC WEB PAGES WITH RAZOR - USING C# IN RAZOR TEMPLATE

- Razor code block, which is normal C# code, identified by the @{} structure.
  - it’s all compiled as though you’d written it in any other normal C# file



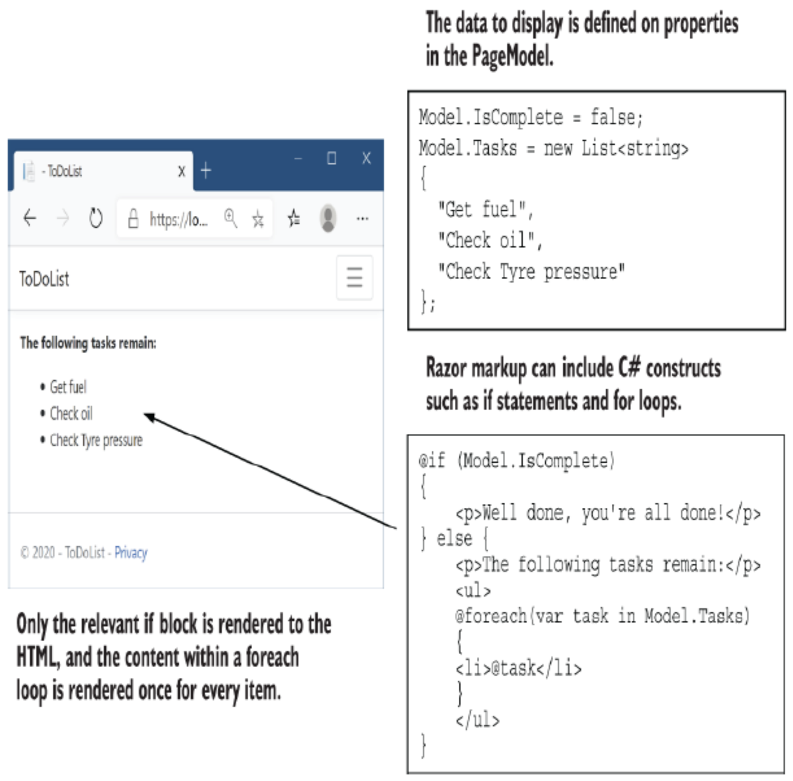
```
@{
    ViewData["Title"] = "Home Page";
}
```

CREATING DYNAMIC WEB PAGES WITH RAZOR - ADDING LOOPS AND CONDITIONALS TO RAZOR TEMPLATES

```
@page
@model ToDoItemModel                                              ❶
<div>
```

```
    @if (Model.IsComplete)
    {                                                             ❷
        <strong>Well done, you’re all done!</strong>              ❷
    }                                                             ❷
    else
    {
        <strong>The following tasks remain:</strong>
        <ul>
            @foreach (var task in Model.Tasks)                    ❸
            {
                <li>@task</li>                                    ❹
            }
        </ul>
    }
</div>
```

❶ The @model directive indicates the type of PageModel in Model.
❷ The if control structure checks the value of the PageModel’s IsComplete property at runtime.
❸ The foreach structure will generate the <li> elements once for each task in Model.Tasks.
❹ A Razor expression is used to write the task to the HTML output.



The data to display is defined on properties in the PageModel.

```
Model.IsComplete = false;
Model.Tasks = new List<string>
{
  "Get fuel",
  "Check oil",
  "Check Tyre pressure"
};
```

Razor markup can include C# constructs such as if statements and for loops.

```
@if (Model.IsComplete)
{
    <p>Well done, you're all done!</p>
} else {
    <p>The following tasks remain:</p>
    <ul>
    @foreach(var task in Model.Tasks)
    {
    <li>@task</li>
    }
    </ul>
}
```

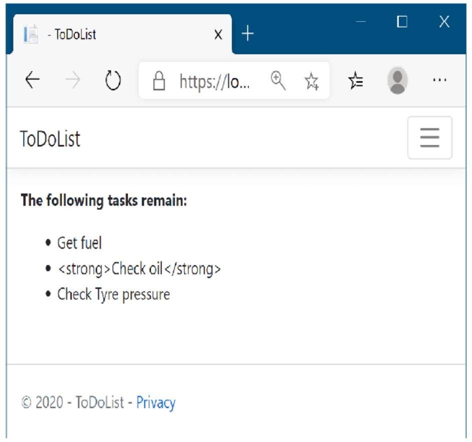
Only the relevant if block is rendered to the HTML, and the content within a foreach loop is rendered once for every item.

