**TUTORIALS START**

-First Open Visual Studio 2022

-Create a new project and type MVC and select C# ASP.Net Core, then click next

-Name the Project Name: BulkyBookWeb

And the Solution Name: BulkyBook

Must be different name but similar.

-Next select the Framework we selected: .NET 6.0(Long Term Support)

-Authentication type : None

**-Check** Configure to HTTPS.

-Then Create the project

IN the solution explorer we can see the folder created by default.

-Once we right click the **BulkyBookWeb** project name and select “**Edit project file**” we can see the “**Configuration**” which we can see the target framework.

Inside we can see the **<ItemGroup></ItemGroup>** which contains all the **NuGet packages** that we are using inside the project. In future we will be installing more pacjages when we connect to database using the **Entity Framework** core and so on. When we add **Nuget** package new entry will be made into csproj file or the project file. Now usually you do not work with the project file. It is always good to know that we have all the packages and the references listed in the project file if incase you need to acces that.

We can expand the **Dependencies** in **Solution Explorer** and we can see the **Packages** we saw previously inside our **project file** above example.

**launchSettings.json**

Next we see the **Properties** folder which we can see the **“launchSettings.json”** We have the different profiles “Using” which we can run our application. We can see the **BulkyBookWeb** profile and **IIS Express Profile.** If we use the **IIS Express** profile, we know the port number that will be used here is explicitly defined above **“sslPort”: 44351.** If we us the **BulkyBookWeb**, in that case, it will use **localhost:7059** and **localhost:5112** base on **https** or **http**. If we run it in visual studio 2022 we can see an command and prompt pop and it will launch the website on **Port:7059** we can see the logs in Command and prompt. If we select the run through **IIS Express** it will use the **port:44351.** The default is using the BulkyBookWeb but, we will be changing that and use the IIS express later on.

**wwwroot** (folder) [**CSS,JavaScript,Images and Libraries stored inside ( wwwroot folder)**]

NOTE: *Does not have any C# Files inside.*

We will see all the static files of our project. So any **static files** like **CSS**, **JavaScript**, **images**, or any **libraries**, everything will go inside the **wwwroot** folder. The **wwwroot** folder **will not have any C# Files,** this folder is only meant to serve the static files of our application. So we will be using the static folder extensively throughout the course, when we are adding some JS or some images or any other libraries. The **wwwroot** folder will be the root folder of our application. Always remember, if we ever have to add any **static file**, it will always go inside the **wwwroot** folder.

**appsettings.json**

This is the file in which we will be **adding all of the connection strings and secrets of our application**, like we might have some **API KEYS**, we might have some **SendGrid keys**, we have **stripe payment keys** any of the **static secret keys** that we want to save, we will be storing them inside **appsettings.json**. If we expand **appsettings.json** we can see the **appsettings.Development.json** so we can create new JSON files and it will **automatically bundle them inside appsettings.json**. Like if we create for another environment, **appsettings.staging.json** then we can have **appsettings.production.json** all of them will be bundle in **one umbrella**. And then based on the environment variable, we can configure it to use the different **appsettings file**. Because connection string for a database in development will be different if we compare that to staging preview of production(Enterprise). So that way, we can go into those configuration. We will be using just **appsettings.json** right now, because we will be working with the **localhost**. Also in production, there are **multiple ways of saving secrets** like you can add them to the **Azure Storage** world and much more. To get started, **we need to remember, all of our application secrets must be inside appsettings.json**, and not direclty inside any of our CS or Class Files. Now we will add our **connection string** next.

**Program.cs**

This is the file that is responsible for running the application. When we open this file, we can see that we have a variable builder “**var builder = WebApplication.CreateBuilder(args);**” where the **WebApplication.CreateBuilder** is passed with the built-in arguments **(args).** When you run with the dotnet command, you can pass custom arguments here if you want, with that, it will configure the application, and it will create the web application builder object. Now in the previous topic, we saw that we can use **dependency injection** with **.Net Core**. When we want to **register** anything with our **dependency injection** container, we will be doing that here:

builder.Services.AddControllersWithViews();

var app = builder.Build();

