**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>

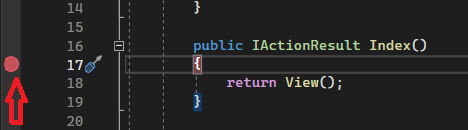
<p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building Web apps with ASP.NET Core</a>.</p>

</div>

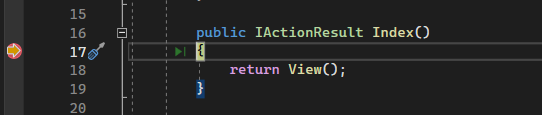
**Run our application** [**BulkBookWeb**] and we can see the route in action. In the **URL** right now, nothing is present “**https://localhost:7059**”. What should happen if there are no **Controllers** and **Actions** in the URL, we define that inside the **Program.cs**. We said of nothing is present,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

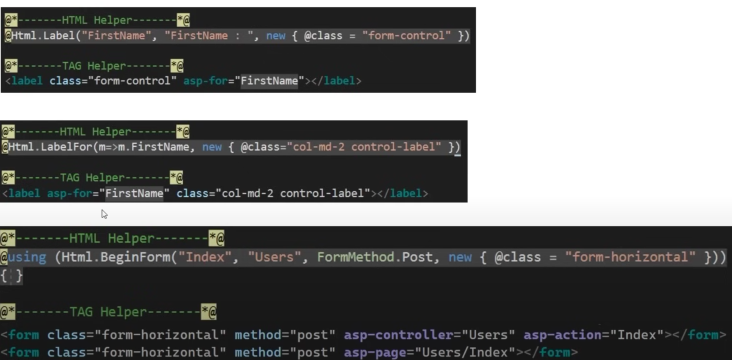
<p>Use this page to detail your site's privacy policy.</p>

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.

**HOT RELOAD**

This is a great capability that has been added with .NET 6. If our website does not refresh and reload for us, we can go in the **HOT RELOAD** drop down above and **check Hot Reload on File Save.** What that will do is whenever we make **some changes** inside our **Index.cshtml** for example, and we save it, it will automatically **refresh our browser**. This is very helpful when we are designing and **MVC and Razor Application**. No more **manual refreshing into our browser** the **Hot Reload** once check will do the work for us. Now, if we’re working with an application, which is not in **.NET 6**, then we need to go into our **Manage NuGet package** and search **Microsoft.Asp.NetCore.Mvc.Razor.RuntimeCompilation** then click **Install,**  to add this to our project. We can find the **Manage NuGet package** when we right click our **solution** which is the **BulkyBookWeb** for example, and we can see it below. Consider that we installed the **Microsoft.Asp.NetCore.Mvc.Razor.RuntimeCompilation,** after installing we need to go into our **Program.cs** to enable our AddRazorRuntimeCompilation by type this below our:

// Add services to the container.

*builder.*Services.AddControllersWithViews();

*builder.*Services.AddRazorPages().AddRazorRuntimeCompilation(); <= Here

var app = builder.Build();

But because of **Hot Reload** and the magic that we have, this is no longer required.

// Add services to the container.

*builder.*Services.AddControllersWithViews();

*builder.*Services.AddRazorPages(); <= Here

var app = builder.Build();

Because we have already an Hot Reload into our.NET 6 there is no need to add the

.AddRazorRuntimeCompilation(); we can just only add this *builder.*Services.AddRazorPages(); for razor pages.That’s the brief overview that we wanted to explain with the Hot Reload in .NET 6.

