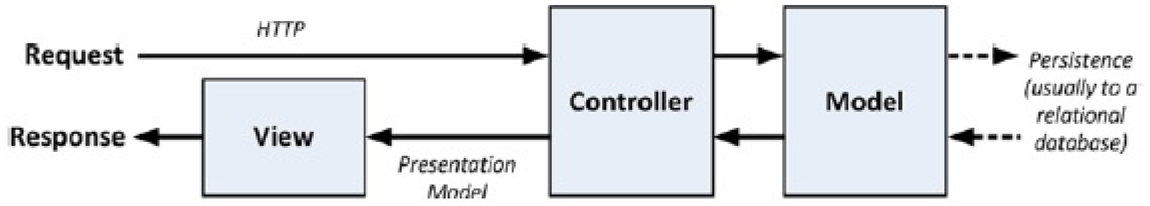
ASP.NET Core Views

MVC as separation of concerns



Software should be separated based on the **kinds of work** it performs

- **Controller** – application logic. Communicates with user. It receives and handles user queries, interrupts with Model, and returns results by View objects
- **Model** – contains classes that represent data, performs operations with data-bases and organizes relations between data-classes.
- **View** – performs UI representation. Works with model of presentation.

Views

- View handles the app's **data presentation** and **user interaction**
- Views allow the **presentation** of data to be separated from the **logic** that processes requests
- View is a file that contains **HTML** elements and **C#** code, which is processed to generate a response
- Using **Razor markup** – view engine – makes it easy to mix HTML and C# content used to display content to the user
- Razor markup **is code** that interacts with HTML markup **to produce** a webpage that's sent to the client

Main view components

IViewEngine
IView
ViewEngineResult

IViewEngine

```
public interface IViewEngine
{
   ViewEngineResult GetView(string executingFilePath, string viewPath, bool isMainPage);

   ViewEngineResult FindView(ActionContext context, string viewName, bool isMainPage);
}
```

- View engines -- classes that implement `IViewEngine`
- Role of a view engine – to translate requests for views into **ViewEngineResult**, that can be used to generate responses
- If **GetView** cannot provide view, then **FindView** is called so that the view engine has a chance to search for the view using **`ActionContext`**, which provides information about the action method that created `ViewResult`

```
InvalidOperationException: The view 'MyView' was not found. The following locations were searched:
/Views/Home/MyView.cshtml
/Views/Shared/MyView.cshtml
```

ViewEngineResult

```
public class ViewEngineResult
{
    public IEnumerable<string> SearchedLocations { get; }
    public IView View { get; }
    public string ViewName { get; }
    public bool Success { get; }

    public static ViewEngineResult Found(string viewName, IView view);

    public static ViewEngineResult NotFound(string viewName,
                                            IEnumerable<string> searchedLocations);

    public ViewEngineResult EnsureSuccessful  (IEnumerable<string> originalLocations);
}
```

- ViewEngineResult allows a **view engine** to respond to the MVC when a view is requested
- **Found** (name, view) -- Calling this method provides MVC with the requested **view**, which is set using the view parameter
- **NotFound(name**, **locations**) -- creates ViewEngineResult , that tells MVC that the requested view **could not be found**; **locations** is an enumeration of string values that describe where the view engine has looked for the view.

IView

```
public interface IView
{
    string Path { get; }
    Task RenderAsync(ViewContext context);
}
```

- **IView** implementation is passed to the constructor of a **ViewEngineResult**, which is then returned from the view engine methods
- **Path** – returns path of the view, which assumes that views are defined as files
- **RenderAsync** – is called by MVC to generate the response to the client;
- **ViewContext** provides information about the request from the client and the output from the action method, and defines properties that give access to information about the request and details of how the MVC has processed it:
  - **ViewData** – returns a **ViewDataDictionary** that contains the view data provided by the controller
  - **TempData** – returns a dictionary containing the temp data
  - **Writer** – returns a TextWriter that should be used to write the output from view.

