Agenda – Exploring Razor Views

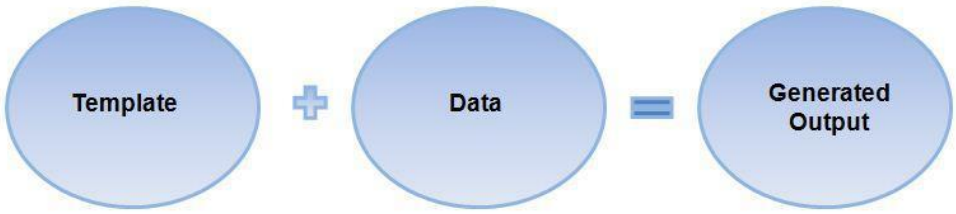
- Introducing Razor View
- Razor Syntax
- Advantages of Razor View
- Types of Views
- Partial Views
- Layout Pages (Master Pages)
- Special Views
- View Categorization
- View Component

Introduction to Razor View

Popular ASP.NET view engines used today include **ASPX, Spark, Nhaml and Razor.** The View is said to be UI part of a web application. In simple, it can be called HTML Template which is used to generate final HTML.

1. Razor View is a file with **.cshtml** extension.
2. These files are not included in the build of MVC application (DLL)
3. Files that cannot be shown by direct requests (master pages, partial views etc) have underscore (_) in the beginning of their names.
4. The Razor View engine allows us to use razor templates to produce HTML.

These are **templates** consisting of HTML and code expressions that place data in to the markup.



Razor view engine:

i. Microsoft introduced the Razor view engine. You can write a mix of html tags and server side code in razor view. Razor uses @ character for server side code instead of traditional <% %>. You can use C# syntax to write server side code inside razor view.

ii. Razor view engine maximize the speed of writing code by minimizing the number of characters and keystrokes required when writing a view. Razor views files have .cshtml extension.

Razor Syntax

Inline expression:

Start with @ symbol to write server side C# code with Html code.

```
<h1>Razor syntax demo</h1>
<h2>@DateTime.Now </h2>
```

Multi-Statement Code block:

You can write multiple line of server side code enclosed in braces @{ ... }. Each line must ends with semicolon same as C#.

```
@{
    var date = DateTime.Now.ToString();
    <h2>Today's date is: @date </h2>
    var message = "Hello World";
    <h3>@message</h3>
}
```

Display text from code block:

Use @: or <text>/<text> to display texts within code block.

```
@{
    <text>This is a Demo</text>
    var date = DateTime.Now.ToString();
    string message = "Hello World!";
    @:Today's date is: @date <br />
    @message
}
```

if-else condition:

Write if-else condition starting with @ symbol. The if-else code block must be enclosed in braces { }, even for single statement.

```
@if (DateTime.IsLeapYear(DateTime.Now.Year))
{
    @DateTime.Now.Year @:is a leap year.
}
else
{
    @DateTime.Now.Year @:is not a leap year.
```

```
    }
```

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

Looping: @for, @foreach, @while, and @do while

Razor is smart enough to implicitly identify a lot of code nugget scenarios. But there are times when you want/need to be more explicit in how you scope the code nugget expression. The @(expression) syntax allows you to do that:

```
@{
      int var = 10;
   }
  @var/2
```

Output: 10/2

```
@{
      int var = 10;
    }
   @(var/2)
```

Output: 5

Using "@" in string:

```
@{
      var domain = "deccansoft";
      @:sandeepsoni@deccansoft.com<br />
      @:sandeepsoni@domain.com<br />
      @:sandeepsoni@(domain).com<br /> <!-- Explicit Expression -->
      @:sandeepsoni@@@(domain).com
```

```
}
```

The following markup is **not** valid Razor:

```
<p>@GenericMethod<int>()</p>
```

The code must be written as an explicit expression:

```
<p>@(GenericMethod<int>())</p>
```

Comments:

Comments in Razor are delimited by @**@. If you are inside a C# code block, you can also use // and /* */ comment delimiters.

```
@*
    A Razor Comment
*@
  @{
    //A C# comment
    /* A Multi
      line C# comment
    */
  }
```

To avoid encoding

```
@{
    string str = "<b>demo</b>";
  }
  @Html.Raw(str) @*To avoid encoding*@
  @str @*encodes to &lt;b&gt;demo&lt;/b&gt;*@
```

Appending/Inserting Dynamic values in Static text:

You can use the explicit expression syntax to append static text at the end of a code nugget without having to worry about it being incorrectly parsed as code:

```
<img src="/Images/@(Model.EmployeeName).jpg" />
```

Above we have embedded a code nugget within an <img> element's src attribute. It allows us to link to images with URLs like "/Images/E1.jpg". Without the explicit parenthesis, Razor would have looked for a ".jpg" property on the EmployeeName (and raised an error). By being explicit we can clearly denote where the code ends and the text begins.

foreach statement:

```
@{
    <h3>Team Members</h3>
    string[] teamMembers = { "Matt", "Joanne","Scott" };
    foreach (var person in teamMembers)
    {
      <p>@person</p>
    }
}
```

Working with Razor Directives

Razor directives are represented by implicit expressions with reserved keywords following the `@` symbol. A directive typically changes the way a view is parsed or enables different functionality.