So, let’s say if we want to register our **database** or **email** or anything else, we will have to do that between the builder. And before we call build on the builder object. So right here, we are just adding one service to the container which is: **builder.Services.AddControllersWithViews();** we are adding the service in the container because we are using MVC application for our project. If we we’re using razor pages, then service **builder.Services.AddControllersWithViews();** will be different. Now in the future, when we configure database in our project, and we add that to dependency injection, we will be adding a new service here in our container => **builder.Services.AddControllersWithViews();** our **DBcontext**. If we are working with any version prior to .NET 6 or even some of the initial preview versions of .NET 6, then this file was divided into a separate startup.cs class file. And the services that we add to container were inside a method configure services. And everything from line 9 onwards was inside a configure method. So, what we have on the top is we will be adding services to our container, then we need to configure request **pipeline**. And that **pipeline** will be configured from this section:

if (!app.Environment.IsDevelopment())

{

app.UseExceptionHandler("/Home/Error");

// The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(

name: "default",

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

app.Run();

You might be wondering what is this **Pipeline,** the pipeline specifies how application should respond to a **web request**. When your application **receives a request from the browser**, that request goes back and forth through the **pipeline**.

**.Net Core Pipeline Visual Example**

**Different Browsers**

**Mozilla**

Request from Browser

**Google Chrome**

**Microsoft Edge**

Auth

MVC

Static Files

The pipeline specifies how **application should respond to a request that is received**. When your application receives a request from the browser, **that request goes through the pipeline**. In the **pipeline**, we can add items that we want. **Pipeline** is made up of different **middleware’s**, and **MVC** is a **type of middleware itself**. So, if we want an application to be built using **MVC**, we have to add that middleware. Other example could be **authentication middleware**, **authorization middleware**, and so on. What exactly happens is when your request will go through each of the **middleware**, it gets modifies by them, and eventually it is **passed to the next middleware** if that is the last middleware in the pipeline, the response is returned back to the server.

if (!app.Environment.IsDevelopment())

{

app.UseExceptionHandler("/Home/Error"); <= Error page.

// The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.

app.UseHsts();

}

You can see in the pipeline above, first we are checking if it is development or not in the environment. If it is, then we are adding the use developer exception page “**app.UseDeveloperExceptionPage();”** that will show you user friend exceptions, so that we can debug and solve them. But if it is not development, then we are just redirecting them to an error page.

app.UseExceptionHandler("/Home/Error"); <= Error page.

// The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.

app.UseHsts();

The next middleware is HTTPS redirection **app.UseHttpsRedirection();.** And then we have a middleware to use to use our static files **app.UseStaticFiles();** that are defined in **wwwroot** folder. We also have routing middleware **app.UseRouting();** , and we have authorization middleware **app.UseAuthorization();** and we add authentication to our project, we will have to add a new middleware inside the **Program.cs** as well. Then we have used a map controller route that will map the different pattern that we have

app.MapControllerRoute(

name: "default",

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

For **MVC**. Based on this routing above, it will be able to redirect a request to the corresponding **controllers** and **action**, then you should always keep in mind that order of pipeline is extremely important. The way you write **middleware’s** in the **pipeline**, that is exactly how the **request will be passed**. So first, **routing** will be done and it checks for the **authorization** and so on.

app.UseRouting();

app.UseAuthorization();

So in this scenario, if we want to used authentication to our pipeline, we have a middleware, which is “**app.UseAuthentication();”.** **Authentication middleware should always come before you authorize a user.**

**THIS WILL WORK**

app.UseRouting();

**app.UseAuthentication(); <= Middleware if we want to use authentication to our pipeline.**

app.UseAuthorization();

If we will move our authentication middleware below our **app.UseAuthorization();** it will not work. As we said above **authentication middleware should always come before you authorize a user.**

app.UseRouting();

app.UseAuthorization();

**app.UseAuthentication(); <= Middleware if we want to use authentication to our pipeline.**

**THIS WILL NOT WORK**

Because you only **authorize the user that is authenticated**. That is the **basic fundamentals of authentication and authorization**. So, if we place the pipeline in some different order that will break things.

app.MapControllerRoute(

name: "default",

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

Inside the **endpoints** here, you can see we have a **controller name** and **action name**, and some **ID**. This **controller route** will make more sense when we understand **routing**.