CREATING DYNAMIC WEB PAGES WITH RAZOR - RENDERING HTML WITH RAW

- What if the task variable contains **HTML** you want to display, so instead of "Check oil" it contains “ <strong>Check oil</strong>"?

```
<li><strong>Check oil</strong></li>
```

```
<li>&lt;strong&gt;Check oil&lt;/strong&gt;</li>
```

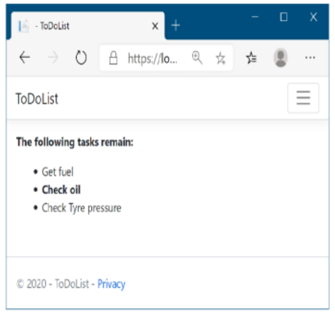


The following tasks remain:

- Get fuel
- <strong>Check oil</strong>
- Check Tyre pressure

© 2020 - ToDoList - Privacy

```
<li>@Html.Raw(task)</li>
```

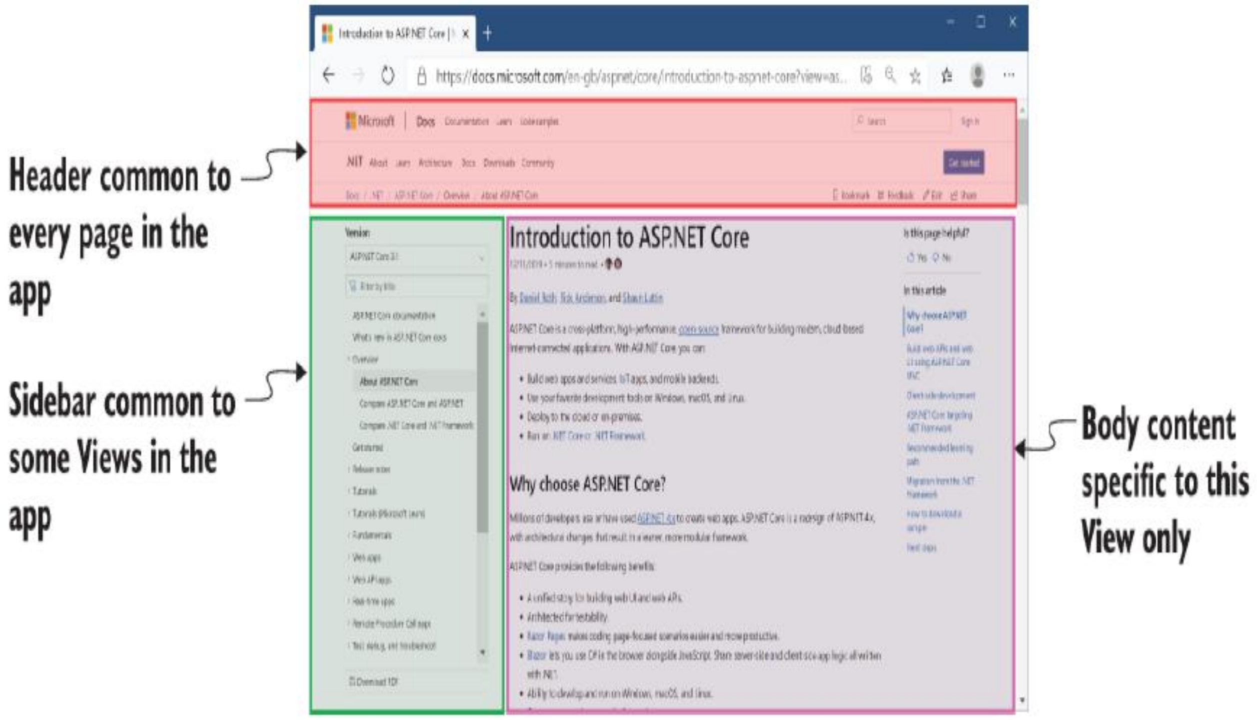


The following tasks remain:

- Get fuel
- **Check oil**
- Check Tyre pressure

© 2020 - ToDoList - Privacy

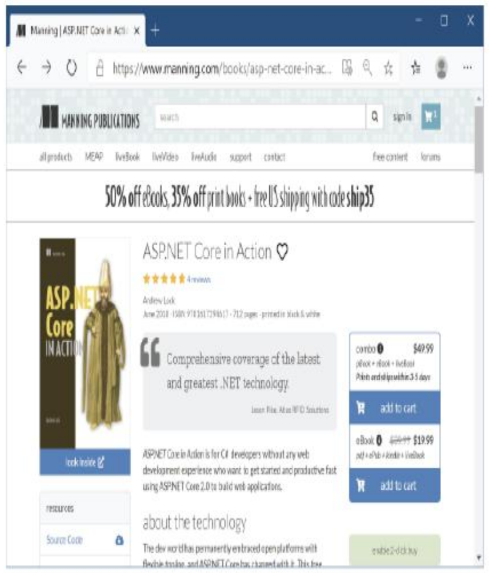
LAYOUTS, PARTIAL VIEWS, AND _VIEWSTART


Header common to every page in the app
Sidebar common to some Views in the app
Body content specific to this View only
Introduction to ASP.NET Core
Why choose ASP.NET Core?

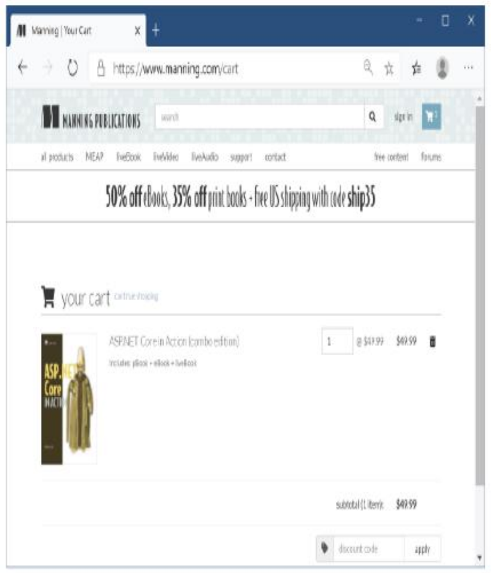
A **layout** in Razor is a template that includes common code.
It **can't** be rendered directly, but it can be rendered in conjunction with normal Razor views.

USING LAYOUTS FOR SHARED MARKUP

- An ASP.NET Core app can have **multiple layouts**, and layouts can reference other layouts.
- A common use for this is to have different layouts for different sections of your application.


https://www.manning.com/books/asp-net-core-in-ac...
MANNING PUBLICATIONS
50% off eBooks, 35% off print books - free US shipping with code ship35
ASP.NET Core in Action
Comprehensive coverage of the latest and greatest .NET technology.
add to cart
about the technology

Three-column layout


https://www.manning.com/cart
MANNING PUBLICATIONS
50% off eBooks, 35% off print books - free US shipping with code ship35
your cart
ASP.NET Core in Action (combo edition)
$49.99
apply

Single-column layout

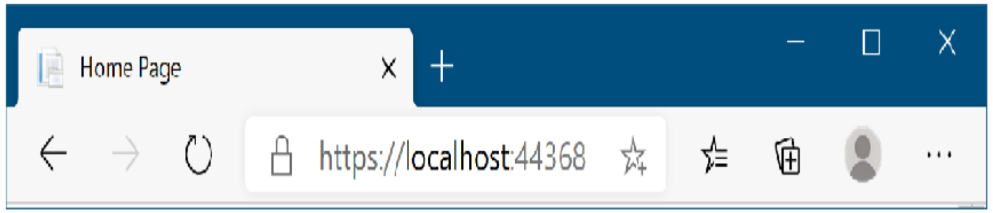
USING LAYOUTS FOR SHARED MARKUP

- A common convention is to prefix your layout files with an underscore (_) to distinguish them from standard Razor templates in your Pages folder.
- Placing them in Pages/**Shared** means you can refer to them by the short name, such as "**_Layout**", without having to specify the full path to the layout file.
- A layout file looks similar to a normal Razor template, with one exception: **every layout must call the @RenderBody() function**.
  - This tells the templating engine where to insert the content from the child views.