**Tutorials Guides Starts Here.**

Examine all the files and folder structures.(Created by default)

Creating a Model into our (Models Folder) below

When we work with any project, the main piece or the **Heart of the application** is the **Data**,we need a **Database** to **store our data**. So, we will be using **SQL Server** to create a **Database**, and **store all the data for our website**. With **.Net Core**, we might have heard about term called as **Entity Framework Core** as well. **Entity Framework Core** is what we’ll be using to **create Database**, and we will be using the same to perform all the **Data Operations**. So with **.NET Core**, we do not need **stored procedures or writing SQL Statements in the code**. **Entity Framework** is a **spot editor** that will help us with all the **Data related operations**. That being said, it does not mean that you cannot use stored procedures, you can still use stored procedures if we want. But typically with **MVC Application** and **Entity Framework core** is used to **manipulate the data layer**. We will go into those details. But right now, we first need to create our **Model**. **Model** will basically resemble a **Table** into **Database**. It is not always the case, but whatever **Tables** we need into our **Database**, we will need a corresponding **Model** for the **Code First Migration**. Now inside our **Models Folder** we will add a new **Class File,** We want to create a **table** for **Category**, so we will name our **Model** as **Category** then click **Add**. Now it creates a **public class file** and it place it inside the namespace **BulyBookWeb.Models.**

using System.Diagnostics;

namespace BulkyBookWeb.Models

{

public class Category

{

public int Id { get; set; }

public string Name { get; set; }

public int DisplayOrder { get; set; }

public DateTime CreatedDateTime { get; set; } = DateTime.Now;

}

}

That is just the location so it knows that it is inside **BulkyBookWeb** **(Models Folder)**. Now inside this **Category.cs** class, we need to create all the **Properties** that we found for our **Table**. We create and **ID**, **Name**, **Display Order**. Typing **prop** is the code snippet for property and press **TAB** twice to **create**. The properties we created now look like this:

namespace BulkyBookWeb.Models

{

public class Category

{

}

}

Then we want to push this **Model** into our **Database** to **create a table** with this **four** columns. But when we **create a database table**, we might want to add some **configurations**. Like we might want to see that **Id** is a **primary key**, and since it’s an **integer**, we want to make that an **Identity column**, so we do not have to **populate** that when we **create a row inside the table** that we will create. We could also add a **Validation** that **Name** is **Required Property,** and it should not be **NULL.** If we had to write a **SQL Statement**, we cold have done all of that by using **NOT NULL** an **identity** in our **SQL Script**. But how did we do that using **Entity Framework**? For that .NET team has come up with alternatives called as **Data Annotation.** So in the **Properties** where we want to configure some details, we have **special attributes** that we can use.

using System.ComponentModel.DataAnnotations;

using System.Diagnostics;

namespace BulkyBookWeb.Models

{

public class Category

{

[Key]

public int Id { get; set; }

[Required]

public string Name { get; set; }

public int DisplayOrder { get; set; }

public DateTime CreatedDateTime { get; set; } = DateTime.Now;

}

}

We want to make this **Id** column and **identity column** which will be the **Primary Key** of this table. In order to do that, we have a **Data Annotation**, or an **Attribute** known as **[Key]**. If we enter the **[Key]** we will notice an red squiggly lines **[Key**] if we hover on that, it will display an error that **key could not be found**. That is because we have to add a **Using Statement**

using System.ComponentModel.DataAnnotations;

To add **automatically** add the **using statement** above, we need to make sure that our **mouse cursor** is on the **[Key]** then press **“CTRL + .”** it will display that we can add **using statement** to resolve this. So we’ll press **Enter**, and that will add the **using statement** above which has the **Data Annotation** that we want. Once this **[Key] Annotation** has been applied, it will tell **Entity Framework Core** that “Hey when you create a script to create this table, you need to make sure that **ID** is a **Primary Key**, and it should also an **identity column**.”. It does all the configuration and talking by itself. We just need to write one **Attribute** and everything will be done for us. The next thing that we wanted to do is we want to make sure that name is a **required property**. So here we have **another attribute** which is [Required].Once you assign that, when it creates the script, it will make sure that **Name** is **NOT a NULL(able) Property**. There more **Data Annotations** that we can do for **validations**. With this in place, the initial equation of our **Model** looks good.

Creating a database inside SQL Management Studio using Entity Framework.