`ViewContext` Properties

- `Controller` returns `IController` implementation that processed the current request
- `RequestContext` returns details of the current request
- `RouteData` returns the routing data for the current request
- `TempData` returns the temp data associated with the request
- `View` returns the implementation of the `IView` interface that will process the request
- `ViewBag` returns an `object` that represents the view bag
- `ViewData` returns a dictionary `ViewDataDictionary` with
  - **Model** -- model data provided by the controller
  - **ModelMetadata** -- meta data for the model
  - **ModelState** -- state of the model
  - **Keys** -- an enumeration of key values that can be used to access **ViewBag** data

Views_f1

- DebugDataView

Razor Syntax

Character @ precedes code instructions in the following contexts:

- for a single code line/values

```
<p>
    Current time is: @DateTime.Now
</p>
```

- **@{ … }** for code blocks with multiple lines

```
@{
     var name = “John”;
     var nameMessage = "Hello, my name is " + name + " Smith";
}
```

- **@:** For single plain text to be rendered in the page

```
@{
    @:The day is: @DateTime.Now.DayOfWeek. It is a
        <b>great</b> day!
}
```

Razor Syntax

- HTML markup lines can be included at any part of the code

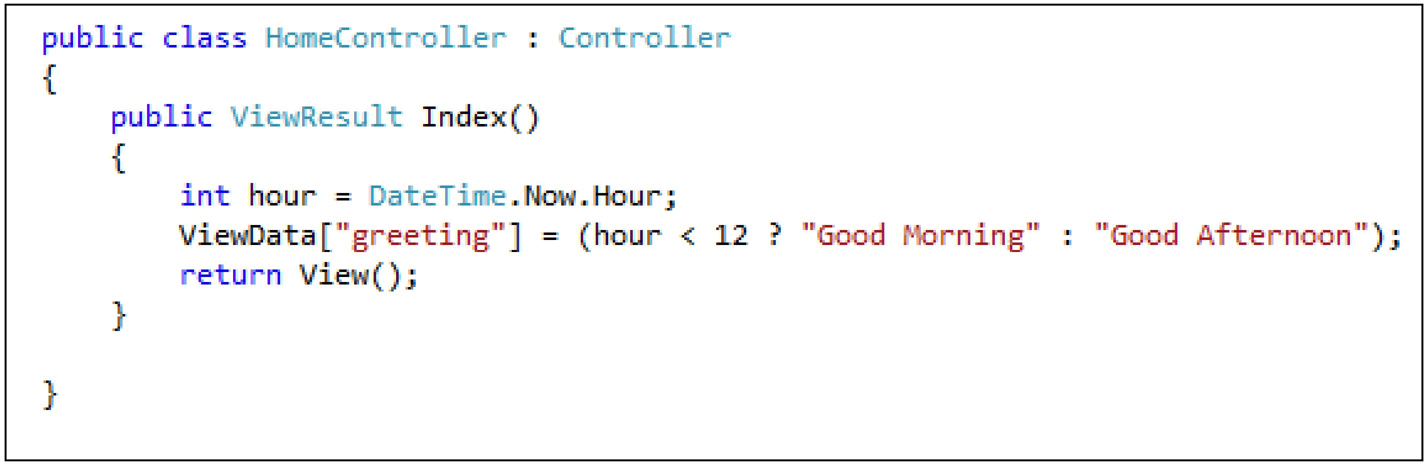
```
@if (IsPost)
{
    <p>
        Hello, the time is @DateTime.Now and this page is a postback!
    </p>
}
else
{
    <p>Hello, today is: </p> @DateTime.Now
}
```

- Razor converts CSHTML files into C# classes, compiles them, and then creates new instances **each time** a view is required to generate a result