**Details on how Routing Works**

You can see when it comes to routing in MVC application, we have controllers and we have actions. Before we explore the routing, let’s walk through the main components of an MVC application, which is folder name **Models**, **Views**, and **Controllers.** This is a brief overview below:

**MVC ARCHITECTURE**

**MODEL**

Represents the shape of the data

Now, if we remember, we had three folders it is for **Models**, **Views**, and **Controllers**, and that is what **MVC** stands for. The first thing in **MVC** is **Model**, which represents the **shape of the data**. **A class in C# is used to describe a model**. The **Model component** corresponds to all the **data related logic** that users work with. Let’s say inside your application, you have a table that stores all the **Category** and all the product details then **that product will be a model itself. Model** basically represent **all the data in our application**, it can be a **table** that we are storing inside **SQL Server**, or it can be a **model**, which will be a **combination of multiple tables**, and so on. This **model** can either represent the **data that is being transferred between view and controllers**, or any **business-related data model that will represent all the tables of the database**. So, **if we have 10 tables in our database, we will have at least 10 models that corresponds to them**. There is also more complexity, but we will go into details later on. Right now, we can think **all the tables in our database will be a class file**, which will be a **model** and **all the properties “public string Name {get; set;}” of that class file will be the columns of the table**. That is a simple relation that we can think of right now.

**VIEW**

Represents the user interface

Then we have **View** in an **MVC**, which is the **user interface**. You can be tired of **HTML** and **CSS** that you write, to make things **fancy and beautiful**. Whatever you see on the website with your eyes, is basically the **View** that is being displayed to you. But now you need to think of what happens if in a website, you have a **button and you click that button**. What happens is that **View will interact with your model to display some of the data**. But **View** **does not interact directly with the models**. For that we have something known as **Controller**.

**CONTROLLER**

Handles the user request and acts as an interface between Model and View

**Controller** acts as an **interface** between **Model and View** to **process all the business logic** and **incoming request**. So, **Controller** acts as an **interface** between **Model and View to process all the business logic** and **it manipulates** that **data** using **Model** and **interacts** with the **View** to render the **final output**. This is just a brief overview of how **Mode**, **Views**, and **Controller** works.

So, let’s say if a **user** clicks on a **button**, **Controller** is the first thing that will receive that request and control will have lots of **action methods**, based on those action methods **Controller** will redirect the request to one of the **action method** and **Controller** will use the **Model** it will **fetch all the data** that it needs to display inside the **View**. Once the **View** is rendered, it will pass all of that to the **Controller** and **Controller** will then **pass a response** which will be **sent back** and the **user** will finally be able to see the **page**. So, we can see **Controller** can be treated as **heart of the application**. That is where we will have **all the logic of our application**, and it is the one which will be interacting with **Models** and **Views**. So, with that in place, now we see that the request first comes to the **Controller** and **its action methods**.

**VIEW**

Represents the user interface

**MODEL**

Represents the shape of the data

2. Get Data

3. Get Presentation

1. Request

USER CLICKS

**CONTROLLER**

Handles the user request and acts as an interface between Model and View

**VIEW (User Interface)**

4. Response



So, with that general idea if we go back, we can see inside the **app.MapControllerRoute**, we have a pattern where we define a **Controller** "{controller=Home} and an **action method** {action=Index}/. So here we are saying that the default if nothing is provided it should go to the **home controller**"{controller=Home}, and it should call the **index action method**{action=Index}.

app.MapControllerRoute(

name: "default",

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

But that is 10,000 fate overviews. Let’s take a look at routing a little more with some theory.

**ROUTING IN MVC**

The URL pattern for routing is considered after the domain name.