1. **@using <Namespace>**
2. **@model <Type of Model in the Page/View>**
3. **@inherits** <TypeNameOfClassToInheritFrom (should be inherited from RazorPages<TModel>)>

Step1: Add a New class to Project - CustomRazorPage

```
using Microsoft.AspNetCore.Mvc.Razor;
public abstract class CustomRazorPage<TModel> : RazorPage<TModel>
{
   public string CustomText { get; } = "This is Custom Text";
}
```

Step2: Content in .cshtml file:

```
@inherits CustomRazorPage<TModel>
<div>Custom text: @CustomText</div>
```

4. **@inject** directive enables the Razor Page to inject a service from the service container into a view.

```
@inject IMath m;
```

5. **@section** directive is used in conjunction with the layout to enable views to render content in different parts of the HTML page.
6. **@functions** directive enables a Razor Page to add function-level content to a view:

```
@functions {
1.  public string GetHello()
```

```
  {
      return "Hello";
  }
}
2.  <div>From method: @GetHello()</div>
```

Types of Views

There are following main types of views:

- Action Specific Views
- Partial Views
- Layouts
- Special View

Action Specific Views:

Action Specific View are called from some action method and they generally stored in the view folder related controller and by default have the same name as of action method.

We can call views of different name by specifying view name and from the different folder by specifying full path from an action method.

Example:

In **Controller:**

```
public IActionResult Index()
{
    return View();
}
```

Add **Index.cshtml:**

```
@{
  ViewData["Title"] = "Home Page";
}
<div class="row">
  <p>Your application Home Page</p>
</div>
```

Partial Views

Partial View is rendered within another view.

- A partial view allows us to put HTML and C# code in to a file so that we can reuse it in several different places. A best practice to start partial view with an underscore (_).
- The partial view is same as a normal view but we make its Layout null. Furthermore, special views like _ViewStart are not executed for partial views.
- We can add a partial view into a view with **@Html.Partial("_ViewName")**, while partial view can have relative or full qualified path. And if we have strongly typed partial view, then we may add partial view with **@Html.Partial("_ViewName", dataModel).**
- We can add a partial view in another partial view. Within each view or partial view, relative paths are always relative to that view, not the root or parent view.

Create a partial view in Shared folder name **_RenderTable.cshtml:**

```
<table>
  @for (int i = 1; i <= 10; i++)
  {
    <tr>
      <td>@ViewBag.Number</td>
      <td>*</td>
      <td>@i</td>
      <td>=</td>
      <td>@(ViewBag.Number * i)</td>
    </tr>
  }
</table>
```

In **Index.cshtml** add the following:

```
@{
    ViewBag.Number = 2;
  }
  @await Html.PartialAsync("_RenderTable")
OR
  @{
    await Html.RenderPartialAsync("_RenderTable"); //Can be used in code block
  }
```

Note:

If placed in Views folder of controller it can be used only by views of that controller. If Partial View is placed in shared folder, it can be shared by all the views of different controller.

Layout Pages (Master Pages)

Layout Views are said to be main structure and they are very similar to Master Page in ASP.NET Web Forms.

- Layout View defines the main structure of a web application and may contain header, menu, footer and page content area as per requirements.
- They facilitate us to define and handle all these at one place. It is common to have at least one layout view, but we can have more than one layout views to meet different requirements.
- Layout view contains special tag **@RenderBody()** in which main called view is added and **@RenderSection()** which specifies a section to be rendered in the view.

Layout View (Shared Folder):

- Go to \Views\Shared_Layout.cshtml" file.
- As _Layout.cshtml is file in Shared folder, this is shared by all the views.

```
@inject Microsoft.ApplicationInsights.AspNetCore.JavaScriptSnippet JavaScriptSnippet
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>@ViewData["Title"] - DemoCoreMVCApp</title>
  - - - - -
  - - - - -
  @Html.Raw(JavaScriptSnippet.FullScript)
</head>
<body>
  <div class="container body-content">
    @RenderBody()
 - - - - -
 - - - - -
  </div>
 @RenderSection("Scripts", required: false)
</body>
</html>
```

- We are calling the **@RenderBody()** method within the layout file above to indicate where we want the views based on this layout to "fill in" their core content at that location in the HTML.