Passing data to the View

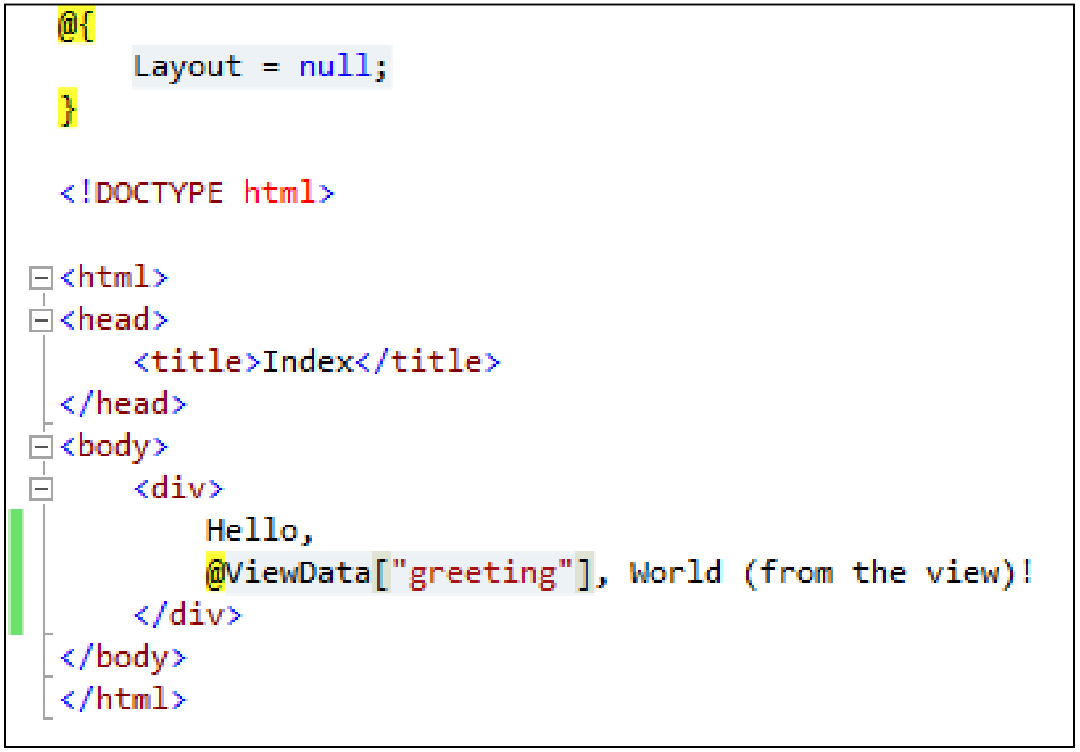
- by using the ViewDataDictionary
- by using the ViewBag
- by using strongly typed views

ViewDataDictionary



```
public class HomeController : Controller
{
    public ViewResult Index()
    {
        int hour = DateTime.Now.Hour;
        ViewData["greeting"] = (hour < 12 ? "Good Morning" : "Good Afternoon");
        return View();
    }

}
```



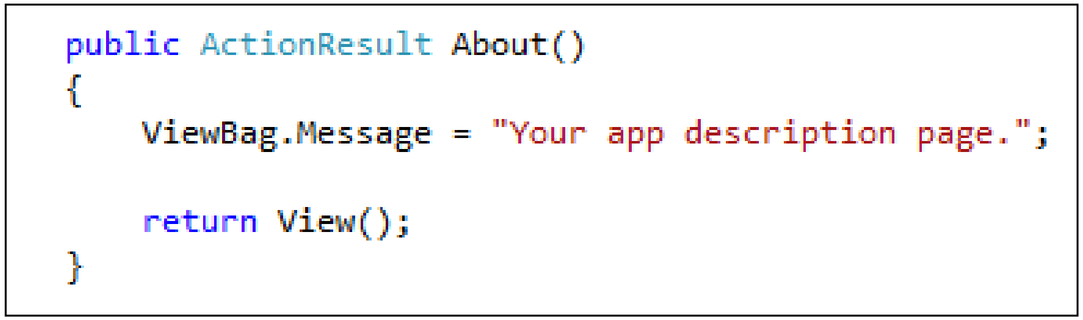
```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <title>Index</title>
</head>
<body>
    <div>
        Hello,
        @ViewData["greeting"], World (from the view)!
    </div>
</body>
</html>
```