How do we create that in the Database now? Now we need to create a **table** and a **database inside our SQL Server**. Open up our **SQL Server Management Studio**. Make sure that **SQL Server Management Studio** is installed to our machine, because that is what we will be using to play with the **Database**. Once our **SSMS** opens up we will be using our **Local Database** example. **(LANCEGAMINGPC\SQLEXPRESS2k14)** then click **Connect**. As long as it is able to **connect**, that **server name** is what you will be using inside our **Connection String**. If we use something else then **it won’t work**. If we Expand the **Databases Folder** we can see a few **database** inside, but we want to create a new **Database** for our **Project**. So for that, we need to create a **Connection String** inside our project. In order to store all of the **secrets of our application**, we will be doing that inside the **appsettings.json** file. We can hard code the **Connection String** inside the class file but that is a **bad approach.** The **appsettings.json** file is the file where we should have all the **Secrets**. That way, if we have to update anything in the future, we know it will be always available inside the **appsettings.json** file. And then we can crate different app settings for different environment. Like if we have development, staging, preview, production, we can create all of those inside app settings. And we can configure to use that app settings when we deploy the application. That way based on our environment name, we can use different app settings, or different database and so on. We will be working directly in **appsettings.json** and should work with the local development. Below is a simple JSON file.

{

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft.AspNetCore": "Warning"

}

},

"AllowedHosts": "\*"

}

Now we’ll be adding a new Connection String inside. As we can see, it is just a dictionary with string key and value pairs. What we have inside "Logging" is known as a block, where inside the “Logging” block we have another block of “LogLevel” and that has two key value pairs.

{

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft.AspNetCore": "Warning"

}

},

"AllowedHosts": "\*"

}

Two key value pairs(UNDERLINED)

So we can either create a Block, or we can directly configure the Connection String. Now the .NET team thought that connection string is very common, and almost of the project will have a connection string, that is why they have create a default block for Connection String that we can use if we want. We can probably use something else. But we can also use the default block name that they have given.

{

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft.AspNetCore": "Warning"

}

},

"AllowedHosts": "\*",

"ConnectionStrings": {

"DefaultConnection": "Server:LANCEGAMINGPC\\SQLEXPRESS2k14;Database=Bulky;Trusted\_Connection:True;"

}

}

We can see it is automatically suggesting us that block name which is the “**ConnectionStrings**”, we can use something else, but we will use the exact same name for now. We have a reason behind that along the way. So inside this **code block**, we will have a **key value name** "DefaultConnection" we can use whatever we want, but we will just use the “**DefaultConnection**”. We can actually use any name that we want for the key right here. Within the **double quotes** on the **right side** we need to enter the **Connection String**, we are using at the moment the string LANCEGAMINGPC\\SQLEXPRESS2k14; because it is our **Default server name**. We can see the **default server name** once we opened the **SSMS** under **Server name: (Server Name)**. Before entering the our default server name it should start with “**Server:”** parameter as shown above example. Next property that we have is the Database name **“Database=Bulky;”**, then after that set the **“Trusted\_Connection:True”** trusted connection flag to be true, make sure there is **no spelling mistakes**. That is all that we want to configure in the **Connection String**. We can make it more **complicated**, but it is recommended that we will keep things **simple**. Now we have the database name **“Bulky”** that we want to create inside our particular server that we are able to connect in SQL Server and the trusted connection is true, that way when we connect into our **MSSQL** server with the **default settings**, optional is to add **Authentication** and we can add **Username** and **Password**.

**Creating DbContext with Entity Framework.**

**Now how can we use this connection string to actually create the database and create our Category table inside the Database?** For that we will be using **Entity Framework Core** and we will have to create an **object** of the **DbContext**. Using that **DbContext** we will be able to make **connection to the Database**. So, how do we do that? It is best to create a **New Folder** for all the data related changes. Right click our solution **BulkyBookWeb** and inside Add click “**New Folder**” and name it **Data** and inside there we will create our **DBcontext**, right click the **Data Folder** and click “**Add/Class**” and name it **ApplicationDbContext**, it should create a class named **ApplicationDbContext.cs.** We can use any name that we want here, but we used **ApplicationDbContext** so that it’s easy to identify. Now we need to **Inherit** this class file from the **DbContext(Class from Entity Framework)** that is inside our **Entity Framework Core**. In our current project we still did not add the Entity Framework Core right now. There are two ways to add that, first is we know that this will **Inherit** from the **DbContext**, so we can type that and we’ll see the red squiggly lines on there.

