MVC – Views

.NET CORE

In the Model-View-Controller (MVC) pattern, the view handles the app's data presentation and user interaction. A view is an HTML template with embedded Razor markup.

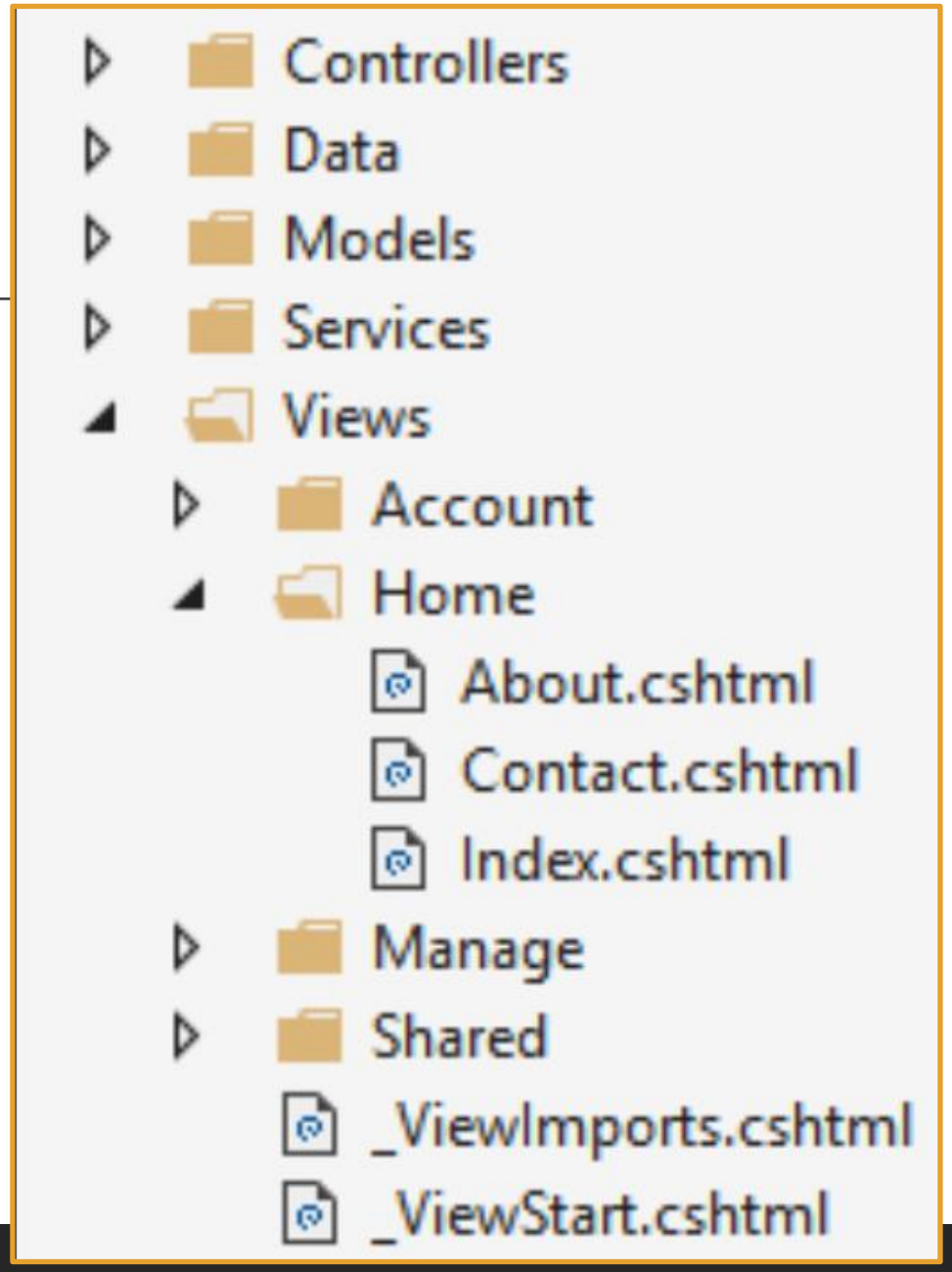
https://docs.microsoft.com/en-us/aspnet/core/mvc/views/overview?view=aspnetcore-3.1

Views – Overview

https://docs.microsoft.com/en-us/aspnet/core/mvc/views/overview?view=aspnetcore-3.1

In ASP.NET Core MVC, ***views*** are .cshtml files that use the C# programming language in ***Razor*** markup. Usually, ***view*** files are grouped into folders named for each of the app's ***controllers***. The folders are stored in a Views folder at the root of the app

The *Home* ***controller*** is represented by a *Home* folder inside the *Views* folder. The *Home* folder contains the ***views*** for the *About*, *Contact*, and *Index* (homepage) webpages. When a user requests one of these three webpages, ***controller actions*** in the *Home* ***controller*** determine which of the three ***views*** is used to build and return a webpage to the user.