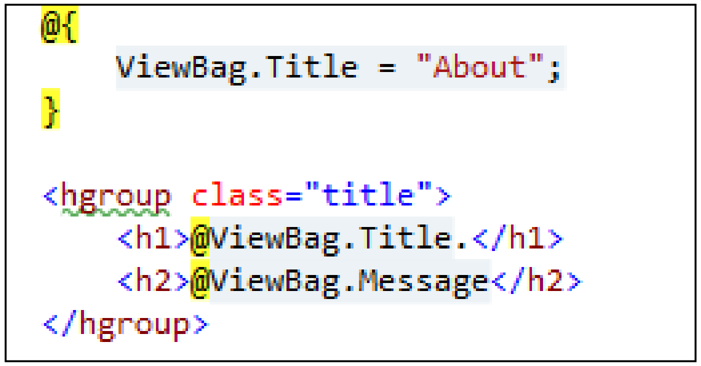
ViewBag

- **ViewBag** provides a way to pass data from the controller to the view
- Set properties on the dynamic **ViewBag** property within controller
- **ViewBag** property is also available in the view



```
public ActionResult About()
{
    ViewBag.Message = "Your app description page.";

    return View();
}
```



```
@{
    ViewBag.Title = "About";
}

<hgroup class="title">
    <h1>@ViewBag.Title.</h1>
    <h2>@ViewBag.Message</h2>
</hgroup>
```

Strongly Typed Views

If the **@model** directive has been used to specify a model type then `View` classes inherit from `RazorPage` or `RazorPage<` TModel `>`

RazorPage<TModel>

- View classes inherit from **RazorPage<** TModel **>** with **@model** directive
- Properties of **RazorPage<** TModel **>**
  - **Model** returns the model data provided by the action method
  - **ViewData** returns a **ViewDataDictionary** that provides access to other view data features
  - **ViewContext** returns a **ViewContext** object
  - **Layout** is used to specify a layout
  - **ViewBag** provides access to the view bag object
  - **TempData** provides access to the temp data
  - **Context** returns an **HttpContext,** that describes the current request and the response that is being prepared
  - **User** returns the profile of the user associated with this request
  - **RenderSection**() is used to insert a section of content from view into a layout
  - **RenderBody**() inserts all the content in a view that is not contained in a section into a layout
  - **IsSectionDefined**() is used to determine whether a view defines a section

Generating response content

```
public ViewResult Index() =>
            View(new string[] { "Apple", "Orange", "Pear" });
```

This is a list of fruit names: **Apple Orange Pear**

Index.cshtml

```
@model string[]
@{ Layout = null; }
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Razor</title>
    <link asp-href-include="lib/bootstrap/dist/css/*.min.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
    This is a list of <b>fruit</b> names:
@foreach (string name in Model)
    {
        <span><b> @name</b></span>
    }
</body> </html>
```

Class generated by Razor

```
public class ASPV_Views_Home_Index_cshtml : RazorPage<string[]> {. . .}
```

View Rendering

```
public override async Task ExecuteAsync()
{
    Layout = null;
    WriteLiteral(
        @"<!DOCTYPE html><html><head>
         <meta name=""viewport"" content=""width=device-width"" />
         <title>Razor</title>
         <link asp-href-include=""lib/bootstrap/dist/css/*.min.css""
         rel=""stylesheet"" />
         </head><body class=""m-1 p-1"">This is a list of fruit names:“);

    foreach (string name in Model)
    {
        WriteLiteral("<span><b>");
        Write(name);
        WriteLiteral("</b></span>");
    }
    WriteLiteral("</body></html>");
}
```