USING LAYOUTS FOR SHARED MARKUP

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>@ViewData["Title"]</title>                    ❶
```

```
    <link rel="stylesheet" href="~/css/site.css" />     ❷
</head>
<body>
    @RenderBody()                                        ❸
</body>
</html>
```



```
@{
    Layout = "_Layout";                    ❶
    ViewData["Title"] = "Home Page";       ❷
}
<h1>@ViewData["Title"]</h1>                ❸
<p>This is the home page</p>               ❸
```

❶ Sets the layout for the page to _Layout.cshtml
❷ ViewData is a convenient way of passing data from a Razor view to the layout.
❸ The content in the Razor view to render inside the layout

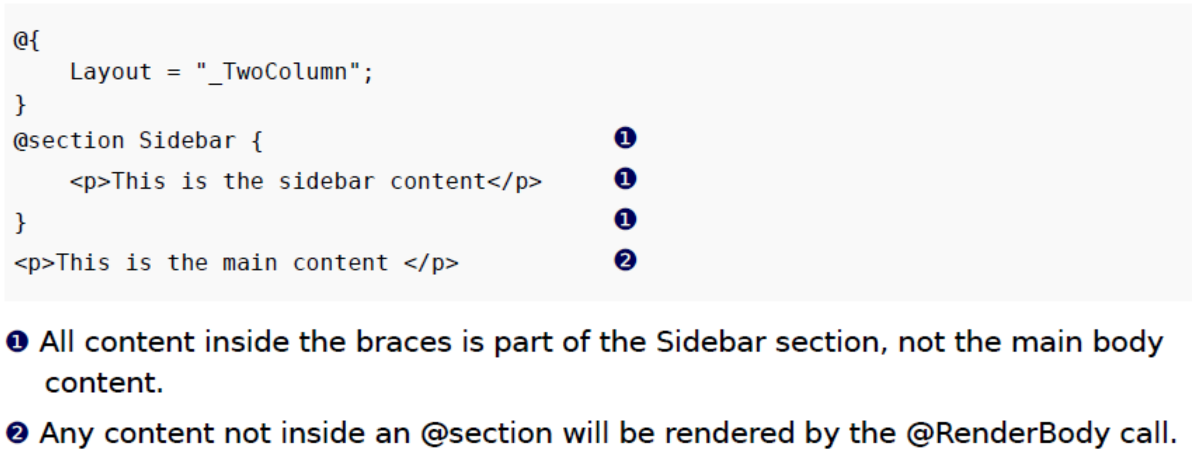
```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Home Page</title>          ❶
    <link rel="stylesheet" href="/css/site.css" />
</head>
<body>
    <h1>Home Page</h1>                ❷
    <p>This is the home page</p>      ❷
</body>
<html>
```

❶ ViewData set in the view is used to render the layout.
❷ The RenderBody call renders the contents of the view.

OVERRIDING PARENT LAYOUTS USING SECTIONS

- **Sections** provide a way of organizing where view elements should be placed within a layout.
  - defined in the **view** using an **@section** definition that be placed anywhere in the file, top or bottom, wherever is convenient.
- The section is rendered in the **parent layout** with a call to @**RenderSection()**.
- Sections can be either required or optional.

OVERRIDING PARENT LAYOUTS USING SECTIONS

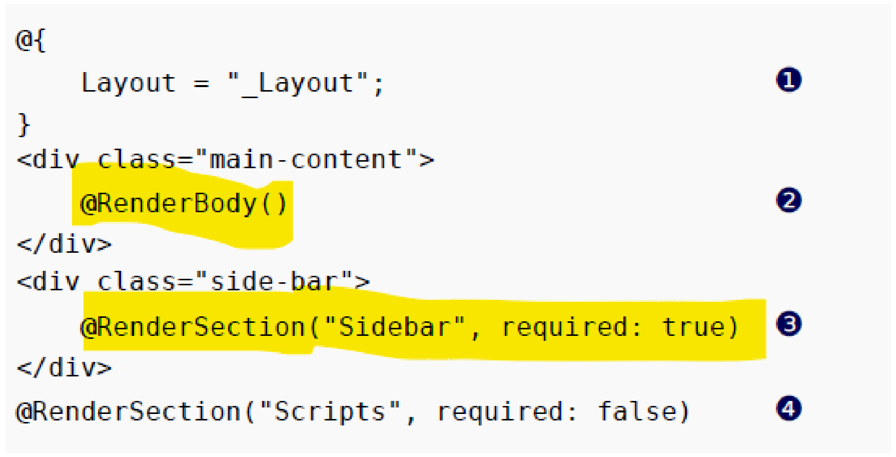