* <https://localhost:55555/Category/Index/3>
* https://localhost:55555/{controller}/{action}/{id}

|  |  |  |  |
| --- | --- | --- | --- |
| URL | Controller | Action | Id |
| <https://localhost:55555/Category/Index/3> | Category | Index | Null |
| [https://localhost:55555/Category](https://localhost:55555/Category/Index/3) | Category | Index | Null |
| <https://localhost:55555/Category/Edit/3> | Category | Edit | 3 |
| <https://localhost:55555/Product/Details/3> | Product | Details | 3 |

Before we see routing in action, let’s see routing with some examples. You can see we have general pattern of routing. The first thing highlighted in **yellow** here is the **domain of the URL**. When we run on the local computer, you will have a **localhost** and a **port number**. Whatever it is, after that port number will be the **route** that we want to use when we are calling a **page to be loaded Category/Index/3**. In the first example, we see we have something called as **Category**, then we have an **Index** and **some number**. When we are working with **MVC**, after our **port number**, or **domain**, whatever is the first thing that we have will be the name of the **Controller,** then the next forward slash after that will be the **Action** of **Controller**. And after that if we have something that will be the **ID**. This is the **pattern of routing** with **MVC**. If we go back to the application, we can see the same format right here.

app.MapControllerRoute(

name: "default",

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

First, we have the **Controller** name, then forward slash, we have an **Action** name, then forward slash, we have the **ID,** that corresponds with the request that we have here. Keep in mind that **ID** is an **optional field**, **Controller** and **Action** are **not optional**. But if they are not defined, we have set a **default route** “name: "default",”, that if there is no **Controller** and **Action**, you can use home **Controller** {controller=Home}/and index action {action=Index} as the **default route**. Because of that, we have our **home controller**. And we also have the **index action**, which I will show you. Before we dive into those details. Based on the understanding that we have here, I have given some sample URL, I want you to try to find out what will be the controller, action and ID based on this URL, I can remember if controller is not defined default, one that we have in our application is let’s go back the home controller. And if the action is not defined, that will be index action.

So, based on that the first URL that we have, the **Controller** name is **Category**, the **Action** is **Index**, and we do not have any **ID**.

|  |  |  |  |
| --- | --- | --- | --- |
| URL | Controller | Action | Id |
| <https://localhost:55555/Category/Index/3> | Category | Index | Null |

For the next one, we have **Controller** name as **Category**, we do not have any **Action.** So, Index will be the default action. And finally, ID is null.

|  |  |  |  |
| --- | --- | --- | --- |
| [https://localhost:55555/Category](https://localhost:55555/Category/Index/3) | Category | Index | Null |

Next, what we have for the third one, we have **Controller** as **Category**, **Action** as **Edit** and **ID** as **Three**.

|  |  |  |  |
| --- | --- | --- | --- |
| <https://localhost:55555/Category/Edit/3> | Category | Edit | 3 |

The last one, we have **Controller** as **Product**, **Action** as **Details**, and we have **ID** as **(3)** **three**.

|  |  |  |  |
| --- | --- | --- | --- |
| <https://localhost:55555/Product/Details/3> | Product | Details | 3 |

So, with that if you get a URL, now you can identify what is the **Controller name**, what is the **Action name**, and if there is and **ID** or not. So, with that brief understanding of routing, lets actually run our application, see routing in action.

Now we want to understand routing in **MVC**. This is one of the tricky topics when someone start learning MVC. So, we want to make sure that we get familiar with the routing. Before we see the complete routing in action. We may have a walk through to the **three folders** that we have. We have a folder for **Controllers**, for **Models**, and for **Views**. By default, we have a **HomeController.cs** that has been created. **Models** will be **all the data related models** that people want in the project. So, let’s say if you are dealing with **Products** that we want to **display on the page**, we will have a **Product Model** and we will be using that **Model (Product Model)** in **Controller** and **Views**.Right now, we can just think of **Models** as **Tables** that we want in our **Database**. If we want a **Product table**, we will have a **Product Model.** That is not always the case, but we will explore **Models** later on. The main thing we want to work on is understanding **Controllers** and **Views**. **Controller** is the **heart of the application**. At the same time **Views** is what will be **displayed on the screen** when **user** is looking at the page. Now the way **navigation** works is when we have a **Home Controller**, all of the **Views** or **UI pages** that are displayed with the **home controller** will be placed inside the **same folder** name as the **controller’s name**. So, we can see when we expand **Home folder under Views**, they have also created a **Home Folder** by default, that is also a convention with the naming of **Controllers**, it should always end with the keyword of **controller**. That is how the application will know that this is a **Controller.** Example of that is we have inside our folder **Controllers** a named class **HomeController.cs,** when we go to our folder **Views** it is necessary that we also have a folder named **Home.** If we open the class **HomeController.cs** we will see some code inside, we have class with the name “HomeController” and it implements the default or the base class of **Controller:** “public class HomeController : Controller”. And right below we are registering the logger using Dependency Injection.