namespace BulkyBookWeb.Data

{

public class ApplicationDBContext : DbContext

{

}

}

If we press “**Ctrl + .** “ here, inside the suggestions we can see

**Install package ‘Microsoft.EntityFrameworkCore’** this will automatically find and install the latest version. Now second method is to do this our self, by right clicking the Project **BulkyBookWeb**  and click **Manage NuGet Packages** we can also see the **Manage NuGet Packages** when we go to tools. Now we can open this and then go to the **Browse** tab to see all the **NuGet packages** that are available. If we choose earlier the **preview** version of this **.NET** example **.NET 6.0(Preview)** we may check the checkbox “**Include prerelease**”

Now search and select **Microsoft.EntityFrameworkCore**, at the right side check our project name BulkyBookWeb and must match the version of .NET number that we use when we first created our project. In that case we used **.NET 6** so we need to select the **version 6.0.0** then we click **Install**. Once that is installed, if we go back to our **ApplicationDbContext.cs**, and now if we press “**CTRL+.”** here, it will tell us that we just have to add the USING statement **(using Microsoft.EntityFrameworkCore;)** since we have already installed the package. So we will add the using statement for our **DbContext**.

using Microsoft.EntityFrameworkCore;

namespace BulkyBookWeb.Data

{

public class ApplicationDBContext : DbContext

{

}

}

Once we did that, then there is one line of configuration that we have to do inside the constructor of this class file. You can think of that as the general syntax that is needed to **Establish the connection** with **Entity Framework**. So first, we need to create a **Constructor(“ctor” code snippet)**, once you press the **TAB twice** it should automatically create the **constructor.**

using Microsoft.EntityFrameworkCore;

namespace BulkyBookWeb.Data

{

public class ApplicationDBContext : DbContext

{

public ApplicationDBContext()

{

}

}

}

We just have to write some parameters here, because when we get the **DbContext**, we need to pass that onto the **base class** which is **DbContext**. So here, we will have to configure the **DbContextOptions** on the class that we are on right now, which is **ApplicationDbContext** and call this **options**.

using Microsoft.EntityFrameworkCore;

namespace BulkyBookWeb.Data

{

public class ApplicationDBContext : DbContext

{

public ApplicationDBContext(DbContextOptions<ApplicationDBContext> options) :base(options)

{

}

}

}

So here we are saying that in the constructor here, we will receive some options and those options, we just have to pass to the base class which is **DbContext**. This is a general setup that you have to do that will configure our **DbContext**. Now once we configure our DbContext, we still have one main feature, we still have to create our **Category table** inside our **Database.** So whatever Models that we have to create inside the **Database**, you will have to create a **DbSet** inside the **ApplicationDbContext** the file that we are currently working on.

Now how do you create a **DbSet**?

using BulkyBookWeb.Models;

using Microsoft.EntityFrameworkCore;

namespace BulkyBookWeb.Data

{

public class ApplicationDBContext : DbContext

{

public ApplicationDBContext(DbContextOptions<ApplicationDBContext> options) :base(options)

{

}

public DbSet<Category> Categories { get; set; }

}

}

Above we will say public **DbSet** here, and we need to write the **Model** name **Category**. When we see the red squiggly lines, because it cannot find anything with the name of **Category** in the same folder or file. To fix this, just **hover mouse cursor to the error** and “**CTRL+.”** to use the using statement

**“using BulkyBookWeb.Models” .** Next **parameter** to add is the table name which is “**Categories**” so that inside our **database** the table that will be created will called as “**Categories**” **not Category**. And we will add the **Getter** and the **Setter** “**{ get; set; }**”. That’s all that we had to do to create the **category table** with the name of **Categories** and it will have **four columns** that we have wrote inside the **Category Model (Category.cs).** When it creates that table, it will make sure that the **ID** is an **identity column** and N**ame** is **Required field as well**. So, you can see it is doing all the configuration by writing just few lines of code.