```
@{
    Layout = "_TwoColumn";
}
@section Sidebar {                              ❶
    <p>This is the sidebar content</p>          ❶
}                                               ❶
<p>This is the main content </p>                ❷
```

❶ All content inside the braces is part of the Sidebar section, not the main body content.

❷ Any content not inside an @section will be rendered by the @RenderBody call.

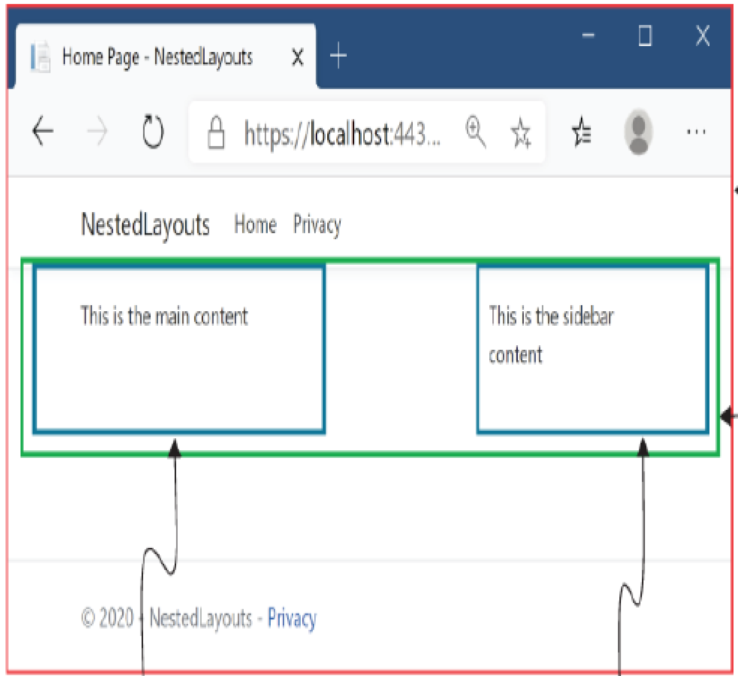
Defining a section in a view template