Views – Benefits

https://docs.microsoft.com/en-us/aspnet/core/mvc/views/overview?view=aspnetcore-3.1#benefits-of-using-views

Views separate the user interface markup from other parts of the app. (***Separation of Concerns)***

The app is easier to maintain because it's better organized.

The parts of the app are ***loosely coupled***. Build and update the app's ***views*** separate from the business logic and data access components.

Views – How Controllers Specify Views

https://docs.microsoft.com/en-us/aspnet/core/mvc/views/overview?view=aspnetcore-3.1#how-controllers-specify-views

---

Views are typically returned from actions as a ***ViewResult***, which is a type of ***ActionResult***. Your ***action method*** can create and return a ***ViewResult*** (not common). Since most controllers inherit from ***Controller***, you simply use the View helper method to return the ***ViewResult***:

```
public IActionResult About()
{
    ViewData["Message"] = "Your application description page.";

    return View();
}
```

Views – Return Options

https://docs.microsoft.com/en-us/aspnet/core/mvc/views/overview?view=aspnetcore-3.1#how-controllers-specify-views

The ***View*** helper method has several overloads.

- An explicit view to return:

```
C#

return View("Orders");
```

- A model to pass to the view:

```
C#

return View(Orders);
```

- Both a view and a model:

```
C#

return View("Orders", Orders);
```

Dynamic Views

https://docs.microsoft.com/en-us/aspnet/core/mvc/views/overview?view=aspnetcore-3.1#dynamic-views

---

Views that don't declare a **model** type using ***@model*** but that have a ***model*** instance passed to them (for example, return ***View(Address)***;) can reference the instance's properties dynamically. This feature offers flexibility but doesn't offer compilation protection or IntelliSense. If the property doesn't exist, webpage generation fails at runtime.

```
<address>
    @Model.Street<br>
    @Model.City, @Model.State @Model.PostalCode<br>
    <abbr title="Phone">P:</abbr> 425.555.0100
</address>
```

Partial Views

https://docs.microsoft.com/en-us/aspnet/core/mvc/views/partial?view=aspnetcore-3.1

A ***partial view*** is a ***Razor*** markup file (.cshtml) that renders ***HTML*** output within another markup file's rendered output. ***Partial view*** file names often begin with an underscore (_).

Partial views are an effective way to break up large markup files into smaller components and reduce the duplication of common markup content across markup files.

Partial views shouldn't be used to maintain common layout elements (use _Layout.cshtml) or where complex rendering logic or code execution is required to render the markup.

The ***Partial Tag Helper*** renders content asynchronously and uses an HTML-like syntax:

```
<partial name="_PartialName" />
```

```
<partial name="_PartialName.cshtml" />
```

Views – Passing Data

https://docs.microsoft.com/en-us/aspnet/core/mvc/views/overview?view=aspnetcore-3.1#passing-data-to-views

You can pass Strongly-Typed data and Weakly-Typed data to views.

Strongly typed - Viewmodel

Specify a ***model*** (aka, viewmodel) type in the view and pass it from the action method.
This allows the view to have ***strong type checking(***and ***Intellisense!)***. Strong typing (or strongly typed) means every variable and constant has an explicitly defined type (string, int, or DateTime). The validity of types used in a view is checked at compile time.

```
public IActionResult Contact()
{
    ViewData["Message"] = "Your contact page.";

    var viewModel = new Address()
    {
        Name = "Microsoft",
        Street = "One Microsoft Way",
        City = "Redmond",
        State = "WA",
        PostalCode = "98052-6399"
    };

    return View(viewModel);
}
```

Views – Passing Data

https://docs.microsoft.com/en-us/aspnet/core/mvc/views/overview?view=aspnetcore-3.1#weakly-typed-data-viewdata-viewdata-attribute-and-viewbag

Weakly typed ViewData and ViewBag

Weak types (or loose types) means that you don't explicitly declare the type of data you're using. You can use the collection of *weakly typed* data for passing small amounts of data in and out of *controllers* and *views*.

