In the context of hosting ASP.NET Core applications with Kestrel, there is a specific sequence and hierarchy that determines how requests are processed and responded to. This involves pipelines, Kestrel, middlewares, and other components. Here's a breakdown of their relationship and sequence:

1. \*\*Kestrel\*\*:

- Kestrel is the web server that listens for incoming HTTP requests.

- It's responsible for accepting incoming connections and managing the low-level details of network communication.

- Kestrel acts as the entry point for all HTTP requests to your ASP.NET Core application.

2. \*\*Request Pipeline\*\*:

- The request pipeline is a series of processing steps that an incoming HTTP request goes through before generating a response.

- It's defined in your ASP.NET Core application's `Startup.cs` file.

- The request pipeline is responsible for handling various tasks, such as authentication, routing, and middleware execution.

3. \*\*Middlewares\*\*:

- Middlewares are components that are added to the request pipeline.

- Each middleware performs a specific task or set of tasks during request processing.

- Middlewares can modify the request, generate responses, or perform various operations like logging or authentication.

- Middleware components are executed in the order in which they are added to the pipeline, from the first to the last.

Here's the sequence of events as an HTTP request passes through this hierarchy:

1. An incoming HTTP request is received by Kestrel.

2. Kestrel hands off the request to the ASP.NET Core application's request pipeline.

3. The request enters the pipeline and starts traveling through the registered middleware components in the order they were added.

4. Each middleware component in the pipeline has the opportunity to inspect or modify the request or perform other operations as needed.

5. The request continues through the pipeline until it reaches the endpoint or handler responsible for generating a response. This could be a controller action in MVC or a Razor Page handler, for example.

6. After the response has been generated, it travels back up through the pipeline in the reverse order of middleware execution.

7. Each middleware component can inspect or modify the response as it passes through.

8. Finally, the response is sent back to Kestrel, which in turn sends it back to the client that made the initial request.

In summary, Kestrel is the web server that receives incoming requests, and the request pipeline, composed of middlewares, processes those requests. The sequence in which middlewares are added to the pipeline determines the order of execution, allowing each middleware to contribute to the request processing and response generation. This architecture provides a flexible and extensible way to handle various aspects of request handling in ASP.NET Core applications.

app.MapControllerRoute(...):

csharp code:

app.MapControllerRoute(

name: "default",

pattern: "{controller=Home}/{action=Index}/{id?}");

This code configures a default route for your ASP.NET Core MVC application. Let's break it down:

name: This is the name of the route. In this case, it's named "default," but you can name it whatever you like. Route names are useful when generating URLs within your application.

pattern: This is the URL pattern that the route should match. It's defined using placeholders enclosed in curly braces. Here's what each part means:

{controller=Home}: This segment represents the controller name. If no controller is specified in the URL, it defaults to "Home." For example, if you access /, it maps to the Index action of the HomeController.

{action=Index}: This segment represents the action method name. If no action is specified in the URL, it defaults to "Index." For example, if you access /Employee, it maps to the Index action of the EmployeeController.

{id?}: This is an optional segment that represents a parameter named "id." The ? makes it optional, meaning you can access URLs with or without an "id" parameter. For example, /Employee/Details/1 and /Employee/Details both work.

Essentially, this route configuration allows you to access controllers and their actions using URLs like /ControllerName/ActionName/Id.

app.MapFallbackToController("Index", "Employee"):

csharp code"

app.MapFallbackToController("Index", "Employee");

This line of code configures a fallback route that, when no other route matches, directs the request to the Index action of the EmployeeController. Let's clarify how this works:

When a URL doesn't match any of the routes defined in your application (e.g., a URL that doesn't match the "default" route or any other custom routes), the MapFallbackToController method allows you to specify a controller and action method to handle such requests.

In this case, if a request doesn't match any known route, it defaults to the Index action of the EmployeeController.

The purpose of this is to ensure that if someone enters a URL that doesn't match any specific route, they will still see some content, which is the Index action of the EmployeeController in this context.