```
@{
    Layout = "_Layout";                                  ❶
}
<div class="main-content">
    @RenderBody()                                        ❷
</div>
<div class="side-bar">
    @RenderSection("Sidebar", required: true)            ❸
</div>
@RenderSection("Scripts", required: false)               ❹
```

Rendering a section in a layout file, _TwoColumn.cshtml

OVERRIDING PARENT LAYOUTS USING SECTIONS



This is the main content

This is the sidebar content

© 2020 - NestedLayouts - Privacy

_Layout.cshtml defines the HTML in the Header and footer.

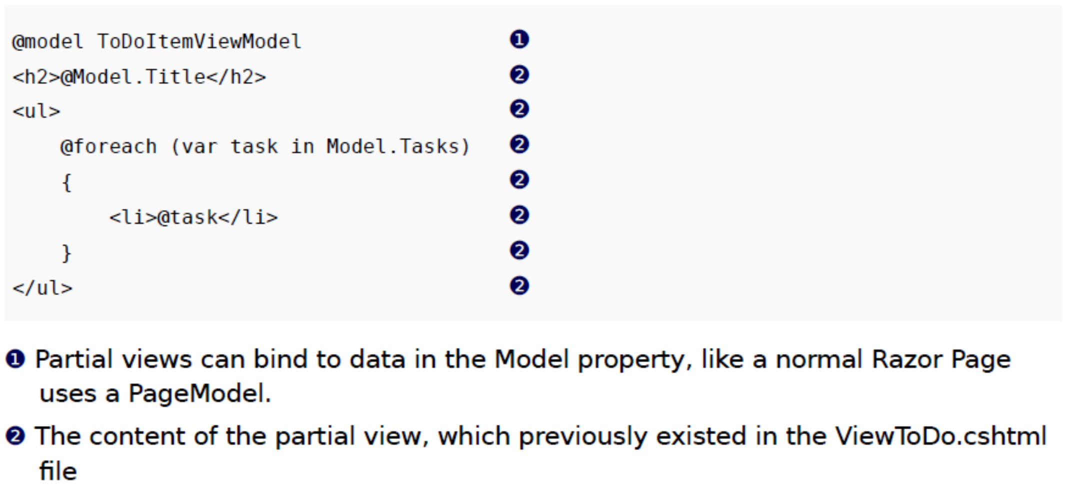
_TwoColumn.cshtml is rendered inside _Layout.cshtml.

The main content of the View is rendered in _TwoColumn.cshtml by RenderBody.

The sidebar content of the View is rendered in _TwoColumn.cshtml by RenderSection(Sidebar).

USING PARTIAL VIEWS TO ENCAPSULATE MARKUP

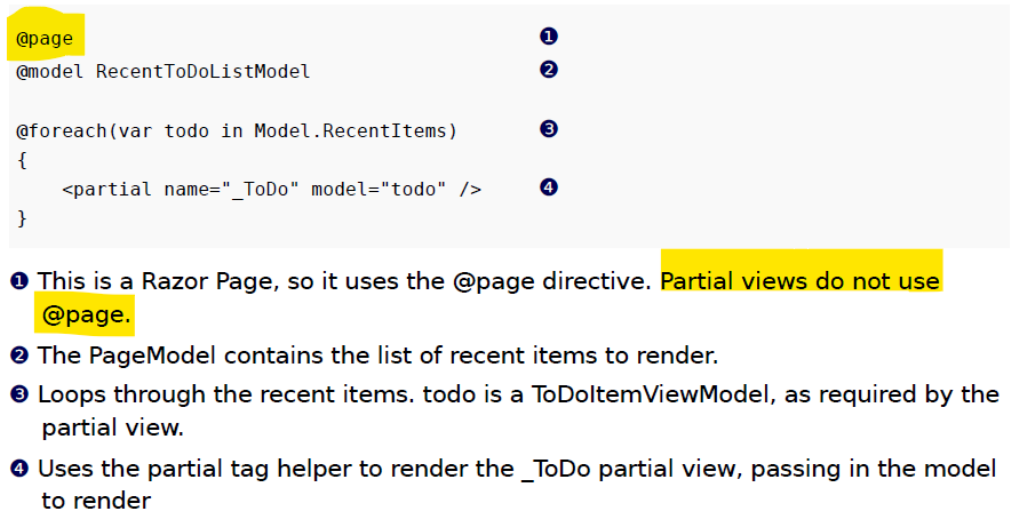
- **Partial views** are exactly what they sound like: part of a view. They provide a means of breaking up a larger view into smaller, reusable chunks.
- Partial views are rendered using the **<partial />** Tag Helper



```
@model ToDoItemViewModel                ❶
<h2>@Model.Title</h2>                   ❷
<ul>                                    ❷
    @foreach (var task in Model.Tasks)  ❷
    {                                   ❷
        <li>@task</li>                  ❷
    }                                   ❷
</ul>                                   ❷
```

❶ Partial views can bind to data in the Model property, like a normal Razor Page uses a PageModel.

❷ The content of the partial view, which previously existed in the ViewToDo.cshtml file