Now, when we are working with **Entity Framework core**, there are **two models**, one is **Code First** and the other one is **Database first**, what we are doing is **code first**, because here we are writing the **code of our model and based on that model**, we will be **creating the database**. So, that is the **Code First Approach**. **Database first approach** will be something right **database is already created** and base of that **Database**, you will be **scaffolding Models (match the database already created).** So with that we have added our DbSet for the Category table public DbSet<Category> Categories { get; set; } , But we are missing one small configuration or application still does not know that it has to use the **Connection String** that we wrote in **appsettings.json**. And it’s still does not know that it has to use **ApplicationDbContext.cs** to create a **DbContext** and that it has to work with **SQL Server**. So, let’s see how we can pull everything together. We just need to tell our application that it has to use the **DbContext** which has inside **ApplicationDbContext** and then, it has to use a **SQL Server** using the **Connection String** that were defined inside the **appsettings.json**. We will tell our application to do that inside our **Program.cs** where we configure the services that our application will use.

using System.ComponentModel.DataAnnotations;

using System.Diagnostics;

namespace BulkyBookWeb.Models

{

public class Category

{

[Key]

public int Id { get; set; }

[Required]

public string Name { get; set; }

public int DisplayOrder { get; set; }

public DateTime CreatedDateTime { get; set; } = DateTime.Now;

}

}

So here we have the comments to add the services to the container, make sure you always do that before we build the builder.

So right here, we want to add a new service.

using BulkyBookWeb.Data;

using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllersWithViews();

builder.Services.AddDbContext<ApplicationDBContext>(options => options.UseSqlServer(

builder.Configuration.GetConnectionString("DefaultConnection")

));

var app = builder.Build();

The service that we want to add is **DbContext**. So we can see we have **.AddDbContext** and that expects a **class file**, the class file that we are using for **DbContext** is the **ApplicationDbContext**. If we press **CTRL + .** here ApplicationDbContext we will just have to add the **Using Statement (using BulkyBookWeb.Data;)** of the **Data** Folder. Now, when we configure this **DbContext**, if we go back to the **ApplicationDbContext.cs**, right here, we are passing the **options** and we are sending those **options** to the **base class**.

// Add services to the container.

builder.Services.AddControllersWithViews();

--Put Here--

var app = builder.Build();

**ApplicationDbContext.cs**

using BulkyBookWeb.Models;

using Microsoft.EntityFrameworkCore;

namespace BulkyBookWeb.Data

{

public class ApplicationDBContext : DbContext

{

public ApplicationDBContext(DbContextOptions<ApplicationDBContext> options) :base(options)

{

}

public DbSet<Category> Categories { get; set; }

}

}

So inside options, we have to configure use of **SQL Server** and **Connection String**. So above we will say options => options.UseSqlServer( as we have noticed we can see the red squiggly line, it is not available like that, even if you press **CTRL + .** it will not give you the package name that you have to add. So to fix that, we will have to go into our **Manage NuGet Packages,** search and install the **Microsoft.Entity.FrameworkCore.SqlServer.** Make sure we use the consistent version, since we are using **.NET 6** we need to install the version **6.0.0 of EF**. If we install the wrong version of EF then things won’t work and we will run into error message. So always make sure that we are using the same version. Now once we press the **CTRL + .** again here options => options.UseSqlServer( it will now use the using using Microsoft.EntityFrameworkCore; displayed above. Next in this SQL Server we need to write the **Connection String ,** so where exactly our Connection String? It is inside our **appsettings.json,** we used a special block with the name of **ConnectionStrings**  "ConnectionStrings": { , since we put our connection string inside this special block thing, we can directly use the key value here **“DefaultConnection”** to extract the **connection string**.