- We are outputting the "**ViewData["Title"]"** property within the <title> element of our <head> section
- We did not need to wrap our main body content within a tag or element – by default Razor will automatically treat the content of .cshtml as the "body" section of the layout page. We *can* optionally define "named sections" if our layout has multiple replaceable regions.
- We are programmatically setting the "**ViewData["Title"]"** in our .cshtml file. The code within our .cshtml file will run ***before*** the _Layout.cshtml code runs – and so we can write view code that programmatically sets values we want to pass to our layout to render. This is particularly useful for things like setting the page's title, as well as <meta> elements within the <head> for search engine optimization (SEO).
- At the moment, we are programmatically setting the Layout template to use within our EmployeeDetails.cshtml page. We can do this by setting the Layout property on the View.

Special Views

ASP.NET has couple of other special views:

1. **_ViewImports.cshtml:**_ViewImports view is used to perform activities like importing namespaces or performing dependency injection, shared directive. Generally, we have _ViewImports at the root Views folder, but we can have as many as required.
2. **_ViewStart.cshtml:** _ViewStart.cshtml is used to execute common tasks like setting Layout View. The statements listed in _ViewStart.cshtml are run before every view except layouts and partial views. This is fine for cases where we have some view-specific logic where the layout file will vary depending on the specific view. But setting it this way can end up being redundant and duplicative for most web applications – where either all of the views use the same layout, the logic for which layout to pick is common across all of the views.

Open **\Views_ViewStart.cshtml** underneath the folder of your project:

It can be used to define **common view code** that you want to execute at the start of each View's rendering.

For example, we could write code to programmatically set the Layout property for each View to be the _Layout.cshtml file by default:

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
```

How can we have an independent view without _Layout.cshtml as Master

In view do the following

```
@{
    Layout = null;
}
```

View Discovery: View Discovery in the process to identify the viewed called by an action method or a partial view during rendering of final Html. If a view name is not specified or name without path is specified then this process determines which view file will be used based on predefined steps.

View Categorization based on Model

We can categorize views on bases of Model or data manipulation as following:

- Loosely Typed or Type Less Views
- Strongly Typed Views
- Dynamic Views

1. Loosely Typed or Type Less View:

We can pass data to views using loosely typed data collections: ViewData and ViewBag.

2. Strongly Typed Views:

Strongly Typed View has a specific a model type in the view, and it is mapped to an instance of this type passed to the view from the action as a parameter. We can specify a model for a view using the **@model** directive then the instance sent to the view can be accessed in a strongly-typed manner using @Model as an object of specified type. It is highly practiced and is most recommended option to pass data as it gives the lot of benefits like robustness, compilation protection over other approaches.

Example:

a. Add **Employee** class in Models Folder of the application.

```
public class Employee
{
    public int EmpId { get; set; }
    public string EmpName { get; set; }
    public double Salary { get; set; }
    public string DeptName { get; set; }
}
```

b. In Controller, modify the Index method as below, see that Employee Model object is created and passed to View method as argument.

```
public IActionResult ViewEmployees()
{
    List<Employee> lst = new List<Employee>()
```

```
    {
            new Employee() { EmpId = 1, EmpName = "E1", Salary = 1000, DeptName = "D1" },
            new Employee() { EmpId = 2, EmpName = "E2", Salary = 1000, DeptName = "D2" },
            new Employee() { EmpId = 1, EmpName = "E3", Salary = 1000,DeptName="D3"}
    };
    return View(lst);
}
```

c. Create a View for the Index and add the following to the View

Strongly Typed Model View example:

```
@model IEnumerable<DemoMVCCoreApp.Models.Employee>
@{
  ViewData["Title"] = "Home Page";
}
@*Loosely typed data using ViewBag*@
@ViewBag.Message
<div class="row">
  <table border="1">
    <tr>
      <th>Employee Id</th>
      <th>Employee Name</th>
      <th>Salary</th>
      <th>Department Name</th>
    </tr>
    @{
      foreach (var emp in Model)
      {
        <tr>
          @*Strongly typed data*@
          <td>@emp.EmpId</td>
          <td>@emp.EmpName</td>
          <td>@emp.Salary</td>
          <td>@emp.DeptName</td>
        </tr>
      }
```