```
@page                                             ❶
@model RecentToDoListModel                        ❷

@foreach(var todo in Model.RecentItems)           ❸
{
    <partial name="_ToDo" model="todo" />         ❹
}
```

❶ This is a Razor Page, so it uses the @page directive. Partial views do not use @page.

❷ The PageModel contains the list of recent items to render.

❸ Loops through the recent items. todo is a ToDoItemViewModel, as required by the partial view.

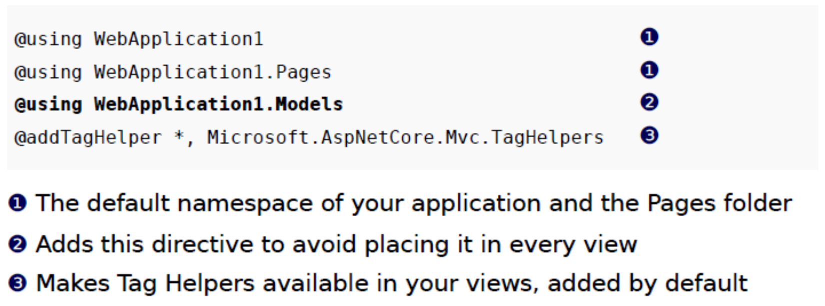
❹ Uses the partial tag helper to render the _ToDo partial view, passing in the model to render

USING PARTIAL VIEWS TO ENCAPSULATE MARKUP

- When you render a **partial view** without providing an **absolute path** or file extension, such as _ToDo, the framework tries to locate the view by searching the Pages folder, **starting** from the Razor Page **that invoked it**.
- For example, if your Razor Page is located at Pages/Agenda/ToDos/RecentToDos.chstml, the framework would look in the following places for a file called _ToDo.chstml:
  - Pages/Agenda/ToDos/ (the current Razor Page's folder)
  - Pages/Agenda/
  - Pages/
  - Pages/Shared/
  - Views/Shared/

IMPORTING COMMON DIRECTIVES WITH _VIEWIMPORTS

- The **_ViewImports.cshtml** file contains directives that are inserted at the top of every Razor view.
  - This can include things like the @using and @model statements.
- The _ViewImports.cshtml file can be placed in any folder, and it will apply to **all views and subfolders in that folder**.

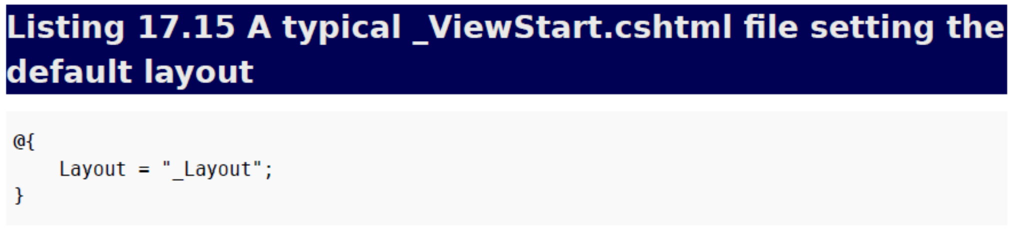


```
@using WebApplication1                                ❶
@using WebApplication1.Pages                          ❶
@using WebApplication1.Models                         ❷
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers  ❸
```

❶ The default namespace of your application and the Pages folder
❷ Adds this directive to avoid placing it in every view
❸ Makes Tag Helpers available in your views, added by default

RUNNING CODE FOR EVERY VIEW WITH _VIEWSTART

- _ViewStart.cshtml file can contain any Razor code, but it's typically used to set the Layout for all the pages in your application.
- Any code in the _ViewStart.cshtml file runs **before** the view executes.
- Note that _ViewStart .cshtml runs only for Razor Page views; it doesn't run for layouts or partial views.



Listing 17.15 A typical _ViewStart.cshtml file setting the default layout

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