So, in summary:

The first route configuration (app.MapControllerRoute) defines the default routes for your application, determining how URLs map to controllers and actions.

The second configuration (app.MapFallbackToController) acts as a fallback route for handling requests that don't match any other route, ensuring that some content is displayed when an unknown URL is accessed.

Together, these configurations help define the routing behavior of your ASP.NET Core MVC application, allowing you to map URLs to controllers and actions.

With and without pageMaster:

Certainly, here's a simple example of creating a web form page both with and without a master page in ASP.NET Web Forms:

1. \*\*With Master Page:\*\*

Let's assume you have a master page named "Site.Master" with a common header, navigation menu, and footer. You want to create a new content page that inherits this master page's layout.

- Create a new web form page (e.g., "ContentPage.aspx") and select "With Master Page" during the creation process.

- In the code-behind file for "ContentPage.aspx," you can add specific content to this page without worrying about the common layout. Here's a simple example:

```html

<!-- ContentPage.aspx -->

<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" runat="server">

<h1>Welcome to Content Page</h1>

<p>This content is unique to the content page.</p>

</asp:Content>

```

In this example, "ContentPage.aspx" uses the master page "Site.Master," and you only provide the unique content within the `<asp:Content>` tags.

2. \*\*Without Master Page:\*\*

Let's say you want to create a standalone web form page without any common layout from a master page.

- Create a new web form page (e.g., "StandalonePage.aspx") and select "Without Master Page" during the creation process.

- This page won't inherit any layout from a master page. You can design it as a standalone page.

```html

<!-- StandalonePage.aspx -->

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">

<head runat="server">

<title>Standalone Page</title>

</head>

<body>

<h1>Welcome to Standalone Page</h1>

<p>This page doesn't use a master page.</p>

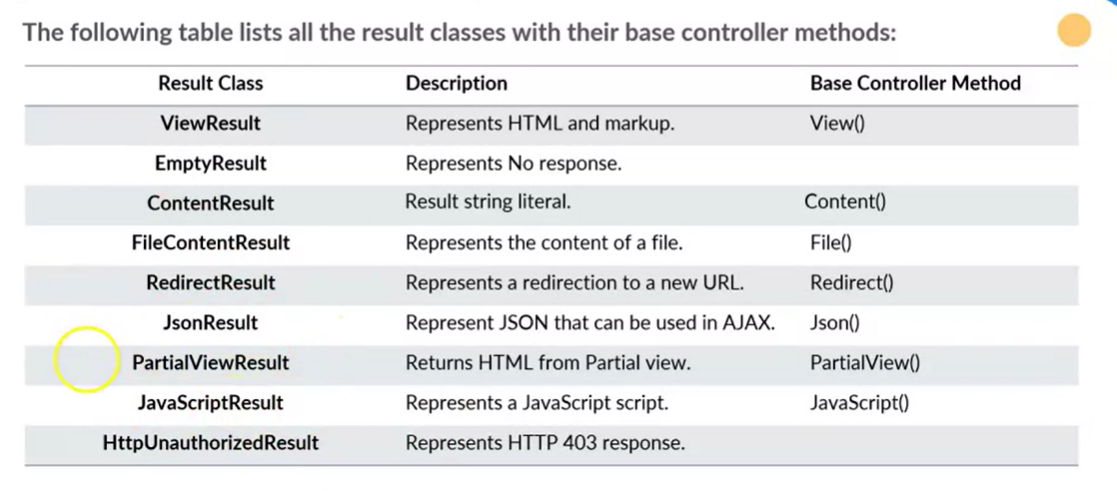
</body>

</html>

```

In this example, "StandalonePage.aspx" is a self-contained page without any master page.

You can navigate to these pages in your web application, and you'll see the difference between the one that uses a master page and the one that doesn't. The one with a master page will have the common layout elements from the master page, while the standalone page won't. These examples illustrate the choice between "With Master Page" and "Without Master Page" in ASP.NET Web Forms.



MVC all type of ways included in code.