private readonly ILogger<HomeController> \_logger;

public HomeController(ILogger<HomeController> logger)

{

\_logger = logger;

}

We do not go into that detail right now; we will explain dependency injection in much detail. But for now, to understand routing, we can see we have two action methods. Inside controllers, we can have multiple action method. (IActionResult => Defines a contract that represents the result of an action method.)

public IActionResult Index()

{

return View();

}

public IActionResult Privacy()

{

return View();

}

If we go back to the URL here, you can notice that we have the Controller name and Action name.

<https://localhost:55555/Category/Index/3>

If the URL was **https://localhost:55555/Home/Privacy**, then it will go to **Home Controller** look for **Privacy** action method. And it will load the **contents** that we have there. The **return type for an action method** is (**IActionResult** => Defines a contract that represents the result of an action method.), because I action result is an **abstraction for multiple return type**, it can return a **View** “return View();”, it can redirect to some action method, or it can redirect a page and much more. Then if we said that if the URL is **https://localhost:55555/Home/Index**, it will return the View that we have defined here:

public IActionResult Index()

{

return View();

}

Now we might be thinking where is the **View** we are talking about, that **View** will be inside the **Views** **folder**. The way it finds our maps the **View** for this **Index Action Method** is inside the **Views folder**, it will look for the **name of the Controller**, which is **HomeController**.**cs**. “public class HomeController : Controller => Folder name “Home””. Inside **Home Folder**, we will have a **View** with the name of this **Action Method**  and it is **Index()** action method.

public IActionResult Index()

{

return View();

}

So, that will be mapped to this particular view. If we open that **View(Index.cshtml),** we have some **HTML** and **Bootstrap classes**. So here, nothing fancy is going on, we are just displaying some text.

@{

ViewData["Title"] = "Home Page";

}

<div class="text-center">

<h1 class="display-4">Welcome</h1>

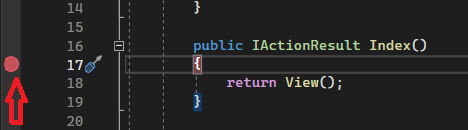
<p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building Web apps with ASP.NET Core</a>.</p>

</div>

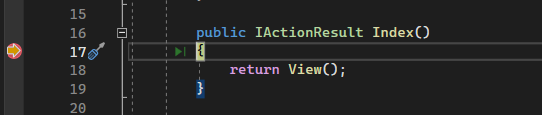
**Run our application** [**BulkBookWeb**] and we can see the route in action. In the **URL** right now, nothing is present “**https://localhost:7059**”. What should happen if there are no **Controllers** and **Actions** in the URL, we define that inside the **Program.cs**. We said of nothing is present, default that to home controller and index action method:

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

So, what we see on the screen is from the **Home Controller** **Index action method**. To confirm that if you go to the **Home Controller**. We can add a **debugging point** by clicking anywhere on the window.



What that will do is when it hits this controller above, this will be yellow because it hits our Breakpoint.



So that means if nothing is present in the URL, it is calling the **HomeController** **Index Action** method.

public IActionResult Index()

{

return View();

}

We can also try this into our **Privacy** action method:

public IActionResult Privacy()

{

return View();

}

Just like the Breakpoint we’ve added above into our action method. If we click the **View();** which is underlined below right click and select “**Go To View**”, it will automatically redirect us into the View(**Index.cshtml) .**

public IActionResult Index()

{

return View();

}

Inside of View(**Index.cshtml)**

@{

ViewData["Title"] = "Home Page";

}

<div class="text-center">

<h1 class="display-4">Welcome</h1>

<p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building Web apps with ASP.NET Core</a>.</p>

</div>

Because it knows it has to find the **Home folder** inside there, there will be and **Index.cshtml View**. This is how **Controllers** and **Views** are associated.