"AllowedHosts": "\*",

"ConnectionStrings": {

"DefaultConnection": "Server:LANCEGAMINGPC\\SQLEXPRESS2k14;Database=Bulky;Trusted\_Connection:True;"

Now go back to our **Program.cs** and right here on **builder.Configuration**, we have an existing method that is provided which is to **GetConnectionString.** Inside this method we have to pass the string name which is “DefaultConnection”. Once we used that, it will automatically find the **Connection String** and configure our **SQL Server**.

using BulkyBookWeb.Data;

using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllersWithViews();

builder.Services.AddDbContext<ApplicationDBContext>(options => options.UseSqlServer(

builder.Configuration.GetConnectionString("DefaultConnection")

));

var app = builder.Build();

Now this “.GetConnectionString” is a special method, and this method will only look for "DefaultConnection" inside a block with the name of “**ConnectionStrings”** in **appsettings.json**.

"AllowedHosts": "\*",

"ConnectionStrings": {

"DefaultConnection": "Server:LANCEGAMINGPC\\SQLEXPRESS2k14;Database=Bulky;Trusted\_Connection:True;"

If you named this connection string to "ConnectionStrings1" , then it will not be able to find this **Connection String**, because this **method** will only look inside the **block**, which is "ConnectionStrings". So with this, our **DbContext** will be configured with the **Connection String**. So all the configuration is done that was needed for the **DbContext**.

**Creating Database and Tables with Entity Framework.**

Now we are on the final step, where we have to create **Database**, and then the table inside **SQL Server**. Now that the **program.cs** has been configured, the next step that we have is to actually create the database and the table. When you are using Entity Framework Code First, there are **migrations** that you have to run using **Entity Framework** to push the changes to database. It is not as complex as it sounds, we have done all the setup that is needed. In order to run the **migration**, you will go to **Tools NuGet Package Manager**, and this time, we will select the **Package Manager Console**. The first thing that we have to do is we have to add a migration. **Migration** is basically **keeping a track of all the DB changes** that are needed. And once that migration is created, you push that migration to the database to actually **create the database** or **make change to our table**. The command to add a migration is “**add-migration**” and then we have to give our migration a **meaningful** name. Do not use any spaces when we are writing the name. The name that we want to give to our migration is **AddCategoryToDatabase**

“**add-migration AddCategoryToDatabase**” now we see an **error**, because when we run the **“add-migration**”, we have to add new **NuGet Package**, we can see the term **“add-migration” was not recognized**. Now to fix this we can just install in the **NuGet Package** “**Microsoft.EntityFrameworkCore.Tools**” that is the package that is required to enable migrations in our project, make sure that the version we selected is consistent we always choose **version 6.0.0** this time because this is the **version we use since we created the application**. Now it should finished with a success, now if you notice on the right hand side here, a new folder with the name **Migrations** was created and there are two files that are added. Our migration file is the **first one**. Inside migrations we have two methods the **Up** and **Down. “Up”** method is what needs to happen inside the migration and **Down** is if something goes down, we need to rollback the changes. So do not worry about the **Down** right now. On the **MigrationBuilder**, we have a method with the name of **CreateTable** and it is creating that table with the name of **Categories**. That is the exact name that we defined inside our **ApplicationDbContext.cs** public DbSet<Category> Categories { get; set; } . So it will create a table with the name of **Categories** and then that will have columns. The first column will be the **Id** column which will be of the type **integer** and **nullable is false**, since we said it is a **[Key]** column, it is automatically making that an **identity** column and **increment that by one** every time. The next thing is the **Name** column, and we can see the **nullable** is **false** here, if we did not add the **[Required]** attribute that **nullable would be true**. We also see that **DisplayOrder** is also a **nullable false**, but we did not add that inside **Category.cs**? The reason behind that is, it is an **integer** property and **not a string**, that is why **nullable is false**, because it is an **integer**. Lastly, we have the **CreatedDateTime** of the type **DateTime** too, and that look good.

public partial class AddCategoryToDatabase : Migration

{

protected override void Up(MigrationBuilder migrationBuilder)

{

migrationBuilder.CreateTable(

name: "Categories",

columns: table => new

{

Id = table.Column<int>(type: "int", nullable: false)

.Annotation("SqlServer:Identity", "1, 1"),

Name = table.Column<string>(type: "nvarchar(max)", nullable: false),

DisplayOrder = table.Column<int>(type: "int", nullable: false),

CreatedDateTime = table.Column<DateTime>(type: "datetime2", nullable: false)