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

    return View();
}
```

Views – Passing Data

https://docs.microsoft.com/en-us/aspnet/core/mvc/views/overview?view=aspnetcore-3.1#weakly-typed-data-viewdata-viewdata-attribute-and-viewbag

- ***ViewData*** is a Dictionary.
- ***ViewBag*** is a wrapper around ***ViewData*** that provides dynamic properties for the underlying ***ViewData*** collection.
- Key lookups are case-insensitive for both ***ViewData*** and ***ViewBag***.

ViewData and ***ViewBag*** don't offer compile-time type checking and are more error-prone than using a ***viewmodel***. Some developers prefer to minimally or never use ***ViewData*** and ***ViewBag***.

Weakly typed – ViewData, ViewDataAttribute, and ViewBag

Weakly typed means that you don't explicitly declare the type of data you're using. You can use the collection of weakly typed data for passing small amounts of data in and out of controllers and views.

Passing data between a ...	Example
Controller and a view	Populating a dropdown list with data.
View and a layout view	Setting the **<title>** element content in the layout view from a view file.
Partial view and a view	A widget that displays data based on the webpage that the user requested.

ViewData and ViewBag Differences

ViewData()	ViewBag()
• Derives from ***ViewDataDictionary***, so it has dictionary properties that can be useful, such as ***ContainsKey***, ***Add***, ***Remove***, and ***Clear***. • Keys in the dictionary are strings, so whitespace is allowed. Example: ***ViewData["Some Key With Whitespace"]*** • Any type other than a string must be ***cast*** in the view to use ***ViewData***.	• Derives from ***DynamicViewData***, so it allows the creation of dynamic properties using dot notation (@ViewBag.SomeKey = <value or object>), and no ***casting*** is required. The syntax of ***ViewBag*** makes it quicker to add to ***controllers*** and ***views***. • Simpler to check for null values. Example: @ViewBag.Person?.Name • ViewBag isn't available in the Razor Pages.

Views – TempData

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/app-state?view=aspnetcore-3.1#tempdata

The ***TempData*** property stores data until it's read in another request. It is a ***Dictionary*** of string to object. Data is removed after the request that reads it.
The ***Keep(String)*** and ***Peek(string)*** methods can be used to examine the data without deletion at the end of the request. ***Keep*** marks all items in the dictionary for retention.

TempData is useful for 1) redirection when data is required for more than a single request and 2) when implemented by ***TempData*** providers using either ***cookies*** or ***session state***.
TempData is:

- Useful for redirection when data is required for more than a single request.
- Implemented by TempData providers using either cookies or session state.

```
@page
@model IndexModel

<h1>Peek Contacts</h1>

@{
    if (TempData.Peek("Message") != null)
    {
        <h3>Message: @TempData.Peek("Message")</h3>
    }
}
```

TempData Example

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/app-state?view=aspnetcore-3.1#tempdata-samples

In this markup, at the end of the request, ***TempData["Message"]*** is not deleted because ***Peek()*** is used. Refreshing the page displays the contents of ***TempData["Message"]***.

The cookie-based TempData provider is enabled and used by default to store TempData in cookies. You can choose another provider and configure it in your ConfigureServices() method.

```
@page
@model IndexModel

<h1>Peek Contacts</h1>

@{
    if (TempData.Peek("Message") != null)
    {
        <h3>Message: @TempData.Peek("Message")</h3>
    }
}
```

```
public class CreateModel : PageModel
{
    private readonly RazorPagesContactsContext _context;

    public CreateModel(RazorPagesContactsContext context)
    {
        _context = context;
    }

    public IActionResult OnGet()
    {
        return Page();
    }

    [TempData]
    public string Message { get; set; }

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

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _context.Customer.Add(Customer);
        await _context.SaveChangesAsync();
        Message = $"Customer {Customer.Name} added";

        return RedirectToPage("./IndexPeek");
    }
}
```

Razor Syntax

https://docs.microsoft.com/en-us/aspnet/core/mvc/views/razor
https://docs.microsoft.com/en-us/aspnet/core/mvc/views/razor?view=aspnetcore-3.1

Razor is a markup syntax for embedding server-based code into webpages. ***Razor*** markup is code that interacts with HTML markup to produce a webpage that's sent to the client. The ***Razor*** syntax files end in .cshtml and consist of Razor markup, C#, and HTML.

Razor markup starts with the @ symbol. You can run any C# statement control flow syntax in a Razor markup file by placing C# code within Razor code blocks (marked with curly braces ({ })). You can use C# comment syntax.

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

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

Razor @model directive

https://docs.microsoft.com/en-us/aspnet/core/mvc/views/razor?view=aspnetcore-3.1#model
https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-3.1&tabs=visual-studio#strongly-typed-models-and-the--keyword

---

MVC provides the ability to pass ***strongly typed model*** objects to a ***view***. This ***strongly typed*** approach enables compile time code checking. The @model directive specifies the type of the model passed to a view.

```
// GET: Movies/Details/5
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie
        .FirstOrDefaultAsync(m => m.Id == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}
```

```
@model MvcMovie.Models.Movie

@{
    ViewData["Title"] = "Details";
}

<h1>Details</h1>

<div>
    <h4>Movie</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Title)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Title)
```

Razor @model directive

https://docs.microsoft.com/en-us/aspnet/core/mvc/views/razor?view=aspnetcore-3.1#model
https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-3.1&tabs=visual-studio#strongly-typed-models-and-the--keyword

---

The ***@model*** directive allows you to access the list of movies that the ***controller*** passed to the view by using a ***Model*** object that's ***strongly typed***.

In the Index.cshtml ***view***, the code loops through the movies with a foreach statement over the ***strongly typed Model*** object.

```
// GET: Movies
public async Task<IActionResult> Index()
{
    return View(await _context.Movie.ToListAsync());
}
```

```
@model IEnumerable<MvcMovie.Models.Movie>

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Genre)
            </th>
            <th>
```

ASP.NET Core MVC Tutorial

Complete the **ASP.NET Core MVC** tutorial here