Now we saw the Home folder inside Views that corresponds to the views of the home controller. But on top of that, we have something called a shared, Shared Folder is used for partial views. Partial views are similar to user components if you are coming from classic C#. So, it’s basically a view that you can call within multiple place in your application. Along with that, we have a special partial view, which is the **\_Layout.cshtml** this is the default master page of our application. So if we open that up, we can see the styling at the top, header and we have a **div** where we render the body **@RenderBody**, since this is the master page, whatever we displayed inside the other **Views**, it will use this **\_Layout.cshtml** as the **default master page**. So inside index, when we are displaying this content below:

@{

ViewData["Title"] = "Home Page";

}

<div class="text-center">

<h1 class="display-4">Welcome</h1>

<p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building Web apps with ASP.NET Core</a>.</p>

</div>

It was displaying that with the navigation on the top, we also have a **footer inside \_Layout.cshtml**, common **JavaScript** that we want across the application. We have the **main HTML** and **body tag**. We have header above where we add the styling:

<head>

<meta charset="utf-8" />

<meta name="viewport" content="width=device-width, initial-scale=1.0" />

<title>@ViewData["Title"] - BulkyBookWeb</title>

<link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />

<link rel="stylesheet" href="~/css/site.css" asp-append-version="true" />

<link rel="stylesheet" href="~/BulkyBookWeb.styles.css" asp-append-version="true" />

</head>

We will be updating this in future discussions. Whenever we want to add some **CSS** and **JS globally**, this is the place where we will be adding that. We also have the \_**ValidationScriptsPartial.cshtml** and this is a **Partial View** where we are just adding scripts for some **validations**. Wherever in some **Views**, let’s say **Index View**, we want to use **Validations**, then we will include this **Partial view** on that page. So that way, we do not have to write those script tags, we will just include this Partial View \_**ValidationScriptsPartial.cshtml** and that will be included. We also have an **error** partial view **Error.cshtml** that will be used to **display the errors** that we encounter into our application. We will be adding more **partial views as we proceed**. Always remember that **Shared** Folder will contain all the **Partial Views** and **\_Layout.cshtml** is the **master page of our application**. We can see that on our application, we can see the **header** and the **footer** and we have also the **body**. So where we have this **@RenderBody** at that place, whatever we had inside the **Privacy**, these two lines are being displayed:

<h1>@ViewData["Title"]</h1>

<p>Use this page to detail your site's privacy policy.</p>

Now we also have the **\_ViewImports.cshtml**  inside display the global namespace right here:

@using BulkyBookWeb

@using BulkyBookWeb.Models

@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers

So, inside our application, let’s say we want to access some namespace in all of the pages, if we add the **USING statement** here example:” @using BulkyBookWeb.Models” it will be accessible across all the P**ages**, **Controllers, and Classes** in our project. That way we do not have to type this namespace every time. One imporant thing that we see here is the tag helpers:

“@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers”.

**Helpers are bindings** that are provided by the .NET Core team that looks like **HTML tags**, but they are **special tags** that are **adopted by the Microsoft team** from other languages. After looking at the **success like Angular and React**. Example here was in **\_Layout.cshtml,** if we go back to our running application and click on Privacy link we’ll see that it uses the URL **<https://localhost:55555/Home/Privacy>** here, it is provided using a Special Tag Helper, we can see it on our **\_Layout.cshtml** where we have tag Helpers, example of that are these:

**asp-controller**="Home" **asp-action**="Privacy"

Tag helpers will start with the prefix of “asp-” then the “name”

**asp-action and the asp-controller**. So here we have a tag helper

**asp-controller**="Home" which says **asp-controller** and then we define the controller name for routing “Home” it is the action to be called, and that is the ASP action. We will be using Tag Helpers later on. Tag helpers have been included in the project and we define them **globally** inside the **\_ViewImports.cshtml** file. The last file inside the **Views Folder** is the \_**ViewStart.cshtml,** this file will **define what is the default master page for our application**

@{

Layout = "\_Layout";

}

Inside **Privacy.cshtml**, we see we have not defined what will be the master page. But it is by default using **\_Layout.cshtml,** because that is what has been defined inside the **\_ViewStart.cshtml** file. For example if we want to explicitly define a master page for **Privacy**, which is different, we can do that directly by defining that on the top.