Adding dynamic content to a Razor View

- **Inline code** -- for small, self-contained pieces **of view logic**, such as **if** and **foreach** statements; this is the **fundamental tool** for creating dynamic content in views
- **Tag helpers** – for generating attributes on HTML elements
- **Sections** -- for creating sections of content that will be inserted into **layout** at specific locations
- **Partial views** -- for sharing subsections of view markup between views
  - partial views can contain inline code, HTML helper methods, and references to other partial views
  - Partial views **do not invoke** an action method, so they cannot be used to perform business logic
- **View components** -- for creating reusable UI controls or widgets that need to contain business logic

Dynamic view with Layout sections

```
@model  string[]
@{ Layout = "_Layout"; }

@section Header{
       <div class="bg-success">
              @foreach (string str in new[] { "Home", "List", "Edit"  })
               {
                      <a class="btn btn-sm btn-primary" asp-action="str"> @str</a>
               }
   </div>
}

This is a list of fruit names:
@ foreach (string name in Model)
{
   <span><b> @ name</b></span>
}

@section Footer {
   <div class="bg-success">
      This is the footer
   </div>
}
```

Index.cshtml

Layout page

- Layouts are a specialized forms of view
- Call to **@RenderBody, @RenderSection("Header"), @RenderSection("Footer")** inserts the contents of the view specified by the action method into the layout markup
- Using Layout property to specify a layout inside a view
- Layout has access to the same properties the Razor view has, including:
  - **AjaxHelper** (through **Ajax** property)
  - **HtmlHelper** (through **Html** property)
  - **ViewData** and **model**
  - **UrlHelper** (through **Url** property)
  - **TempData** and **ViewContext**

Using Layout sections

- Sections – providing regions of content within a layout
- When Razor parses layout, **RenderSection()** helper method is **replaced** with contents of section in view with the specified name.
- Parts of the view that are **not contained with a section** are inserted into layout using the **RenderBody()** helper
- Sections give **control** over which parts of the view are inserted into the layout and where they are placed. Optional sections. Testing for sections:

```
...
@if (IsSectionDefined("Footer"))
{
    @RenderSection("Footer")
}
else
{
    <h4>This is the default footer</h4>
}
```

- In layout pages, renders the content of a named section and specifies whether the section is required:

```
@RenderSection("Footer", false)
```

Partial Views

- Partial views -- separate view files that contain fragments of tags and markup that **can be included** in other **views**
- **Create as partial view (**check option) on **Add View**

MyPartial.cshtml

```
<div>
    This is the message from the partial view.
    @Html.ActionLink("This is a link to the Vegetables", "Index1")
</div>
```

Index.cshtml

```
  .  .  .
@Html.Partial("MyPartial")
  .  .  .
```

Strongly typed partial views

MyStronglyTypedPartial.cshtml

```
@model IEnumerable<string>
<div>
    This is the message from the partial view
    <ul>
        @foreach (string str in Model)
        {
            <li>@str</li>
        }
    </ul>
</div>
```

List.cshtml

```
 . . .
 @Html.Partial("MyStronglyTypedPartial",
              new[] {"Apple", "Orange", "Pear"})
 . . .
```

- **Partial** helper method with additional argument which defines view model
- **http://localhost:53742/home/list**

ViewComponent

- **ViewComponents** are classes that provide action-style logic to support partial views:
  - complex content can be embedded in **views**
  - allowing the C# code that supports it
  - unit testing
- View components:
  - are typically derived from **ViewComponent** class
  - are applied in a **parent view** using **@await Component.InvokeAsync**
- A view component provides a partial view with the data that it needs:
  - independently from the parent view and the action that renders it
  - can be thought of as a **specialized action**, but one that is used only to provide a partial view with data
  - it cannot receive HTTP requests
  - content that it provides will always be included in the parent view