Razor view engine:

Razor simplifies the process of creating dynamic web content in ASP.NET applications by providing a clean and readable syntax for mixing HTML markup with server-side C# code.

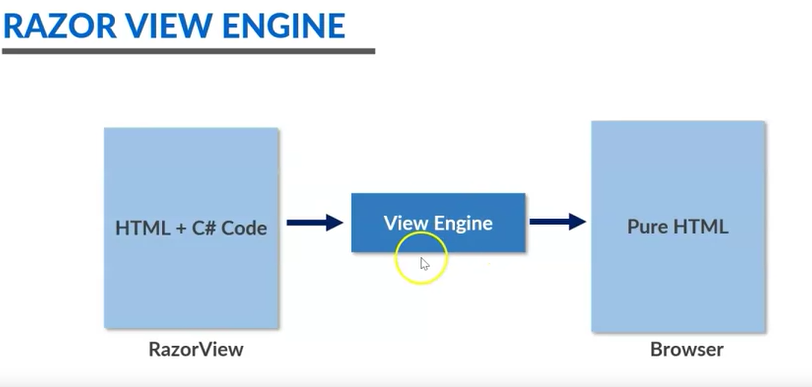
Syntax:  
@CODE.PROPERTY

<P>@@USERNAME</P>

<P>@USERNAME</P>

**<P>@@USERNAME</P>**:

* In this snippet, the Razor syntax **@@** is used as an escape sequence.
* The **@@** sequence is treated as a literal **@** character in the HTML output.
* The result would be a paragraph (**<P>**) element containing the text **@USERNAME** as it is, without server-side code execution.



1. **Layouts**:
   * A layout is a common HTML structure that wraps around multiple views to provide a consistent look and feel to your web application.
   * Layouts typically contain elements like headers, footers, navigation menus, and styles that should be shared across multiple pages.
   * You define a layout as a regular Razor view (**.cshtml**) with placeholders for the main content. In ASP.NET Core MVC, a common convention is to place layouts in the **Views/Shared** folder.
   * To specify that a view should use a particular layout, you set the **Layout** property within the view.

Example Layout (**\_Layout.cshtml**):

<!DOCTYPE html> <html> <head> <!-- Common styles and scripts --> </head> <body> <header> <!-- Navigation menu --> </header> <div class="container"> @RenderBody() <!-- This is where the main content of views will be inserted --> </div> <footer> <!-- Footer content --> </footer> </body> </html>

Setting Layout in a View:

@{ Layout = "\_Layout"; // The name of the layout view without the .cshtml extension } <!-- The content of the view goes here -->

1. **Sections**:
   * Sections allow you to define placeholders within a layout that can be filled with content from individual views.
   * You define a section within a layout using the **@section** directive.
   * In your views, you use the **@section** directive with the same name to provide content for that section.
   * Sections are helpful when you need to customize specific parts of a layout for different views, like the title or a sidebar.

Example Layout (**\_Layout.cshtml** with a Section):

<!DOCTYPE html> <html> <head> <!-- Common styles and scripts --> <title>@RenderSection("Title", required: false)</title> </head> <body> <header> <!-- Navigation menu --> </header> <div class="container"> @RenderBody() <!-- Main content --> </div> <footer> <!-- Footer content --> </footer> </body> </html>

Using a Section in a View:

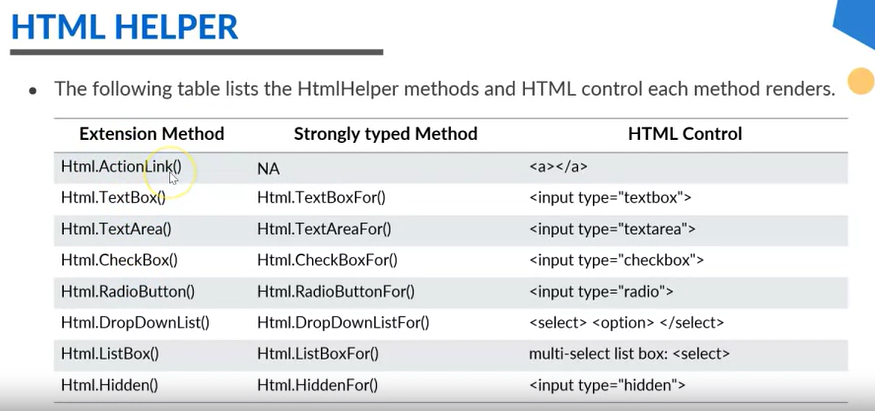
@{ Layout = "\_Layout"; } @section Title { <title>Page Title</title> } <!-- The content of the view goes here -->