@{

ViewData["Title"] = "Privacy Policy";

}

<h1>@ViewData["Title"]</h1>

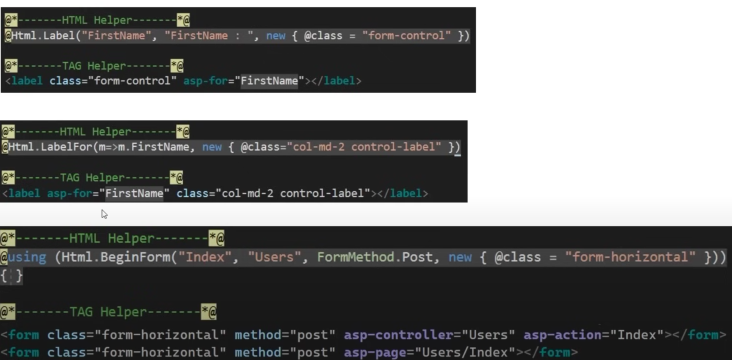
<p>Use this page to detail your site's privacy policy.</p>

And that will take preference over what is defined in **\_ViewStart.cshtml** but ViewStart will have the default master page for the application. This overview was critical when it comes to understanding the default Views that have been provided.

**TAG HELPERS(Similar to AngularJS directives)**

* Tag Helpers are introduced with ASP.NET Core. (Brand new). - Microsoft looked at the success around libraries like Angular, React, and decided that implementing an Angular directive like experience in the new ASP.NET Core that they’ve decided to create Tag Helpers from grounds up. Even though there are similarities between Angular directives and Tag Helpers, there is a major difference.
* Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. While the AngularJS directives are all about client side rendering. If we have worked with older versions of .NET Core, we had something called as HTML helpers. They are still around, a Tag Helpers are being modern with the tag like approach, so it’s much user friendly.
* Tag Helpers are very focused around the HTML elements and much more natural to use as compared to HTML Helpers.

**TAG HELPERS EXAMPLES**

****

This examples above compared the **Tag Helpers** to the old **HTML Helpers**. As we noticed **Tag Helpers simplifies** all of them, Because we can use the existing label tag, and we will just add a **Tag Helper “asp-for”** and bind that to a Model **asp-for = “FirstName”.** So we can see things are getting much more simpler when we are using **Tag Helpers**. That being said, if we are coming to .NET brand new, and we have never worked with HTML Helpers, this is even better, we don’t have to worry about the old syntax and what is different. We will be working with Tag Helpers in upcoming lessons. This is just a brief snippet of what Tag Helpers are and how they look like when we are using the same tags, we just use the **asp-(Tag Helper Name).**

**Action Result**

When we will be working with a .NET Core application we will see the return type of **Action Result.** It doesn’t matter if we’re using MVC Application or Razor Page Application. In both the cases we can see that we have **IActionResult**

**MVC APPLICATION**

**Razor Page Application**

public IActionResult Index()

{

return View();

}

public IActionResult OnPost()

{

return Page();

}

We have **IActionResult** in **MVC** we are returning back **View** in this example. And in **Razor Page Application** where **Page Handler** we are returning back to the page. But the return type is IActionResult. **IActionResult** is a **generic** type that **implements all of the other return types**. Now if you want to be explicit about the return type in both of these cases, then that would look something like this.

**MVC APPLICATION**

**Razor Page Application**

public PageResult OnPost()

{

return Page();

}

public ViewResult Index()

{

return View();

}

If the return type is **View**, we can write **ViewResult**. But in **Razor Pages**, when we return back to page, we can write **PageResult.** So now what is the advantage of IActionResult?

**Action Result**

* ActionResult - is a result of action methods/pages or return types of action methods page handlers.
* Action result is a parent class for many of the derived classes that have associated helpers.
* The IActionResult return type is appropriate when multiple ActionResult return types are possible in an action.