},

constraints: table =>

{

table.PrimaryKey("PK\_Categories", x => x.Id);

});

}

protected override void Down(MigrationBuilder migrationBuilder)

{

migrationBuilder.DropTable(

name: "Categories");

}

}

If we see on the constraints, it is also adding a primary key on the Id column with a name. So migrations is exactly what we wanted. But can we take a look at the SQL that gets executed? No! What Entity Framework code does is based on this Migrations, it will **create an optimized version** of the **SQL query**, and it will automatically run that on the database, we do not have to do anything with that, what we work with is just the **Models**, we create a **Migration** and we push that to the **Database**. So once we verify that **migration is looking good**, we can just run the command and **“update-database”** and that will **push the migrations to database**. We encountered an **error** here because, in the **appsettings.json**, instead of this colon **“:”** we change this character into **“=”** equal. Now enter the command **“update-database”** again to push Migrations into our database.

.

Before

"AllowedHosts": "\*",

"ConnectionStrings": {

"DefaultConnection": "Server:LANCEGAMINGPC\\SQLEXPRESS2k14;Database=Bulky;Trusted\_Connection:True;"

}

After

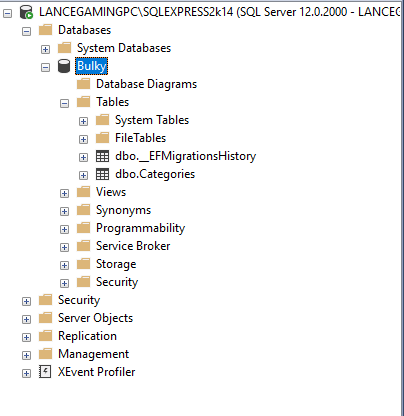
"AllowedHosts": "\*",

"ConnectionStrings": {

"DefaultConnection": "Server=LANCEGAMINGPC\\SQLEXPRESS2k14;Database=Bulky;Trusted\_Connection=True;"

}

The Migrations were completed. But what actually happened is first it will connect to the server, it will check that is there a database with the name if Bulky? If nothing, it will create that database first, and then the Migrations here, it will convert them to SQL and execute them on our database. So if we go back, now, if we refresh the database, we will see the new database with the name “**Bulky**” and if we open the tables, there will be two tables.



Now if we examine the columns inside **Categories** table it should contain four columns which is the **properties** that we had into our **Category.cs** Model. Now inside the EF Migrations if we right click and select “**Select Top 1000 Rows**”, the code keeps a track of which migrations have been applied. So next time when we run the **update-database,** it will not apply the same migration, it will only apply the migrations that have **not been applied** on this database. **Entity Framework** code is pretty smart with all the configuration and cracking that it needs to do. So with that, using Entity Framework Core, we have created our database, and we’ve added our table based on the **Model** inside our main project. Now before we work on anything else, let’s run the project, right now it is running on port 5001, now let’s change that so that the **command and prompt** will not open ever time, we can just run that using **IIS Express** profile by clicking it. That way it won’t run on the port 5001 and it will get the new port that is defined in the **launchsettings.json**. Once we run the application we can see that both of these pages Privacy and Home are inside the HomeController.

**Creating a Category Controller.**

Now in category we will be creating editing and deleting category. So for that rather than working on the home controller, let’s create a new controller for our Category. Inside our Controllers folder we will right click press “Add/Controller” now we can see few options. We will go with the empty controller to keep things very simple and start from scratch, select **MVC Controller - Empty,** now let’s name that as **CategoryController.cs.** Now always remember that if we name a controller make sure that we append controller at the end **CategoryController.cs**  . Whatever name we want, must come before the controller. So, once we have that name, we click the Add button. Now we get an empty controller with an index action method, our HomeController.cs also had the index action method, and our CategoryControler.cs also had the index action method. But the the index action method in our HomeController had a View that is inside the Home Folder. Our CategoryController does not have a View at the moment. Now we can add the View in two ways. First, we can create a folder with the name of Controller which is Category inside the folder Views and then we can add Index.cshtml view inside there. Or if we want to do that directly from the controller, we can just right click on the action Index and we have “**Add View**” then select the **Razor View**.