```
    }
  </table>
</div>
```

3. Dynamic Views:

Dynamic Views are hybrid of both Strongly Typed and Loosely-Typed Views. In this kind of view, a model is not declared but have a model instance passed to them. But @model and its properties can be used in view dynamically. In this case, we don't have compilation protection or IntelliSense. If the property doesn't exist, then an application crashes at runtime.

Dynamic View example:

Remove this line "@model IEnumerable<DemoMVCCoreApp.Models.Employee>" from the above example and see output it will display the data same as in the above example without "@model", now this view is Dynamic View.

View Component

ViewComponents = HtmlHelper + Child Actions + Separation of Concern + Asynchronous + Dependency Injection

- View components are similar to partial views, but they're much more powerful. View components don't use model binding, and only depend on the data provided when calling into it.
- Unlike Partial View, View components does not depend on controllers. It has its own class to implement the logic to develop the component's model and razor markup view page. You may need to write a some **business logic** where for example you might need to access a 3rd party web service and get the data and do something with it and then display this information.
- View Components supports SOC (Separation-Of-Concerns).
- View Components involves processing by both a view component class and a View.
- View Components can be implemented in any part of the web application where there are some possibilities of code duplicate like Navigation Pane, Login Panel, Menu, Shopping Cart etc. So, in simple word, View Components is actually behaving like a web part which contains both business logic and UI design to create a web part package which can be reused in the multiple parts of the web application.
- By using View Components, you will not have to wait for all of the content of an entire page to load like you might typically do now. This "chunking" approach will allow you to quickly send out various partial sections of your page as they become available. Basically, it reduce the overall "**time to first byte**", which can be commonly used to judge the speed of a site.

View components are intended anywhere you have reusable rendering logic that's too complex for a partial view, such as:

1. Dynamic navigation menus

2. Tag cloud (where it queries the database)
3. Login panel
4. Shopping cart
5. Sidebar content on a typical blog
6. 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

1. The View Component class must be derived from the ViewComponent class.

we have two processes to declare a view component. One is we can create a class with a name and then append the ViewComponent or we can decorate our class with the ViewComponent attribute, by setting our view components name through the attribute's name property.

```
public class EmployeeViewComponent: ViewComponent
{ }
OR
[ViewComponent(Name = "Employee")]
public class EmployeeInfo: ViewComponent
{}
```

2. In view component class, we must defined one method and it –

a) Must have the name **InvokeAsync**
b) Must return a **Task** object types to **IViewComponentResult**
c) Must be decorated with the **async** Keyword
d) Can accept **parameters** (of any type)

3. To invoke a view component from any other view:

@await **Component.InvokeAsync**("EmployeeViewComponent",<anonomous type containing params>)

Example:

1. Create a ***ViewComponents*** folder and add the following class:

```
public class Menu
{
   public string Text;
   public string URL;
   public bool isSelected;
```

```
    public bool isBold;
}
public class MenuViewComponent : ViewComponent
{
    //private readonly MenuContext db;
    public MenuViewComponent()//MenuContext context)
    {
        //db = context;
    }
    public async Task<IViewComponentResult> InvokeAsync(int selectedMenuIndex, bool isBold)
    {
        List<Menu> menus = new List<Menu>() {
            new Menu() { Text = "Home",URL = "/",isSelected = selectedMenuIndex==0, isBold = isBold},
            new Menu() { Text = "Services",URL = "/services",isSelected = selectedMenuIndex==1, isBold = isBold},
            new Menu() { Text = "AboutUs",URL = "/home/aboutus",isSelected = selectedMenuIndex==2, isBold =
isBold},
            new Menu() { Text = "ContactUs",URL = "/contactus",isSelected = selectedMenuIndex==3, isBold = isBold},
        };
        //var menus = await db.Menus.ToListAsync();
        return View(menus);
    }
}
```

2. Create a ***Views/Shared/Components/Menu/Default.cshtml*** Razor view

```
@*@model IEnumerable<ViewComponentSample.Models.TodoItem>*@
```

```
@model System.Collections.Generic.List<Menu>
<h3>Menu Items</h3>
<ul>
    @foreach (var menu in Model)
    {
        <li> <a href="@menu.URL">@menu.Text</a> </li>
    }
</ul>
```

3. Edit _Layout.cshtml: Enter the below between <header> and <footer> sections

```
<div class="container">
  <div class="row">
    <div class="col-md-2">
      @await Component.InvokeAsync("Menu", new { selectedMenuIndex = 0, isBold = false })
    </div>
    <div>
      <div class="container">
        <partial name="_CookieConsentPartial" />
        <main role="main" class="pb-3">
          @RenderBody()
        </main>
      </div>
    </div>
  </div>
</div>
```