**Action Result in Razor Pages**

|  |  |  |
| --- | --- | --- |
| ActionResult | Helper | Description |
| ContentResult | Content | Takes a string and returns it with a text/plaincontent-type header by default. Overlaods enable you to specify the content-type to return other formats such as text/html or application/json, for example. |
| FileContent | File | Returns a file from a byte array, stream or virtual path. |
| NotFoundResult | NotFound | Returns an HTTP 404 (Not Found) Status code indicating that the requested resource could not be found. |
| PageResult | Page | Will process and return the result of the current page. |
| PartialResult | Partial2 | Returns a Partial Page. |
| RedirectToPageResult | RedirectToPage  RedirectToPagePermanent  RedirectToPagePreserveMethod  RedirectToPagePreserveMethodPermanent | Redirects the user to the specified page. |
| ViewComponentResult |  | Returns the result of executing a ViewComponent. |

A Razor page can return Content, File, NotFound, Page, Partial, and Redirect to different pages. For all of them we have, ContentResult, FileContent, NotFoundResult, PageResult, PartialResult, RedirectToPageResult, ViewComponentResult. If we are working with MVC Application we can return back

**Action Result in MVC**

|  |  |  |
| --- | --- | --- |
| ActionResult | Helper | Description |
| ViewResult | View | Renders a view as a Web page. |
| PartialViewResult | PartialView | Renders a partial view, which defines a section of a view that can be rendered inside another view. |
| RedirectResult | Redirect | Redirects to another action method by using its URL. |
| RedurectToRouteResult | RedirectToAction  RedirectToRoute | Redirects to another action method. |
| ContentResult | Content | Returns a user-defined content type. |
| JsonResult | Json | Returns a serialized JSON object. |
| JavaScriptResult | JavaScript | Returns a script that can be executed on the client. |
| FileReuslt | File | Returns binary output to write to the response. |
| EmptyResult | (None) | Represents a return value that is used if the action method must return a null result (void). |

If we are returning any one of these, then we can use the individual return types for result based on the helper method.

But what if we were returning something like this?

**Razor Page Application**

public PageResult OnPost()

{

if(true)

{

return RedirectToPage(“Index”);

}

return Page();

}

public RedirectToPageResult OnPost1()

{

if(true)

{

return RedirectToPage(“Index”);

}

return Page();

}

public ViewResult Index()

{

if(true)

{

return RedirectToAction(“Index”);

}

return View();

}

public RedirectToActionResult Index1()

{

if(true)

{

return RedirectToAction(“Index”);

}

return View();

}

**MVC APPLICATION**

**MVC APPLICATION(Above Example)**

Right here in MVC, you can see we are returning if true, let’s imagine that was some condition here. Based on that if that condition is true, you want to redirect to action, else you want to return to view, this time you cannot have two different return types. If we use ViewResult here, return view is working, but redirect to action will throw an error.

public ViewResult Index()

{

if(true)

{

return RedirectToAction(“Index”);

}

return View();

}

Similarly, if we use redirect to action result, then redirect to action works. But return view will give you error.

public RedirectToActionResult Index1()

{

if(true)

{

return RedirectToAction(“Index”);

}

return View();

}

**Razor Page Application**

If we are working on razor pages and we have the same situation if we use page result, and return page will work. But redirect to page will fail.

public PageResult OnPost()

{

if(true)

{

return RedirectToPage(“Index”);

}

return Page();

}

And if we use redirect to page result, then return page will fail.

public RedirectToPageResult OnPost1()

{

if(true)

{

return RedirectToPage(“Index”);

}

return Page();

}

The solution to all of this is to use **IActionResult** in both places. Because this is a **parent class**, so it does not care which of its implementation is being returned, it will be able to handle all of them.

**MVC APPLICATION**

**Razor Page Application**

public IActionResult OnPost2()

{

if(true)

{

return RedirectToPage(“Index”);

}

return View();

}

public IActionResult Index2()

{

if(true)

{

return RedirectToAction(“Index”);

}

return View();

}

So that is a brief overview on how **IActionResult** will help us with the return type from action method in MVC application or page handlers in a razor page application.

Ends at 58:20