**CategoryController.cs**

using Microsoft.AspNetCore.Mvc;

namespace BulkyBookWeb.Controllers

{

public class CategoryController : Controller

{

public IActionResult Index()<=

{

return View();

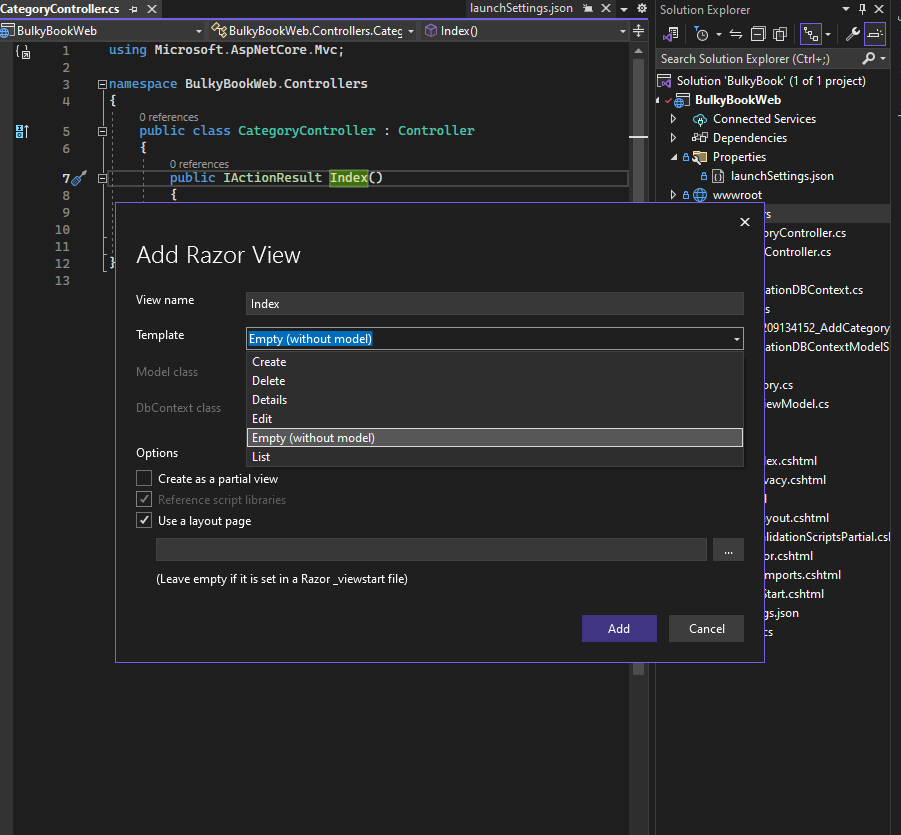
}

}

}

Right click then select “**Add View**” then select “**Razor View”**

There are a few configuration that we have to do. First, what will be the name of the View, we will give that the same name as the action method, which is Index, then do you want to use any templates when you create this View? If we hover, here we can create a View for create.



If we change that, then Model and DbContext class will be enabled. But we don’t want to focus on that right now, we want to start from scratch with an empty template. Next is a partial view, Partial View is basically like User Controls in the platform, so it will be rendered inside some other view. So if we select a partial view, the layout page will not be used. Because we do not need any master page if we create a user component, because we will be calling that inside some other view. We don’t want to take a look at that right now. Let’s keep it simple. Next, we will use the layout page, we want our view to use the default master page, so if we keep it blank it will by default use the layout that we have set inside the view start which is \_Layout.cshtml. We want it to be consistent so that it is what we will be using then we click the Add button. It will scaffold few dependencies, and then it will create the view. If we had any errors while building the project, and if we try to scaffold that will fail. It will ask you to resolve the errors, and only once our project is building successfully, we will be able to scaffold the Views. Now our View has been created, then lets call this Index Category.

@{

ViewData["Title"] = "Index";

}

<h1>Index Category</h1>

Stopped at 1:37:47