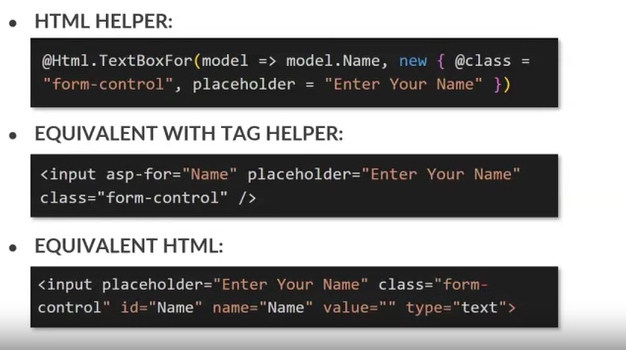
HTML HELPER:

* + To use an HTML Helper, you call the helper method within your Razor view, providing any necessary parameters.
  + HTML Helpers are invoked using the **@** symbol followed by the helper method, such as **@Html.TextBoxFor(model => model.Name)**.

1. **Example**:

@Html.TextBoxFor(model => model.Name, new { @class = "form-control", placeholder = "Enter your name" })





Functionality added in code.

Convention Routing and Attribute Routing:  
//CONVENTIONAL ROUTING

app.MapControllerRoute(

name: "default",

pattern: "{controller=Home}/{action=Index}/{id?}");

//ATTRIBUTE WRITING

[Route("/attribute/routing/home")] //attribute routing overlapping conventional home routing not allowing to work

public IActionResult Index()

{

return View();

}

[Route("/")] //attribute routing overlapping conventional home routing not allowing to work

public string message()

{

return "hello Message";

}

DataPassing Techniques:

ViewBag  
ViewData  
TempData  
Session  
Cookies  
QuerString

**ViewBag and View Data:**

1. **ViewBag**:
   * **ViewBag** is a dynamic property that uses the dynamic C# type. This means you can add properties to it on the fly without strongly defining their types.
   * It is primarily used for transferring small amounts of data from the controller to the view.
   * Data stored in **ViewBag** is not strongly typed, so you need to be careful about runtime errors if the data types are not handled correctly.
   * Accessing data from **ViewBag** in the view typically requires casting, like **@(int)ViewBag.MyValue**.
   * Since it's dynamic, it's more susceptible to runtime errors if property names are misspelled or the data type doesn't match.

Example in Controller:

Example code:  
ViewBag.Message = "Hello, ViewBag!";

Accessing in View:

<p>@ViewBag.Message</p>

1. **ViewData**:
   * **ViewData** is a **ViewDataDictionary** which is a dictionary-like collection that stores key-value pairs.
   * It is also used for transferring data from the controller to the view, but unlike **ViewBag**, it is strongly typed.
   * Data stored in **ViewData** is type-checked at compile time, reducing the chances of runtime errors.
   * Accessing data from **ViewData** in the view may require casting, similar to **ViewBag**, but it's less prone to runtime errors because of its strong typing.

Example in Controller:

ViewData["Message"] = "Hello, ViewData!";

Accessing in View:

<p>@(string)ViewData["Message"]</p>

TempData:

Model-View-Controller Communication: \*?

make web hosted onebmit details --

web form pending vid -- https://www.youtube.com/watch?v=RzDPaDHIbWQ

then coursera --

document of these

alignment